

TURBO C++ PROGRAMMER'S GUIDE

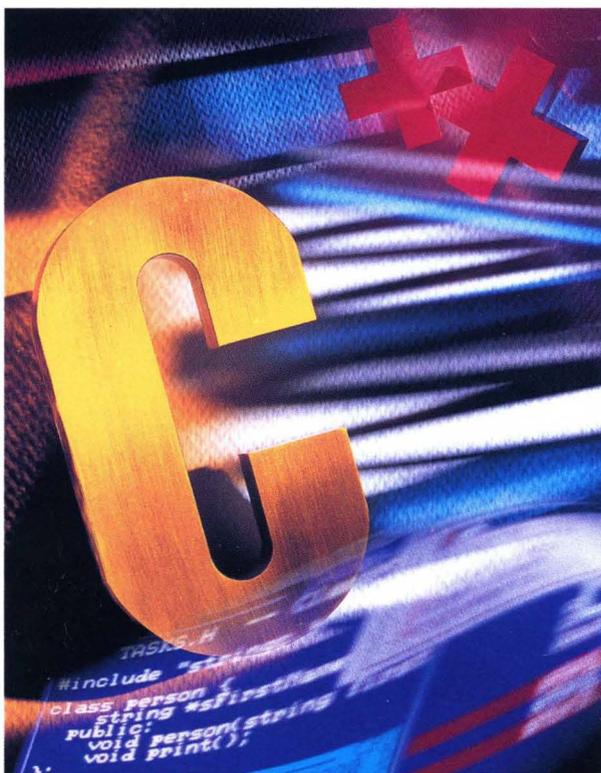
# TURBO C++

PROGRAMMER'S  
GUIDE

■ LANGUAGE GRAMMAR ■ LIBRARY CROSS REFERENCE ■  
■ ADVANCED TOPICS ■ ERROR MESSAGES ■

**B O R L A N D**

**B O R L A N D**



*Turbo C<sup>®</sup>++*

---

## Programmer's Guide

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD  
P.O. BOX 660001, SCOTTS VALLEY, CA 95066-0001

This manual was produced with Sprint®: The Professional Word Processor

---

Copyright © 1990 by Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

# C O N T E N T S

---

<b>Introduction</b> .....	1	String literals .....	17
Contents of this manual .....	1	Constants and internal representation .....	18
<b>Chapter 1 The Turbo C++ language standard</b> .....	3	Constant expressions .....	19
Syntax and terminology .....	4	Operator descriptions .....	20
Lexical and phrase-structure grammars ..	4	Unary operators .....	21
Whitespace .....	5	Binary operators .....	22
Line splicing with \ .....	5	Additive operators .....	22
Comments .....	6	Multiplicative operators .....	22
C comments .....	6	Shift operators .....	22
Nested comments .....	6	Bitwise operators .....	22
C++ comments .....	7	Logical operators .....	22
Comment delimiters and whitespace ..	7	Assignment operators .....	22
Tokens .....	7	Relational operators .....	22
Keywords .....	8	Equality operators .....	23
Identifiers .....	9	Component selection operators ....	23
Naming and length restrictions .....	9	Class-member operators .....	23
Identifiers and case sensitivity .....	9	Conditional operator .....	23
Uniqueness and scope .....	10	Comma operator .....	23
Constants .....	10	Punctuators .....	23
Integer constants .....	11	Brackets .....	23
Decimal constants .....	12	Parentheses .....	23
Octal constants .....	12	Braces .....	24
Hexadecimal constants .....	12	Comma .....	24
Long and unsigned suffixes .....	12	Semicolon .....	24
Character constants .....	13	Colon .....	25
Escape sequences .....	13	Ellipsis .....	25
Turbo C++ special two-character constants .....	15	Asterisk (pointer declaration) .....	25
Signed and unsigned char .....	15	Equal sign (initializer) .....	26
Wide character constants (C only) ..	15	Pound sign (preprocessor directive) ..	26
Floating-point constants .....	16	Declarations .....	26
Floating point constants—data types .....	16	Objects .....	27
Enumeration constants .....	17	Lvalues .....	28
		Rvalues .....	28
		Types and storage classes .....	28
		Scope .....	29

Block scope .....	29	The pointer modifiers .....	51
Function scope .....	29	Function type modifiers .....	51
Function prototype scope .....	29	Complex declarations and declarators ..	52
File scope .....	29	Pointers .....	53
Class scope (C++) .....	29	Pointers to objects .....	54
Scope and name spaces .....	29	Pointers to functions .....	54
Visibility .....	30	Pointer declarations .....	55
Duration .....	31	Pointers and constants .....	56
Static duration .....	31	Pointer arithmetic .....	57
Local duration .....	31	Pointer conversions .....	58
Dynamic duration .....	32	C++ reference declarations .....	58
Translation units .....	32	Arrays .....	58
Linkage .....	32	Functions .....	59
Declaration syntax .....	33	Declarations and definitions .....	59
Tentative definitions .....	34	Declarations and prototypes .....	60
Possible declarations .....	34	Definitions .....	62
External declarations and definitions ..	36	Formal parameter declarations .....	62
Type specifiers .....	38	Function calls and argument	
Type taxonomy .....	38	conversions .....	63
Type void .....	39	Structures .....	64
The fundamental types .....	39	Untagged structures and typedefs ....	65
Integral types .....	40	Structure member declarations .....	65
Floating-point types .....	41	Structures and functions .....	66
Standard conversions .....	41	Structure member access .....	66
Special char, int, and enum		Structure word alignment .....	67
conversions .....	42	Structure name spaces .....	68
Initialization .....	42	Incomplete declarations .....	68
Arrays, structures, and unions .....	43	Bit fields .....	69
Simple declarations .....	44	Unions .....	70
Storage class specifiers .....	45	Union declarations .....	71
Use of storage class specifier <b>auto</b> ..	45	Enumerations .....	71
Use of storage class specifier <b>extern</b> .	45	Expressions .....	73
Use of storage class specifier		Expressions and C++ .....	75
<b>register</b> .....	45	Evaluation order .....	76
Use of storage class specifier <b>static</b> .	45	Errors and overflows .....	77
Use of storage class specifier		Operator semantics .....	77
<b>typedef</b> .....	46	Postfix and prefix operators .....	77
Modifiers .....	46	Array subscript operator [ ] .....	77
The const modifier .....	48	Function call operators ( ) .....	78
The interrupt function modifier ....	48	Structure/union member operator	
The volatile modifier .....	48	. (dot) .....	78
The cdecl and pascal modifiers .....	49	Structure/union pointer operator	
pascal .....	49	-> .....	78
cdecl .....	50	Postfix increment operator ++ .....	78

Postfix decrement operator --	79	Selection statements	93
Increment and decrement operators	79	if statements	93
Prefix increment operator	79	switch statements	94
Prefix decrement operator	79	Iteration statements	95
Unary operators	79	while statements	95
Address operator &	80	do while statements	95
Indirection operator *	80	for statements	95
Unary plus operator +	81	Jump statements	96
Unary minus operator -	81	break statements	97
Bitwise complement operator ~	81	continue statements	97
Logical negation operator !	81	goto statements	97
The sizeof operator	81	return statements	97
Multiplicative operators	82	C++	98
Additive operators	83	Referencing	98
The addition operator +	83	Simple references	99
The subtraction operator -	83	Reference arguments	99
Bitwise shift operators	83	Scope access operator	100
Bitwise left-shift operator <<	84	The new and delete operators	101
Bitwise right-shift operator >>	84	The operator new with arrays	102
Relational operators	84	The ::operator new	102
The less-than operator <	85	Initializers with the new operator	102
The greater-than operator >	85	Classes	102
The less-than or equal-to operator		Class names	103
<=	85	Class types	103
The greater-than or equal-to operator		Class name scope	103
>=	86	Class objects	104
Equality operators	86	Class member list	104
The equal-to operator ==	86	Member functions	105
The inequality operator !=	87	The keyword <b>this</b>	105
Bitwise AND operator &	87	Inline functions	105
Bitwise exclusive OR operator ^	87	Static members	106
Bitwise inclusive OR operator	88	Member scope	107
Logical AND operator &&	88	Member access control	108
Logical OR operator	88	Base and derived class access	110
Conditional operator ?:	89	Virtual base classes	112
Assignment operators	90	Friends of classes	112
The simple assignment operator =	90	Constructors and destructors	114
The compound assignment		Constructors	115
operators	90	The default constructor	115
Comma operator ,	91	The copy constructor	116
Statements	91	Overloading constructors	117
Blocks	92	Order of calling constructors	117
Labeled statements	92	Class initialization	119
Expression statements	93	Destructors	121

When destructors are invoked . . . .	122	The #ifdef and #ifndef conditional directives . . . . .	143
<b>atexit</b> , <b>#pragma exit</b> , and destructors . . . . .	122	The #line line control directive . . . . .	144
<b>exit</b> and destructors . . . . .	122	The #error directive . . . . .	145
<b>abort</b> and destructors . . . . .	122	The #pragma directive . . . . .	146
Virtual destructors . . . . .	123	#pragma argsused . . . . .	146
Overloaded operators . . . . .	124	#pragma exit and #pragma startup . . . . .	146
Operator functions . . . . .	125	#pragma inline . . . . .	147
Overloaded operators and inheritance . . . . .	125	#pragma option . . . . .	148
Overloading <b>new</b> and <b>delete</b> . . . . .	125	#pragma saveregs . . . . .	149
Overloading unary operators . . . . .	127	#pragma warn . . . . .	150
Overloading binary operators . . . . .	127	Predefined macros . . . . .	150
Overloading the assignment operator = . . . . .	127	__CDECL__ . . . . .	150
Overloading the function call operator () . . . . .	128	__cplusplus . . . . .	151
Overloading the subscripting operator [ ] . . . . .	128	__DATE__ . . . . .	151
Overloading the class member access operator -> . . . . .	128	__FILE__ . . . . .	151
Virtual functions . . . . .	128	__LINE__ . . . . .	151
Abstract classes . . . . .	130	__MSDOS__ . . . . .	151
C++ scope . . . . .	131	__OVERLAY__ . . . . .	152
Class scope . . . . .	131	__PASCAL__ . . . . .	152
Hiding . . . . .	132	__STDC__ . . . . .	152
C++ scoping rules summary . . . . .	132	__TIME__ . . . . .	152
Turbo C++ preprocessor directives . . . . .	133	__TURBOC__ . . . . .	152
Null directive # . . . . .	135	<b>Chapter 2 Run-time library cross-reference</b> . . . . .	153
The #define and #undef directives . . . . .	135	Reasons to access the run-time library source code . . . . .	154
Simple #define macros . . . . .	135	The Turbo C++ header files . . . . .	155
The #undef directive . . . . .	136	Library routines by category . . . . .	157
The -D and -U options . . . . .	137	Classification routines . . . . .	157
Keywords and protected words . . . . .	137	Conversion routines . . . . .	158
Macros with parameters . . . . .	137	Directory control routines . . . . .	158
File inclusion with #include . . . . .	140	Diagnostic routines . . . . .	158
Header file search with <header_name> . . . . .	141	Graphics routines . . . . .	158
Header file search with "header_name" . . . . .	141	Input/output routines . . . . .	159
Conditional compilation . . . . .	142	Interface routines (DOS, 8086, BIOS) . . . . .	160
The #if, #elif, #else, and #endif conditional directives . . . . .	142	Manipulation routines . . . . .	161
The operator defined . . . . .	143	Math routines . . . . .	161
		Memory routines . . . . .	162
		Miscellaneous routines . . . . .	163
		Process control routines . . . . .	163
		Standard routines . . . . .	163
		Text window display routines . . . . .	163

Time and date routines .....	164	Memory segmentation .....	190
Variable argument list routines .....	164	Address calculation .....	191
<b>Chapter 3 C++ streams</b> .....	165	Pointers .....	192
New streams for old .....	165	Near pointers .....	192
Using the 2.0 streams .....	165	Far pointers .....	192
What's a stream? .....	166	Huge pointers .....	193
The iostream library .....	166	The six memory models .....	194
streambuf .....	167	Mixed-model programming: Addressing	
ios .....	167	modifiers .....	199
The stream classes .....	167	Declaring functions to be near or far ..	200
The four standard streams .....	168	Declaring pointers to be near, far, or	
Output .....	168	huge .....	201
Built-in types .....	169	Pointing to a given segment:offset	
The <b>put</b> and <b>write</b> functions .....	170	address .....	202
Output formatting .....	170	Using library files .....	203
Conversion base .....	171	Linking mixed modules .....	203
Width .....	171	Floating-point options .....	204
Manipulators .....	172	Emulating the 80x87 chip .....	205
Filling and padding .....	174	Using 80x87 code .....	205
User-defined inserters .....	174	No floating-point code .....	205
Input .....	175	Fast floating-point option .....	205
Chaining extractors .....	175	The 87 environment variable .....	206
Extractors for built-in types .....	176	Registers and the 80x87 .....	207
Integral extractors .....	176	Disabling floating-point exceptions ..	207
Floating-point extractors .....	176	Using complex math .....	208
Character extractors .....	176	Using BCD math .....	209
Putback function .....	177	Converting BCD numbers .....	210
User-defined types for input .....	178	Number of decimal digits .....	210
Initializing streams .....	178	Turbo C++'s use of RAM .....	211
Simple file I/O .....	179	Overlays (VROOMM) .....	211
I/O stream error states .....	181	How overlays work .....	212
Using the older streams .....	183	Getting the best out of Turbo C++	
Guidelines for upgrading to 2.0		overlays .....	213
streams .....	184	Requirements .....	214
<b>Chapter 4 Memory models, floating</b>		Using overlays .....	214
<b>point, and overlays</b> .....	187	Overlay example .....	215
Memory models .....	187	Overlaying in the IDE .....	215
The 8086 registers .....	187	Designing overlaid programs .....	216
General-purpose registers .....	188	The far call requirement .....	216
Segment registers .....	189	Buffer size .....	216
Special-purpose registers .....	189	What not to overlay .....	217
The flags register .....	189	Debugging overlays .....	217
		External routines in overlays .....	217
		Swapping .....	218

Expanded memory .....	219
Extended memory .....	219
<b>Chapter 5 Video functions</b> .....	<b>221</b>
Some words about video modes .....	221
Some words about windows and viewports .....	222
What is a window? .....	222
What is a viewport? .....	223
Coordinates .....	223
Programming in text mode .....	223
The console I/O functions .....	223
Text output and manipulation ....	224
Window and mode control .....	225
Attribute control .....	225
State query .....	226
Cursor shape .....	227
Text windows .....	227
An example .....	227
The <i>text_modes</i> type .....	228
Text colors .....	229
High-performance output: The <i>directvideo</i> variable .....	230
Programming in graphics mode .....	230
The graphics library functions .....	231
Graphics system control .....	231
A more detailed discussion .....	233
Drawing and filling .....	234
Manipulating the screen and viewport .....	236
Text output in graphics mode ....	237
Color control .....	239
Pixels and palettes .....	239
Background and drawing color ...	240
Color control on a CGA .....	240
CGA low resolution .....	240
CGA high resolution .....	242
CGA palette routines .....	242
Color control on the EGA and VGA .....	242
Error handling in graphics mode ..	243
State query .....	243

<b>Chapter 6 Interfacing with assembly language</b> .....	<b>247</b>
Mixed-language programming .....	247
Parameter-passing sequences .....	247
C parameter-passing sequence ....	248
Pascal parameter-passing sequence .....	249
Setting up to call .ASM from Turbo C++ .....	251
Simplified segment directives .....	251
Standard segment directives .....	251
Defining data constants and variables .....	253
Defining global and external identifiers .....	253
Setting up to call Turbo C++ from .ASM .....	255
Referencing functions .....	255
Referencing data .....	255
Defining assembly language routines ..	256
Passing parameters .....	256
Handling return values .....	257
Register conventions .....	261
Calling C functions from .ASM routines .....	262
Pseudovariables, inline assembly, and interrupt functions .....	264
Pseudovariables .....	265
Inline assembly language .....	267
Opcodes .....	270
String instructions .....	271
Prefixes .....	272
Jump instructions .....	272
Assembly directives .....	272
Inline assembly references to data and functions .....	272
Inline assembly and register variables .....	273
Inline assembly, offsets, and size overrides .....	273
Using C structure members .....	273
Using jump instructions and labels ..	274
Interrupt functions .....	275

Using low-level practices .....	276	Warnings .....	318
<b>Chapter 7 Error messages</b>	<b>279</b>	<b>Appendix A ANSI implementation-</b>	
Run-time error messages .....	280	<b>specific standards</b>	<b>327</b>
Compiler error messages .....	283	<b>Index</b>	<b>341</b>
Fatal errors .....	284		
Errors .....	284		

# T A B L E S

---

1.1: All Turbo C++ keywords .....	8	1.19: Turbo C++ expressions .....	74
1.2: Turbo C++ extensions to ANSI C .....	8	1.20: Associativity and precedence of Turbo C++ operators .....	75
1.3: Keywords specific to C++ .....	8	1.21: Bitwise operators truth table .....	87
1.4: Turbo C++ register pseudovariables ..	9	1.22: Turbo C++ statements .....	92
1.5: Constants—formal definitions .....	11	1.23: Turbo C++ preprocessing directives syntax .....	134
1.6: Turbo C++ integer constants without L or U .....	13	3.1: Manipulators .....	173
1.7: Turbo C++ escape sequences .....	15	3.2: <b>ios</b> error bits .....	182
1.8: Turbo C++ floating constant sizes and ranges .....	16	3.3: Current stream state member functions .....	182
1.9: Data types, sizes, and ranges .....	19	4.1: Memory models .....	198
1.10: Turbo C++ declaration syntax .....	35	4.2: Pointer results .....	200
1.11: Turbo C++ declarator syntax .....	36	6.1: Assembly language file format .....	252
1.12: Turbo C++ class declarations (C++ only) .....	37	6.2: Identifier replacements and memory models .....	252
1.13: Declaring types .....	39	6.3: Pseudovariables .....	266
1.14: Integral types .....	40	6.4: Opcode mnemonics .....	271
1.15: Methods used in standard arithmetic conversions .....	42	6.5: String instructions .....	272
1.16: Turbo C++ modifiers .....	47	6.6: Jump instructions .....	272
1.17: Complex declarations .....	53	A.1: Identifying diagnostics in Turbo C++ .....	327
1.18: External function definitions .....	62		

# F I G U R E S

---

1.1: Internal representations of data types .19	
4.1: 8086 registers .....188	
4.2: Flags register of the 8086 .....189	
4.3: Tiny model memory segmentation ..195	
4.4: Small model memory segmentation .196	
4.5: Medium model memory segmentation .....196	
4.6: Compact model memory segmentation .....197	
4.7: Large model memory segmentation .197	
4.8: Huge model memory segmentation .198	
4.9: Memory maps for overlays .....213	
5.1: A window in 80x25 text mode .....228	



*Getting Started* provides an overview of the entire Turbo C++ documentation set. Read the introduction and Chapter 2 in that book for information on how to most effectively use the Turbo C++ manuals.

This manual contains materials for the advanced programmer. If you already know how to program well (whether in C or another language), this manual is for you. It provides a language reference, a cross-reference to the run-time library, and programming information on the C++ streams, memory models, floating point, overlays, video functions, assembly language interfacing, and the run-time and compiler error messages.

Read *Getting Started* if:

1. You have never programmed in any language.
2. You have programmed, but not in C, and you would like an introduction to the C language.
3. You are looking for information on how to install Turbo C++.

Use the *User's Guide* for reference information on the Turbo C++ integrated environment (including the editor), the project manager, the command-line compiler, utilities that come with Turbo C++, and the Turbo Editor Macro Language.

The *Library Reference* contains an alphabetical listing of all of Turbo C++'s functions and global variables.

## Contents of this manual

---

**Chapter 1: The Turbo C++ language standard** describes the Turbo C++ language. Any differences from the ANSI C standard are noted here. This chapter includes a language reference and syntax for C and C++.

**Chapter 2: Run-time library cross-reference** provides some information on the source code for the run-time library, lists and describes the header files, and provides a cross-reference to the

run-time library, organized by subject. For example, if you want to find out which functions relate to graphics, you would look in this chapter under the topic "Graphics."

**Chapter 3: C++ streams** tells you how to use the C++ streams library.

**Chapter 4: Memory models, floating point, and overlays** covers memory models, mixed-model programming, floating-point concerns, and overlays.

**Chapter 5: Video functions** is devoted to handling text and graphics in Turbo C++.

**Chapter 6: Interfacing with assembly language** tells how to write assembly language programs so they work well when called from Turbo C++ programs.

**Chapter 7: Error messages** lists and explains all run-time and compiler-generated fatal errors, errors, and warnings, and suggests possible solutions.

**Appendix A: ANSI implementation-specific standards** describes those aspects of the ANSI C standard that have been loosely defined or undefined by ANSI. These aspects will vary, then, according to each implementation. This appendix tells how Turbo C++ operates with respect to each of these aspects.

## *The Turbo C++ language standard*

This chapter provides a detailed programmer's reference guide to the Turbo C++ language. It is not a language tutorial, but rather a formal description of the C and C++ languages as implemented in Turbo C++. The chapter provides both lexical and phrase structure grammars, together with details of the preprocessor directives available. We've used a modified Backus-Naur notation to indicate syntax, supplemented where necessary by brief explanations and program examples.

Turbo C++ implements the ANSI C standard developed by Technical Committee X3J11 between June 1983 and December 1988, with several extensions as indicated in the text. You can set options in the compiler to warn you if any such extensions are encountered. You can also set the compiler to treat the Turbo C++ extension keywords as normal identifiers (see Chapter 4, "The command-line compiler," in the *User's Guide*).

There are also "conforming" extensions provided via the **#pragma** directives offered by ANSI C for handling nonstandard, implementation-dependent features.

Turbo C++ is also a full implementation of AT&T's C++ version 2.0, the object-oriented superset of C developed by Bjarne Stroustrup of AT&T Bell Laboratories. In addition to offering many new features and capabilities, C++ often veers from C by small or large amounts. We've made note of these differences throughout this chapter. All the Turbo C++ language features

derived from C++ are discussed in greater detail starting on page 98.

## Syntax and terminology

---

Syntactic definitions consist of the name of the nonterminal being defined, followed by a colon (:). Alternatives usually follow on separate lines, but a single line of alternatives can be used if prefixed by the phrase "one of." For example,

*external-definition:*  
*function-definition*  
*declaration*

*octal-digit:* one of  
0 1 2 3 4 5 6 7

Optional elements in a construct are printed within angle brackets:

*integer-suffix:*  
*unsigned-suffix* <*long-suffix*>

Throughout this chapter, the word "argument" is used to mean the actual value passed in a call to a function. "Parameter" is used to mean the variable defined in the function header to hold the value.

## Lexical and phrase-structure grammars

---

*Pages 5 through 58 cover Turbo C++'s lexical grammar; pages 58 through 98 cover the elements of Turbo C++'s phrase structure grammar.*

Lexical grammar is concerned with the different categories of word-like units, known as *tokens*, recognized by a language. Phrase structure grammar details the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.

The tokens in Turbo C++ are derived from a series of operations performed on your programs by the compiler and its preprocessor.

A Turbo C++ program starts life as a sequence of ASCII characters representing the source code, created by keystrokes using a suitable text editor (such as the Turbo C++ editor). The basic program unit in Turbo C++ is the file. This usually

corresponds to a named DOS file located in RAM or on disk and having the extension .C or .CPP.

The preprocessor first scans the program text for special preprocessor *directives* (see page 133). For example, the directive **#include** *<inc\_file>* adds (or includes) the contents of the file *inc\_file* to the program before the compilation phase. The preprocessor also expands any macros found in the program and include files.

## Whitespace

---

In the tokenizing phase of compilation, the source code file is *parsed* (that is, broken down) into tokens and *whitespace*. *Whitespace* is the collective name given to spaces (blanks), horizontal and vertical tabs, newline characters, and comments. Whitespace can serve to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded. For example, the two sequences

```
int i; float f;
```

and

```
int i ;  
float f;
```

are lexically equivalent and parse identically to give the six tokens:

```
int i ; float f ;
```

The ASCII characters representing whitespace can occur within *literal strings*, in which case they are protected from the normal parsing process; in other words, they remain as part of the string:

```
char name[] = "Borland International";
```

parses to seven tokens, including the single literal-string token "Borland International".

### Line splicing with \

---

A special case occurs if the final newline character encountered is preceded by a backslash (\). The backslash and new line are both discarded, allowing two physical lines of text to be treated as one unit.

```
"Borland \  
"
```

International"

is parsed as "Borland International" (see page 17, "String literals," for more information).

## Comments

---

*Comments* are pieces of text used to annotate a program. Comments are for the programmer's use only; they are stripped from the source text before parsing.

There are two ways to delineate comments: the traditional C method and the C++ method. Both are supported by Turbo C++, with an additional, optional extension permitting nested comments. You can mix and match either kind of comment in both C and C++ programs.

**C comments** A traditional C comment is any sequence of characters placed after the symbol pair `/*`. The comment terminates at the first occurrence of the pair `*/` following the initial `/*`. The entire sequence, including the four comment delimiter symbols, is replaced by one space *after* macro expansion. Note that some C implementations remove comments without space replacements.

Turbo C++ does not support the non-portable *token pasting* strategy using `/**/`. Token pasting in Turbo C++ is performed with the ANSI-specified pair `##`, as follows:

```
#define VAR(i, j) (i/**/j) /* won't work */
#define VAR(i, j) (i##j) /* OK in Turbo C++ */
#define VAR(i, j) (i ## j) /* Also OK */
```

In Turbo C++,

```
int /* declaration */ i /* counter */;
```

parses as

```
int i ;
```

to give the three tokens: **int i ;**

**Nested comments** ANSI C doesn't allow nested comments. Attempting to comment out the preceding line with

```
/* int /* declaration */ i /* counter */; */
```

fails, since the scope of the first `/*` ends at the first `*/`. This gives

```
i ; */
```

which would generate a syntax error.

By default, Turbo C++ won't allow nested comments, but you can override this with compiler options. You can enable nested comments with the `-C` option (the command-line compiler) or via the **O|C|Source Options** menu in the integrated environment.

C++ comments  
You can also use `//` to create comments in C code. This is specific to Turbo C++.

C++ allows a single-line comment using two adjacent slashes (`//`). The comment can start in any position, and extends until the next new line:

```
class X { // this is a comment
... };
```

Comment delimiters  
and whitespace

In rare cases, some whitespace before `/*` and `//`, and after `*/`, although not syntactically mandatory, can avoid portability problems. For example, this C++ code

```
int i = j//*/ divide by k*/k;
+m;
```

parses as `int i = j +m;` not as

```
int i = j/k;
+m;
```

as expected under the traditional C convention. The more legible

```
int i = j/ /* divide by k*/ k;
+m;
```

avoids this problem.

## Tokens

---

Turbo C++ recognizes six classes of tokens: *keywords*, *identifiers*, *constants*, *string-literals*, *operators*, and *punctuators* (also known as *separators*). The formal definition of a token is as follows:

*token:*  
*keyword*  
*identifier*  
*constant*  
*string-literal*  
*operator*  
*punctuator*

As the source code is parsed, tokens are extracted in such a way that the longest possible token from the character sequence is selected. For example, **external** would be parsed as a single identifier, rather than as the keyword **extern** followed by the identifier *al*.

## Keywords

**Keywords** are words reserved for special purposes and must not be used as normal identifier names. The following two tables list the Turbo C++ keywords. You can use command-line compiler options (or options in the IDE) to select ANSI keywords only, UNIX keywords, and so on. See chapters 1, "The IDE reference," and 4, "The command-line compiler," both in the *User's Guide*, for information on these options.

Table 1.1  
All Turbo C++ keywords

<b>asm</b>	<b>_ds</b>	<b>interrupt</b>	<b>short</b>
<b>auto</b>	<b>else</b>	<b>_loadds</b>	<b>signed</b>
<b>break</b>	<b>enum</b>	<b>long</b>	<b>sizeof</b>
<b>case</b>	<b>_es</b>	<b>near</b>	<b>_ss</b>
<b>catch</b>	<b>_export</b>	<b>new</b>	<b>static</b>
<b>cdecl</b>	<b>extern</b>	<b>operator</b>	<b>struct</b>
<b>char</b>	<b>far</b>	<b>pascal</b>	<b>switch</b>
<b>class</b>	<b>float</b>	<b>private</b>	<b>template</b>
<b>const</b>	<b>for</b>	<b>protected</b>	<b>this</b>
<b>continue</b>	<b>friend</b>	<b>public</b>	<b>typedef</b>
<b>_cs</b>	<b>goto</b>	<b>register</b>	<b>union</b>
<b>default</b>	<b>huge</b>	<b>_regparam</b>	<b>unsigned</b>
<b>delete</b>	<b>if</b>	<b>return</b>	<b>virtual</b>
<b>do</b>	<b>inline</b>	<b>_saveregs</b>	<b>void</b>
<b>double</b>	<b>int</b>	<b>_seg</b>	<b>volatile</b>
			<b>while</b>

Table 1.2  
Turbo C++ extensions to ANSI C

<b>cdecl</b>	<b>_export</b>	<b>_loadds</b>	<b>_saveregs</b>
<b>_cs</b>	<b>far</b>	<b>near</b>	<b>_seg</b>
<b>_ds</b>	<b>huge</b>	<b>pascal</b>	<b>_ss</b>
<b>_es</b>	<b>interrupt</b>	<b>_regparam</b>	

Table 1.3  
Keywords specific to C++

<b>catch</b>	<b>friend</b>	<b>operator</b>	<b>public</b>
<b>class</b>	<b>inline</b>	<b>private</b>	<b>template</b>
<b>delete</b>	<b>new</b>	<b>protected</b>	<b>this</b>
			<b>virtual</b>

Table 1.4  
Turbo C++ register  
pseudovariabes

<b>_AH</b>	<b>_BL</b>	<b>_CL</b>	<b>_DL</b>
<b>_AL</b>	<b>_BP</b>	<b>_CX</b>	<b>_DX</b>
<b>_AX</b>	<b>_BX</b>	<b>_DH</b>	<b>_FLAGS</b>
<b>_BH</b>	<b>_CH</b>	<b>_DI</b>	<b>_SI</b>
			<b>_SP</b>

## Identifiers

The formal definition of an identifier is as follows:

*identifier:*

*nondigit*

*identifier nondigit*

*identifier digit*

*nondigit:* one of

a b c d e f g h i j k l m n o p q r s t u v w x y z \_

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

*digit:* one of

0 1 2 3 4 5 6 7 8 9

### Naming and length restrictions

*Identifiers* are arbitrary names of any length given to classes, objects, functions, variables, user-defined data types, and so on. Identifiers can contain the letters *A* to *Z* and *a* to *z*, the underscore character (`_`), and the digits 0 to 9. There are only two restrictions:

*Note that identifiers in C++ programs are significant to any length.*

1. The first character must be a letter or an underscore.
2. By default, Turbo C++ recognizes only the first 32 characters as significant. The number of significant characters can be *reduced* by menu and command-line options, but not increased. Use the `-in` TCC option or the integrated environment's **O|C|S|Identifier Length** menu option, where  $1 \leq n \leq 32$ .

### Identifiers and case sensitivity

Turbo C++ identifiers are case sensitive, so that *Sum*, *sum*, and *suM* are distinct identifiers.

Global identifiers imported from other modules follow the same naming and significance rules as normal identifiers. However, Turbo C++ offers the option of suspending case sensitivity to allow compatibility when linking with case-insensitive languages. By checking the **Options | Linker | Case-Sensitive Link** box in the Linker dialog box, or using the `/C` command-line switch with

TLINK, you can ensure that global identifiers are *case insensitive*. Under this regime, the globals *Sum* and *sum* are considered identical, resulting in a possible "Duplicate symbol" warning during linking.

An exception to these rules is that identifiers of type **pascal** are always converted to all uppercase for linking purposes.

Uniqueness and scope Although identifier names are arbitrary (within the rules stated), errors result if the same name is used for more than one identifier within the same *scope* and sharing the same *name space*. Duplicate names are always legal for *different* name spaces regardless of scope. The rules are covered in the discussion on scope starting on page 29.

---

## Constants

*Constants* are tokens representing fixed numeric or character values. Turbo C++ supports four classes of constants: floating point, integer, enumeration, and character.

The data type of a constant is deduced by the compiler using such clues as numeric value and the format used in the source code. The formal definition of a constant is shown in the following table:

Table 1.5: Constants—formal definitions

<i>constant:</i>	0 X <i>hexadecimal-digit</i>
<i>floating-constant</i>	<i>hexadecimal-constant hexadecimal-digit</i>
<i>integer-constant</i>	<i>nonzero-digit: one of</i>
<i>enumeration-constant</i>	1 2 3 4 5 6 7 8 9
<i>character-constant</i>	<i>octal-digit: one of</i>
<i>floating-constant:</i>	0 1 2 3 4 5 6 7
<i>fractional-constant</i> < <i>exponent-part</i> > < <i>floating-suffix</i> >	<i>hexadecimal-digit: one of</i>
<i>digit-sequence</i> <i>exponent-part</i> < <i>floating-suffix</i> >	0 1 2 3 4 5 6 7 8 9
<i>fractional-constant:</i>	a b c d e f
< <i>digit-sequence</i> > . <i>digit-sequence</i>	A B C D E F
<i>digit-sequence</i> .	<i>integer-suffix:</i>
<i>exponent-part:</i>	<i>unsigned-suffix</i> < <i>long-suffix</i> >
e < <i>sign</i> > <i>digit-sequence</i>	<i>long-suffix</i> < <i>unsigned-suffix</i> >
E < <i>sign</i> > <i>digit-sequence</i>	<i>unsigned-suffix: one of</i>
<i>sign: one of</i>	u U
+ -	<i>long-suffix: one of</i>
<i>digit-sequence:</i>	l L
<i>digit</i>	<i>enumeration-constant:</i>
<i>digit-sequence digit</i>	<i>identifier</i>
<i>floating-suffix: one of</i>	<i>character-constant:</i>
f l F L	<i>c-char-sequence</i>
<i>integer-constant:</i>	<i>c-char-sequence:</i>
<i>decimal-constant</i> < <i>integer-suffix</i> >	<i>c-char</i>
<i>octal-constant</i> < <i>integer-suffix</i> >	<i>c-char-sequence c-char</i>
<i>hexadecimal-constant</i> < <i>integer-suffix</i> >	<i>c-char:</i>
<i>decimal-constant:</i>	Any character in the source character set except
<i>nonzero-digit</i>	the single-quote ('), backslash (\), or newline
<i>decimal-constant digit</i>	character <i>escape-sequence</i> .
<i>octal-constant:</i>	<i>escape-sequence: one of</i>
0	\ "    \ '    \ ?    \ \
<i>octal-constant octal-digit</i>	\ a    \ b    \ f    \ n
<i>hexadecimal-constant:</i>	\ o    \ oo    \ ooo    \ r
0 x <i>hexadecimal-digit</i>	\ t    \ v    \ Xh...    \ xh...

**Integer constants** *Integer constants* can be decimal (base 10), octal (base 8) or hexadecimal (base 16). In the absence of any overriding suffixes, the data type of an integer constant is derived from its value, as shown in Table 1.6. Note that the rules vary between decimal and nondecimal constants.

## Decimal constants

*Decimal constants* from 0 to 4,294,967,295 are allowed. Constants exceeding this limit will generate an error. Decimal constants must not use an initial zero. An integer constant that has an initial zero is interpreted as an octal constant. Thus,

```
int i = 10; /*decimal 10 */
int i = 010; /*decimal 8 */
int i = 0; /*decimal 0 = octal 0! */
```

*Negative constants* are simply unsigned constants with the unary minus operator.

## Octal constants

All constants with an initial zero are taken to be octal. If an octal constant contains the illegal digits 8 or 9, an error is reported. Octal constants exceeding 037777777777 will generate an error.

## Hexadecimal constants

All constants starting with 0x (or 0X) are taken to be hexadecimal. Hexadecimal constants exceeding 0xFFFFFFFF will generate an error.

## Long and unsigned suffixes

The suffix *L* (or *l*) attached to any constant forces it to be represented as a **long**. Similarly, the suffix *U* (or *u*) forces the constant to be **unsigned**. It is **unsigned long** if the value of the number itself is greater than decimal 65,535, regardless of which base is used. You can use both *L* and *U* suffixes on the same constant in any order or case: *ul*, *lu*, *UL*, and so on.

The data type of a constant in the absence of any suffix (*U*, *u*, *L*, or *l*) is the first of the following types that can accommodate its value:

---

decimal	<b>int, long int, unsigned long int</b>
octal	<b>int, unsigned int, long int, unsigned long int</b>
hexadecimal	<b>int, unsigned int, long int, unsigned long int</b>

---

If the constant has a *U* or *u* suffix, its data type will be the first of **unsigned int, unsigned long int** that can accommodate its value.

If the constant has an *L* or *l* suffix, its data type will be the first of **long int**, **unsigned long int** that can accommodate its value.

If the constant has both *u* and *l* suffixes (*ul*, *lu*, *Ul*, *lU*, *uL*, *Lu*, *LU*, or *UL*), its data type will be **unsigned long int**.

Table 1.6 summarizes the representations of integer constants in all three bases. The data types indicated assume no overriding *L* or *U* suffix has been used.

Table 1.6  
Turbo C++ integer constants  
without L or U

Decimal constants	
0 to 32,767	<b>int</b>
32,768 to 2,147,483,647	<b>long</b>
2,147,483,648 to 4,294,967,295	<b>unsigned long</b>
> 4294967295	Generates an error.
Octal constants	
00 to 077777	<b>int</b>
0100000 to 0177777	<b>unsigned int</b>
02000000 to 01777777777	<b>long</b>
020000000000 to 03777777777	<b>unsigned long</b>
> 03777777777	Generates an error.
Hexadecimal constants	
0x0000 to 0x7FFF	<b>int</b>
0x8000 to 0xFFFF	<b>unsigned int</b>
0x10000 to 0x7FFFFFFF	<b>long</b>
0x80000000 to 0xFFFFFFFF	<b>unsigned long</b>
> 0xFFFFFFFF	Generates an error.

## Character constants

A *character constant* is one or more characters enclosed in single quotes, such as 'A', '=', '\n'. In C, single character constants have data type **int**; they are represented internally with 16 bits, with the upper byte zero or sign-extended. In C++, a character constant has type **char**. Multicharacter constants in both C and C++ have data type **int**.

## Escape sequences

The backslash character (\) is used to introduce an *escape sequence*, allowing the visual representation of certain nongraphic characters. For example, the constant \n is used for the single newline character.

A backslash is used with octal or hexadecimal numbers to represent the ASCII symbol or control code corresponding to that val-

ue; for example, '\03' for *Ctrl-C* or '\x3F' for the question mark. You can use any string of up to three octal or any number of hexadecimal numbers in an escape sequence, provided that the value is within legal range for data type **char** (0 to 0xff for Turbo C++). Larger numbers generate the compiler error, "Numeric constant too large." For example, the octal number \777 is larger than the maximum value allowed, \377, and will generate an error. The first nonoctal or nonhexadecimal character encountered in an octal or hexadecimal escape sequence marks the end of the sequence.

Originally, Turbo C allowed only three digits in a hexadecimal escape sequence. The new ANSI C rules adopted in Turbo C version 2.0 and in Turbo C++ might cause problems with old code that assumes only the first three characters are converted. For example, using Turbo C 1.x to define a string with a bell (ASCII 7) followed by numeric characters, a programmer might write:

```
printf("\x0072.1A Simple Operating System");
```

This is intended to be interpreted as \x007 and "2.1A Simple Operating System". However, Turbo C++ (and Turbo C version 2.0) compile it as the hexadecimal number \x0072 and the literal string ".1A Simple Operating System".

To avoid such problems, rewrite your code like this:

```
printf("\x007" "2.1A Simple Operating System");
```

Ambiguities may also arise if an octal escape sequence is followed by a nonoctal digit. For example, because 8 and 9 are not legal octal digits, the constant \258 would be interpreted as a two-character constant made up of the characters \25 and 8.

The next table shows the available escape sequences.

Table 1.7  
Turbo C++ escape  
sequences

The \\ must be used to  
represent a real ASCII  
backslash, as used in DOS  
paths.

Sequence	Value	Char	What it does
\\a	0x07	BEL	Audible bell
\\b	0x08	BS	Backspace
\\f	0x0C	FF	Formfeed
\\n	0x0A	LF	Newline (linefeed)
\\r	0x0D	CR	Carriage return
\\t	0x09	HT	Tab (horizontal)
\\v	0x0B	VT	Vertical tab
\\\\	0x5c	\\	Backslash
\\'	0x27	'	Single quote (apostrophe)
\\"	0x22	"	Double quote
\\?	0x3F	?	Question mark
\\O		any	O = a string of up to three octal digits
\\xH		any	H = a string of hex digits
\\XH		any	H = a string of hex digits

### Turbo C++ special two-character constants

Turbo C++ also supports two-character constants (for example, 'An', '\\n\\t', and '\\007\\007'). These constants are represented as 16-bit **int** values, with the first character in the low-order byte and the second character in the high-order byte. These constants are not portable to other C compilers.

### Signed and unsigned char

In C, one-character constants, such as 'A', '\\t', and '\\007', are also represented as 16-bit **int** values. In this case, the low-order byte is *sign extended* into the high byte; that is, if the value is greater than 127 (base 10), the upper byte is set to -1 (=0xFF). This can be disabled by declaring that the default **char** type is **unsigned** (use the -K TCC option or choose **Unsigned Characters** in the **Options | Compiler | Code Generation** menu), which forces the high byte to be zero regardless of the value of the low byte.

### Wide character constants (C only)

A character constant preceded by an *L* is a wide-character constant of data type **wchar\_t** (an integral type defined in `stddef.h`). For example,

```
x = L 'AB';
```

## Floating-point constants

A floating constant consists of six parts:

- decimal integer
- decimal point
- decimal fraction
- e or E and a signed integer exponent (optional)
- type suffix: *f* or *F* or *l* or *L* (optional)

You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter *e* (or *E*) and the signed integer exponent (but not both). These rules allow for conventional and scientific (exponent) notations.

Negative floating constants are taken as positive constants with the unary operator minus (-) prefixed.

Examples:

Constant	Value
23.45e6	$23.45 \times 10^6$
.0	0
0.	0
1.	$1.0 \times 10^0 = 1.0$
-1.23	-1.23
2e-5	$2.0 \times 10^{-5}$
3E+10	$3.0 \times 10^{10}$
.09E34	$0.09 \times 10^{34}$

### Floating point constants—data types

In the absence of any suffixes, floating-point constants are of type **double**. However, you can coerce a floating constant to be of type **float** by adding an *f* or *F* suffix to the constant. Similarly, the suffix *l* or *L* forces the constant to be data type **long double**. The next table shows the ranges available for **float**, **double**, and **long double**.

Type	Size (bits)	Range
<b>float</b>	32	$3.4 \times 10^{-38}$ to $3.4 \times 10^{38}$
<b>double</b>	64	$1.7 \times 10^{-308}$ to $1.7 \times 10^{308}$
<b>long double</b>	80	$3.4 \times 10^{-4932}$ to $1.1 \times 10^{4932}$

Table 1.8  
Turbo C++ floating constant  
sizes and ranges

## Enumeration constants

Enumeration constants are identifiers defined in **enum** type declarations. The identifiers are usually chosen as mnemonics to assist legibility. Enumeration constants are integer data types. They can be used in any expression where integer constants are valid. The identifiers used must be unique within the scope of the **enum** declaration.

See page 71 for a detailed look at **enum** declarations.

The values acquired by enumeration constants depend on the format of the enumeration declaration and the presence of optional *initializers*. In this example,

```
enum team { giants, cubs, dodgers };
```

**giants**, **cubs**, and **dodgers** are enumeration constants of type **team** that can be assigned to any variables of type **team** or to any other variable of integer type. The values acquired by the enumeration constants are

```
giants = 0, cubs = 1, dodgers = 2
```

in the absence of explicit initializers. In the following example,

```
enum team { giants, cubs=3, dodgers = giants + 1 };
```

the constants are set as follows:

```
giants = 0, cubs = 3, dodgers = 1
```

The constant values need not be unique:

```
enum team { giants, cubs = 1, dodgers = cubs - 1 };
```

Negative initializers are allowed.

## String literals

String literals, also known as string constants, form a special category of constants used to handle fixed sequences of characters. A string literal is of data type **array of char** and storage class **static**, written as a sequence of any number of characters surrounded by double quotes:

```
"This is literally a string!"
```

The null (empty) string is written "".

The characters inside the double quotes can include escape sequences (see page 13). This code, for example,

```
"\t\t\"Name\"\\\"Address\n\n"
```

prints out like this:

```
"Name" \ Address
```

"Name" is preceded by two tabs; Address is preceded by one tab. The line is followed by two new lines. The \ " provides interior double quotes.

A literal string is stored internally as the given sequence of characters plus a final null character ('\0'). A null string is stored as a single '\0' character.

Adjacent string literals separated only by whitespace are concatenated during the parsing phase. In the following example,

```
#include <stdio.h>

main()
{
    char    *p;

    p = "This is an example of how Turbo C++"
        " will automatically\ndo the concatenation for"
        " you on very long strings,\nresulting in nicer"
        " looking programs.";
    printf(p);
}
```

The output of the program is

```
This is an example of how Turbo C++ will automatically
do the concatenation for you on very long strings,
resulting in nicer looking programs.
```

You can also use the backslash (\) as a continuation character in order to extend a string constant across line boundaries:

```
puts("This is really \
a one-line string");
```

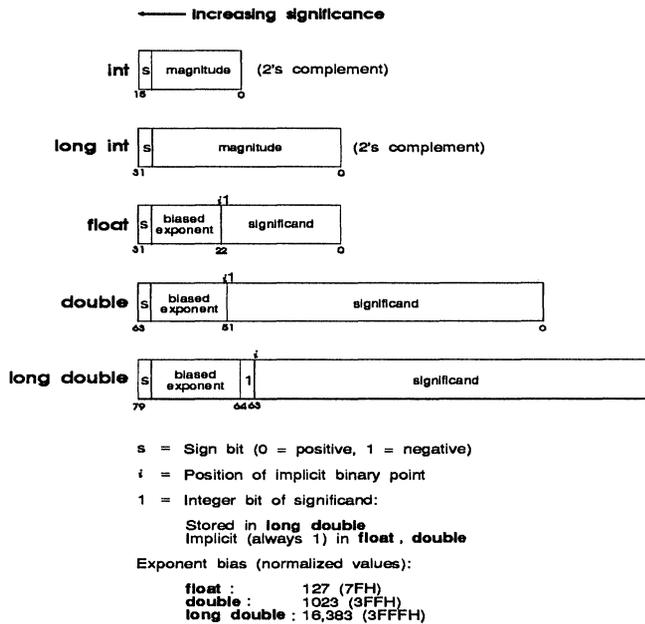
## Constants and internal representation

ANSI C acknowledges that the size and numeric range of the basic data types (and their various permutations) are implementation specific and usually derive from the architecture of the host computer. For Turbo C++, the target platform is the IBM PC family (and compatibles), so the architecture of the Intel 8088 and 80x86 microprocessors governs the choices of inner representations for the various data types. The next table lists the sizes and resulting ranges of the data types for Turbo C++. See page 39 for more information on these data types. Figure 1.1 shows how these types are represented internally.

Table 1.9: Data types, sizes, and ranges

Type	Size (bits)	Range	Sample applications
unsigned char	8	0 to 255	Small numbers and full PC character set
char	8	-128 to 127	Very small numbers and ASCII characters
enum	16	-32,768 to 32,767	Ordered sets of values
unsigned int	16	0 to 65,535	Larger numbers and loops
short int	16	-32,768 to 32,767	Counting, small numbers, loop control
int	16	-32,768 to 32,767	Counting, small numbers, loop control
unsigned long	32	0 to 4,294,967,295	Astronomical distances
long	32	-2,147,483,648 to 2,147,483,647	Large numbers, populations
float	32	$3.4 \times 10^{-38}$ to $3.4 \times 10^{38}$	Scientific (7-digit precision)
double	64	$1.7 \times 10^{-308}$ to $1.7 \times 10^{308}$	Scientific (15-digit precision)
long double	80	$3.4 \times 10^{-4932}$ to $1.1 \times 10^{4932}$	Financial (19-digit precision)
near pointer	16	Not applicable	Manipulating memory addresses
far pointer	32	Not applicable	Manipulating addresses outside current segment

Figure 1.1  
Internal representations of data types



Constant expressions A constant expression is an expression that always evaluates to a constant (and it must evaluate to a constant that is in the range of representable values for its type). Constant expressions are evaluated just as regular expressions are. You can use a constant expression anywhere that a constant is legal. The syntax for constant expressions is

*constant-expression:*  
*Conditional-expression*

Constant expressions cannot contain any of the following operators, unless the operators are contained within the operand of a **sizeof** operator:

- assignment
- decrement
- function call
- comma

## Operator descriptions

---

*Operators* are tokens that trigger some computation when applied to variables and other objects in an expression. Turbo C++ is especially rich in operators, offering not only the common arithmetical and logical operators, but also many for bit-level manipulations, structure and union component access, and pointer operations (referencing and dereferencing).



C++ extensions offer additional operators for accessing class members and their objects, together with a mechanism for overloading operators. *Overloading* lets you redefine the action of any standard operators when applied to the objects of a given class. In this section, we confine our discussion to the standard operators of Turbo C++. Overloading is covered starting on page 124.

After defining the standard operators, we discuss data types and declarations, and explain how these affect the actions of each operator. From there we can proceed with the syntax for building expressions from operators, punctuators, and objects.

The operators in Turbo C++ are defined as follows:

*operator:* one of

<b>[ ]</b>	<b>( )</b>	<b>.</b>	<b>-&gt;</b>	<b>++</b>	<b>--</b>
<b>&amp;</b>	<b>*</b>	<b>+</b>	<b>-</b>	<b>~</b>	<b>!</b>
<b>sizeof</b>	<b>/</b>	<b>%</b>	<b>&lt;&lt;</b>	<b>&gt;&gt;</b>	<b>&lt;</b>
<b>&gt;</b>	<b>&lt;=</b>	<b>&gt;=</b>	<b>==</b>	<b>!=</b>	<b>^</b>
<b> </b>	<b>&amp;&amp;</b>	<b>  </b>	<b>?:</b>	<b>=</b>	<b>*=</b>
<b>/=</b>	<b>%=</b>	<b>+=</b>	<b>-=</b>	<b>&lt;&lt;=</b>	<b>&gt;&gt;=</b>
<b>&amp;=</b>	<b>^=</b>	<b> =</b>	<b>,</b>	<b>#</b>	<b>##</b>

The operators # and ## are used only by the preprocessor (see page 133).

And the following operators specific to C++:

<b>::</b>	<b>.*</b>	<b>-&gt;*</b>
-----------	-----------	---------------

Except for [ ], ( ), and ?:, which bracket expressions, the multicharacter operators are considered as single tokens. The same operator token can have more than one interpretation; depending on the context. For example,

A * B	Multiplication
*ptr	Dereference (indirection)
A & B	Bitwise AND
&A	Address operation
int &	Reference modifier (C++)
label:	Statement label
a ? x : y	Conditional statement
void func(int n);	Function declaration
a = (b+c)*d;	Parenthesized expression
a, b, c;	Comma expression
func(a, b, c);	Function call
a = ~b;	Bitwise negation (one's complement)
~func() {delete a;}	Destructor (C++)

## Unary operators

<b>&amp;</b>	Address operator
<b>*</b>	Indirection operator
<b>+</b>	Unary plus
<b>-</b>	Unary minus
<b>~</b>	Bitwise complement (1's complement)
<b>!</b>	Logical negation
<b>++</b>	Prefix: preincrement; Postfix: postincrement
<b>--</b>	Prefix: predecrement; Postfix: postdecrement

## Binary operators

---

Additive operators	+	Binary plus (addition)
	-	Binary minus (subtraction)
Multiplicative operators	*	Multiply
	/	Divide
	%	Remainder
Shift operators	<<	Shift left
	>>	Shift right
Bitwise operators	&	Bitwise AND
	^	Bitwise XOR (exclusive OR)
		Bitwise inclusive OR
Logical operators	&&	Logical AND
		Logical OR
Assignment operators	=	Assignment
	*=	Assign product
	/=	Assign quotient
	%=	Assign remainder (modulus)
	+=	Assign sum
	-=	Assign difference
	<<=	Assign left shift
	>>=	Assign right shift
	&=	Assign bitwise AND
	^=	Assign bitwise XOR
=	Assign bitwise OR	
Relational operators	<	Less than
	>	Greater than
	<=	Less than or equal to
	>=	Greater than or equal to

Equality operators	<code>==</code> <code>!=</code>	Equal to Not equal to
Component selection operators	<code>.</code> <code>-&gt;</code>	Direct component selector Indirect component selector
Class-member operators	<code>::</code> <code>.*</code> <code>-&gt;*</code>	Scope access/resolution Dereference pointer to class member Dereference pointer to class member
Conditional operator	<code>a ? x : y</code>	“if <i>a</i> then <i>x</i> ; else <i>y</i> ”
Comma operator	<code>,</code>	Evaluate; e.g., <i>a</i> , <i>b</i> , <i>c</i> ; from left to right

The operator functions, as well as their syntax, precedences, and associativities, are covered starting on page 73.

---

## Punctuators

The punctuators (also known as separators) in Turbo C++ are defined as follows:

*punctuator*: one of

`[ ] ( ) { } , ; : ... * = #`

**Brackets** `[]` (open and close brackets) indicate single and multidimensional array subscripts:

```
char ch, str[] = "Stan";
int mat[3][4];          /* 3 x 4 matrix */
ch = str[3];           /* 4th element */
...
```

**Parentheses** `()` (open and close parentheses) group expressions, isolate conditional expressions, and indicate function calls and function parameters:

```
d = c * (a + b);      /* override normal precedence */
if (d == z) ++x;     /* essential with conditional statement */
func();              /* function call, no args */
int (*fptr)();       /* function pointer declaration */
fptr = func;         /* no () means func pointer */
```

```
void func2(int n); /* function declaration with args */
```

Parentheses are recommended in macro definitions to avoid potential precedence problems during expansion:

```
#define CUBE(x) ((x) * (x) * (x))
```

The use of parentheses to alter the normal operator precedence and associativity rules is covered on page 76.

**Braces** { } (open and close braces) indicate the start and end of a compound statement:

```
if (d == z)
{
    ++x;
    func();
}
```

The closing brace serves as a terminator for the compound statement, so a ; (semicolon) is not required after the }, except in structure or class declarations. Often, the semicolon is illegal, as in

```
if (statement)
    {}; /*illegal semicolon*/
else
```

**Comma** The comma (,) separates the elements of a function argument list:

```
void func(int n, float f, char ch);
```

The comma is also used as an operator in *comma expressions*. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them:

```
func(i, j); /* call func with two args */
func((exp1, exp2), (exp3, exp4, exp5)); /* also calls func
with two args! */
```

**Semicolon** The semicolon (;) is a statement terminator. Any legal C expression (including the empty expression) followed by ; is interpreted as a statement, known as an *expression statement*. The expression is evaluated and its value is discarded. If the expression statement has no side effects, Turbo C++ may ignore it.

```
a + b; /* maybe evaluate a + b, but discard value */
++a; /* side effect on a, but discard value of ++a */
; /* empty expression = null statement */
```

Semicolons are often used to create an *empty statement*:

```
for (i = 0; i < n; i++)
{
    ;
}
```

**Colon** Use the colon (:) to indicate a labeled statement:

```
start:   x=0;
...
goto start;
...
switch (a) {
    case 1: puts("One");
            break;
    case 2: puts("Two");
            break;
    ...
    default: puts("None of the above!");
            break;
}
```

Labels are covered on page 92.

**Ellipsis** Ellipsis (...) are three successive periods with no whitespace intervening. Ellipsis are used in the formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types:

```
void func(int n, char ch,...);
```

This declaration indicates that **func** will be defined in such a way that calls must have at least two arguments, an **int** and a **char**, but can also have any number of additional arguments.



In C++, you can omit the comma preceding the ellipsis.

**Asterisk (pointer declaration)**

The \* (asterisk) in a variable declaration denotes the creation of a pointer to a type:

```
char *char_ptr; /* a pointer to char is declared */
```

Pointers with multiple levels of indirection can be declared by indicating a pertinent number of asterisks:

```
int **int_ptr;      /* a pointer to a pointer to an int */
double ***double_ptr; /* a pointer to a pointer to a pointer
                    to doubles */
```

You can also use the asterisk as an operator to either dereference a pointer or as the multiplication operator:

```
i = *int_ptr;
a = b * 3.14;
```

Equal sign (initializer) The = (equal sign) separates variable declarations from initialization lists:

```
char array[5] = { 1, 2, 3, 4, 5 };
int x = 5;
```

In a C function, no code can precede any variable declarations. In C++, declarations of any type can appear (with some restrictions) at any point within the code.

In a C++ function argument list, the equal sign indicates the default value for a parameter:

```
int f(int i = 0) { ... } /* parameter i has default value of
                        zero */
```

The equal sign is also used as the assignment operator in expressions:

```
a = b + c;
ptr = farmalloc(sizeof(float)*100);
```

Pound sign (preprocessor directive) The # (pound sign) indicates a preprocessor directive when it occurs as the first non whitespace character on a line. It signifies a compiler action, not necessarily associated with code generation. See page 133 for more on the preprocessor directives.

# and ## (double pound signs) are also used as operators to perform token replacement and merging during the preprocessor scanning phase.

## Declarations

---

This section briefly reviews concepts related to declarations: objects, types, storage classes, scope, visibility, duration, and linkage. A general knowledge of these is essential before tackling the full declaration syntax.

Scope, visibility, duration, and linkage determine those portions of a program that can make legal references to an identifier in order to

access its object. Scope is discussed starting on page 29; visibility is discussed starting on page 30; duration is discussed starting on page 31; and linkage is discussed on page 32.

## Objects

---

An *object* is an identifiable region of memory that can hold a fixed or variable value (or set of values). (This use of the word *object* is not to be confused with the more general term used in object-oriented languages—see Chapter 5, “A C++ primer,” in *Getting Started*.) Each value has an associated name and type (also known as a *data type*). The name is used to access the object. This name can be a simple identifier, or it can be a complex expression that uniquely “points” to the object. The type is used

- to determine the correct memory allocation required initially,
- to interpret the bit patterns found in the object during subsequent accesses,
- and in many type-checking situations to ensure that illegal assignments are trapped.

Turbo C++ supports many standard (predefined) and user-defined data types, including signed and unsigned integers in various sizes, floating point numbers in various precisions, structures, unions, arrays, and classes. In addition, pointers to most of these objects can be established and manipulated in various memory models.

The Turbo C++ standard libraries and your own program and header files must provide unambiguous identifiers (or expressions derived from them) and types so that Turbo C++ can consistently access, interpret, and (possibly) change the bit patterns in memory corresponding to each active object in your program.

Declarations establish the necessary mapping between identifiers and objects. Each declaration associates an identifier with a data type. Most declarations, known as *defining declarations*, also establish the creation (where and when) of the object, that is, the allocation of physical memory and its possible initialization. Other declarations, known as *referencing declarations*, simply make their identifiers and types known to the compiler. There can be many referencing declarations for the same identifier, especially in a multifile program, but only one defining declaration for that identifier is allowed.

Generally speaking, an identifier cannot be legally used in a program before its *declaration point* in the source code. Legal exceptions to this

rule, known as *forward references*, are labels, structures, union tags, and calls to undeclared functions.

---

## Lvalues

An *lvalue* is an object locator an expression that designates an object. An example of an lvalue expression is  $*P$ , where  $P$  is any expression evaluating to a nonnull pointer. A *modifiable lvalue* is an identifier or expression that relates to an object that can be accessed and legally changed in memory. A **const** pointer to a constant, for example, is *not* a modifiable lvalue. A pointer to a constant can be changed (but its dereferenced value cannot).

Historically, the  $l$  stood for "left," meaning that an lvalue could legally stand on the left (the receiving end) of an assignment statement. Now only modifiable lvalues can legally stand to the left of an assignment statement. For example, if  $a$  and  $b$  are nonconstant integer identifiers with properly allocated memory storage, they are both modifiable lvalues, and assignments such as  $a = 1$ ; and  $b = a + b$  are legal.

**Rvalues** The expression  $a + b$  is not an lvalue:  $a + b = a$  is illegal because the expression on the left is not related to an object. Such expressions are often called *rvalues* (short for right values).

---

## Types and storage classes

Associating identifiers with objects requires that each identifier has at least two attributes: *storage class* and *type* (sometimes referred to as data type). The Turbo C++ compiler deduces these attributes from implicit or explicit declarations in the source code.

Storage class dictates the location (data segment, register, heap, or stack) of the object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the syntax of the declaration, by its placement in the source code, or by both of these factors.

The type, as explained earlier, determines how much memory is allocated to an object and how the program will interpret the bit patterns found in the object's storage allocation. A given data type can be viewed as the set of values (often implementation-dependent) that identifiers of that type can assume, together with the set of operations allowed on those values. The special compile-time opera-

tor, **sizeof**, lets you determine the size in bytes of any standard or user-defined type; see page 81 for more on this operator.

## Scope

---

The *scope* of an identifier is that part of the program in which the identifier can be used to access its object. There are five categories of scope: *block* (or *local*), *function*, *function prototype*, *file*, and *class* (C++ only). These depend on how and where identifiers are declared.

- Block scope** The *scope* of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as the *enclosing* block). Parameter declarations with a function definition also have block scope, limited to the scope of the block which defines the function.
- Function scope** The only identifiers having function scope are statement labels. Label names can be used with **goto** statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing *label\_name*: followed by a statement. Label names must be unique within a function.
- Function prototype scope** Identifiers declared within the list of parameter declarations in a function prototype (not part of a function definition) have function prototype scope. This scope ends at the end of the function prototype.
- File scope** File scope identifiers, also known as *globals*, are declared outside of all blocks and classes; their scope is from the point of declaration to the end of the source file.
- Class scope (C++)** For now, think of a class as a named collection of members, including data structures and functions that act on them. Class scope applies, with some exceptions, to the names of the members of a particular class. Classes and their objects have many special access and scoping rules; see pages 102 to 113.
- Scope and name spaces** *Name space* is the scope within which an identifier must be unique. There are four distinct classes of identifiers in C:

Structures, classes, and enumerations are in the same name space in C++.

1. **goto** label names. These must be unique within the function in which they are declared.
2. Structure, union, and enumeration tags. These must be unique within the block in which they are defined. Tags declared outside of any function must be unique within all tags defined externally.
3. Structure and union member names. These must be unique within the structure or union in which they are defined. There is no restriction on the type or offset of members with the same member name in different structures.
4. Variables, **typedefs**, and enumeration members. These must be unique within the scope in which they are defined. Externally declared identifiers must be unique among externally declared variables.

## Visibility

---

The *visibility* of an identifier is that region of the program source code from which legal access can be made to the identifier's associated object.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily *hidden* by the appearance of a duplicate identifier: The object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended.

Visibility cannot exceed scope, but scope can exceed visibility.

```
...
{
    int i; char ch; // auto by default
    i = 3;         // int i and char ch in scope and visible
...
{
    double i;
    i = 3.0e3;    // double i in scope and visible
                // int i=3 in scope but hidden
    ch = 'A';    // char ch in scope and visible
}
                // double i out of scope
i += 1;        // int i visible and = 4
...           // char ch still in scope & visible = 'A'
}
...           // int i and char ch out of scope
```



Again, special rules apply to hidden class names and class member names: Special C++ operators allow hidden identifiers to be accessed under certain conditions (see page 103).

## Duration

---

*Duration*, closely related to storage class, defines the period during which the declared identifiers have real, physical objects allocated in memory. We also distinguish between compile-time and run-time objects. Variables, for instance, unlike **typedefs** and types, have real memory allocated during run time. There are three kinds of duration: *static*, *local*, and *dynamic*.

### Static duration

Objects with *static* duration are allocated memory as soon as execution is underway; this storage allocation lasts until the program terminates. Static duration objects usually reside in fixed data segments allocated according to the memory model in force. All functions, wherever defined, are objects with static duration. All variables with file scope have static duration. Other variables can be given static duration by using the explicit **static** or **extern** storage class specifiers.

Static duration objects are initialized to zero (or null) in the absence of any explicit initializer or, in C++, constructor.

Static duration must not be confused with file or global scope. An object can have static duration and local scope.

### Local duration

*An object with local duration also has local scope, since it does not exist outside of its enclosing block. The converse is not true: A local scope object can have static duration.*

*Local* duration objects, also known as *automatic* objects, lead a more precarious existence. They are created on the stack (or in a register) when the enclosing block or function is entered. They are deallocated when the program exits that block or function. Local duration objects must be explicitly initialized; otherwise, their contents are unpredictable. Local duration objects always must have local or function scope. The storage class specifier **auto** may be used when declaring local duration variables, but is usually redundant, since **auto** is the default for variables declared within a block.

When declaring variables (for example, **int**, **char**, **float**), the storage class specifier **register** also implies **auto**; but a request (or hint) is passed to the compiler that the object be allocated a register if possible. Turbo C++ can be set to allocate a register to a local integral or pointer variable, if one is free. If no register is free, the variable is allocated as an **auto**, local object with no warning or error.

Dynamic duration *Dynamic* duration objects are created and destroyed by specific function calls during a program. They are allocated storage from a special memory reserve known as the *heap*, using either standard library functions such as **malloc**, or by using the C++ operator **new**. The corresponding deallocations are made using **free** or **delete**.

---

## Translation units

The term *translation unit* refers to a source code file together with any included files, but less any source lines omitted by conditional preprocessor directives. Syntactically, a translation unit is defined as a sequence of external declarations:

```
translation-unit:
    external-declaration
    translation-unit external-declaration

external-declaration
    function-definition
    declaration
```

The word *external* has several connotations in C; here it refers to declarations made outside of any function, and which therefore have file scope. (External linkage is a distinct property; see the following section, "Linkage.") Any declaration that also reserves storage for an object or function is called a definition (or defining declaration). For more details, see "External declarations and definitions" on page 36.

---

## Linkage

An executable program is usually created by compiling several independent translation units, then linking the resulting object files with preexisting libraries. A problem arises when the same identifier is declared in different scopes (for example, in different files), or declared more than once in the same scope. Linkage is the process that allows each instance of an identifier to be associated correctly with one particular object or function. All identifiers have one of three linkage attributes, closely related to their scope: external linkage, internal linkage, or no linkage. These attributes are determined by the placement and format of your declarations, together with the explicit (or implicit by default) use of the storage class specifier **static** or **extern**.

Each instance of a particular identifier with *external linkage* represents the same object or function throughout the entire set of files and libraries making up the program. Each instance of a particular identifier with *internal linkage* represents the same object or function only within one file. Identifiers with *no linkage* represent unique entities.

External and internal linkage rules are as follows:

1. Any object or file identifier having file scope will have internal linkage if its declaration contains the storage class specifier **static**. For C, if the same identifier appears with both internal and external linkage within the same file, the identifier will have internal linkage. In C++, it will have external linkage.
2. If the declaration of an object or function identifier contains the storage class specifier **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no such visible declaration, the identifier has external linkage.
3. If a function is declared without a storage class specifier, its linkage is determined as if the storage class specifier **extern** had been used.
4. If an object identifier with file scope is declared without a storage class specifier, the identifier has external linkage.

The following identifiers have no linkage attribute:

1. any identifier declared to be other than an object or a function (for example, a **typedef** identifier)
2. function parameters
3. block scope identifiers for objects declared without the storage class specifier **extern**

## Declaration syntax

---

All six interrelated attributes (storage class, type, scope, visibility, duration, and linkage) are determined in diverse ways by *declarations*.

Declarations can be *defining declarations* (also known simply as *definitions*) or *referencing declarations* (sometimes known as *nondefining declarations*). A defining declaration, as the name implies, performs both the duties of declaring and defining; the nondefining

declarations require a definition to be added somewhere in the program. A referencing declaration simply introduces one or more identifier names into a program. A definition actually allocates memory to an object and associates an identifier with that object.

---

## Tentative definitions

The ANSI C standard introduces a new concept: that of the *tentative definition*. Any external data declaration that has no storage class specifier and no initializer is considered a tentative definition. If the identifier declared appears in a later definition, then the tentative definition is treated as if the **extern** storage class specifier were present. In other words, the tentative definition becomes a simple referencing declaration.

If the end of the translation unit is reached and no definition has appeared with an initializer for the identifier, then the tentative definition becomes a full definition, and the object defined has uninitialized (zero-filled) space reserved for it. For example,

```
int x;  
int x;          /*legal, one copy of x is reserved */  
  
int y;  
int y = 4;     /* legal, y is initialized to 4 */  
  
int z = 5;  
int z = 6;     /* not legal, both are initialized definitions */
```

---

## Possible declarations

The range of objects that can be declared includes

- variables
- functions
- classes and class members (C++)
- types
- structure, union, and enumeration tags
- structure members
- union members
- arrays of other types
- enumeration constants
- statement labels
- preprocessor macros

The full syntax for declarations is shown in the following tables. The recursive nature of the declarator syntax allows complex declarators. We encourage the use of **typedefs** to improve legibility.

Table 1.10  
Turbo C++ declaration syntax

---

<i>declaration:</i>	<i>&lt;decl-specifiers&gt; &lt;declarator-list&gt;;</i> <i>asm-declaration</i> <i>function-declaration</i> <i>linkage-specification</i>	<b>short</b> <b>int</b> <b>long</b> <b>signed</b> <b>unsigned</b> <b>float</b> <b>double</b> <b>void</b>
<i>decl-specifier:</i>	<i>storage-class-specifier</i> <i>type-specifier</i> <i>fcn-specifier</i> <b>friend</b> (C++ specific) <b>typedef</b>	<i>elaborated-type-specifier:</i> <i>class-key identifier</i> <i>class-key class-name</i> <b>enum</b> <i>enum-name</i>
<i>decl-specifiers:</i>	<i>&lt;decl-specifiers&gt; decl-specifier</i>	<i>class-key:</i> (C++ specific) <b>class</b> <b>struct</b> <b>union</b>
<i>storage-class-specifier:</i>	<b>auto</b> <b>register</b> <b>static</b> <b>extern</b>	<i>enum-specifier:</i> <b>enum</b> <i>&lt;identifier&gt; { &lt;enum-list&gt; }</i>
<i>fcn-specifier:</i> (C++ specific)	<b>inline</b> <b>virtual</b>	<i>enum-list:</i> <i>enumerator</i> <i>enumerator-list , enumerator</i>
<i>type-specifier:</i>	<i>simple-type-name</i> <i>class-specifier</i> <i>enum-specifier</i> <i>elaborated-type-specifier</i> <b>const</b> <b>volatile</b>	<i>enumerator:</i> <i>identifier</i> <i>identifier = constant-expression</i>
<i>simple-type-name:</i>	<i>class-name</i> <b>typedef-name</b> <b>char</b>	<i>constant-expression:</i> <i>conditional-expression</i> <i>linkage-specification:</i> (C++ specific) <b>extern</b> <i>string</i> { <i>&lt;declaration-list&gt;</i> } <b>extern</b> <i>string</i> <i>declaration</i>
		<i>declaration-list:</i> <i>declaration</i> <i>declaration-list ; declaration</i>

---

For the following table, note that there are restrictions on the number and order of modifiers and qualifiers. Also, the modifiers listed are the only addition to the declarator syntax that are not ANSI C or C++. These modifiers are each discussed in greater detail starting on page 46.

Table 1.11: Turbo C++ declarator syntax

<i>declarator-list</i> :	<i>class-name</i> (C++ specific) ~ <i>class-name</i> (C++ specific) <b>typedef-name</b>
<i>init-declarator</i> :	<i>type-name</i> :
<i>declarator</i> <initializer>	<i>type-specifier</i> < <i>abstract-declarator</i> >
<i>declarator</i> :	<i>abstract-declarator</i> :
<i>dname</i>	<i>ptr-operator</i> < <i>abstract-declarator</i> >
<i>modifier-list</i>	< <i>abstract-declarator</i> > ( <i>argument-declaration-list</i> ) < <i>cv-qualifier-list</i> >
<i>ptr-operator declarator</i>	< <i>abstract-declarator</i> > [ < <i>constant-expression</i> > ]
<i>declarator</i> ( <i>parameter-declaration-list</i> ) < <i>cv-qualifier-list</i> >	( <i>abstract-declarator</i> )
(The < <i>cv-qualifier-list</i> > is for C++ only.)	<i>argument-declaration-list</i> :
<i>declarator</i> [ < <i>constant-expression</i> > ]	< <i>arg-declaration-list</i> >
( <i>declarator</i> )	<i>arg-declaration-list</i> , ...
<i>modifier-list</i> :	< <i>arg-declaration-list</i> > ... (C++ specific)
<i>modifier</i>	<i>arg-declaration-list</i> :
<i>modifier-list modifier</i>	<i>argument-declaration</i>
<i>modifier</i> :	<i>arg-declaration-list</i> , <i>argument-declaration</i>
<b>cdecl</b>	<i>argument-declaration</i> :
<b>pascal</b>	<i>decl-specifiers declarator</i>
<b>interrupt</b>	<i>decl-specifiers declarator</i> = <i>expression</i> (C++ specific)
<b>near</b>	<i>decl-specifiers</i> < <i>abstract-declarator</i> >
<b>far</b>	<i>decl-specifiers</i> < <i>abstract-declarator</i> > = <i>expression</i> (C++ specific)
<b>huge</b>	<i>fcn-definition</i> :
<i>ptr-operator</i> :	< <i>decl-specifiers</i> > <i>declarator</i> < <i>ctor-initializer</i> > <i>fcn-body</i>
* < <i>cv-qualifier-list</i> >	<i>fcn-body</i> :
& < <i>cv-qualifier-list</i> > (C++ specific)	<i>compound-statement</i>
<i>class-name</i> :: * < <i>cv-qualifier-list</i> > (C++ specific)	<i>initializer</i> :
<i>cv-qualifier-list</i> :	= <i>expression</i>
<i>cv-qualifier</i> < <i>cv-qualifier-list</i> >	= { <i>initializer-list</i> }
<i>cv-qualifier</i>	( <i>expression-list</i> ) (C++ specific)
<b>const</b>	<i>initializer-list</i> :
<b>volatile</b>	<i>expression</i>
<i>dname</i> :	<i>initializer-list</i> , <i>expression</i>
<i>name</i>	{ <i>initializer-list</i> <> }

## External declarations and definitions

The storage class specifiers **auto** and **register** cannot appear in an external declaration (see "Translation units," page 32). For each identifier in a translation unit declared with internal linkage, there can be no more than one external definition.

An external definition is an external declaration that also defines an object or function; that is, it also allocates storage. If an identifier declared with external linkage is used in an expression (other than as part of the operand of **sizeof**), there must be exactly one external definition of that identifier somewhere in the entire program.

Turbo C++ allows later re-declarations of external names, such as arrays, structures, and unions, to add information to earlier declarations. For example,

```

int a[];           // no size
struct mystruct;  // tag only, no member declarators
...
int a[3] = {1, 2, 3}; // supply size and initialize
struct mystruct {
    int i, j;
};                // add member declarators

```

The following table covers class declaration syntax. Page 98 covers C++ reference types (closely related to pointer types) in detail.

Table 1.12: Turbo C++ class declarations (C++ only)

<i>class-specifier:</i> class-head { <member-list> }	<i>access-specifier</i> <virtual> class-name																																										
<i>class-head:</i> class-key <identifier> <base-spec> class-key class-name <base-spec>	<i>access-specifier:</i> <b>private</b> <b>protected</b> <b>public</b>																																										
<i>member-list:</i> member-declaration <member-list> access-specifier : <member-list>	<i>conversion-function-name:</i> <b>operator</b> conversion-type-name																																										
<i>member-declaration:</i> <decl-specifiers> <member-declarator-list> ; function-definition <;> qualified-name ;	<i>conversion-type-name:</i> type-specifiers <ptr-operator>																																										
<i>member-declarator-list:</i> member-declarator member-declarator-list, member-declarator	<i>ctor-initializer:</i> : mem-initializer-list																																										
<i>member-declarator:</i> declarator <pure-specifier> <identifier> : constant-expression	<i>mem-initializer-list:</i> mem-initializer mem-initializer, mem-initializer-list																																										
<i>pure-specifier:</i> <b>= 0</b>	<i>mem-initializer:</i> class name ( <argument-list> ) identifier ( <argument-list> )																																										
<i>base-spec:</i> : base-list	<i>operator-function-name:</i> <b>operator</b> operator																																										
<i>base-list:</i> base-specifier base-list, base-specifier	<i>operator:</i> one of <b>new delete sizeof</b>																																										
<i>base-specifier:</i> class-name <b>virtual</b> <access-specifier> class-name	<table border="0"> <tbody> <tr> <td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td></tr> <tr> <td>&amp;</td><td> </td><td>~</td><td>!</td><td>=</td><td>&lt; &gt;</td></tr> <tr> <td>+=</td><td>-=</td><td>*=</td><td>/=</td><td>%=</td><td>^=</td></tr> <tr> <td>&amp;=</td><td> =</td><td>&lt;&lt;</td><td>&gt;&gt;</td><td>&gt;&gt;=</td><td>&lt;&lt;=</td></tr> <tr> <td>==</td><td>!=</td><td>&lt;=</td><td>&gt;=</td><td>&amp;&amp;</td><td>  </td></tr> <tr> <td>++</td><td>--</td><td>,</td><td>-&gt;*</td><td>-&gt;</td><td>()</td></tr> <tr> <td>[]</td><td>.*</td><td></td><td></td><td></td><td></td></tr> </tbody> </table>	+	-	*	/	%	^	&		~	!	=	< >	+=	-=	*=	/=	%=	^=	&=	=	<<	>>	>>=	<<=	==	!=	<=	>=	&&		++	--	,	->*	->	()	[]	.*				
+	-	*	/	%	^																																						
&		~	!	=	< >																																						
+=	-=	*=	/=	%=	^=																																						
&=	=	<<	>>	>>=	<<=																																						
==	!=	<=	>=	&&																																							
++	--	,	->*	->	()																																						
[]	.*																																										

## Type specifiers

---

The *type specifier* with one or more optional *modifiers* is used to specify the type of the declared identifier:

```
int i; // declare i as a signed integer
unsigned char ch1, ch2; // declare two unsigned chars
```

By long-standing tradition, if the type specifier is omitted, type **signed int** (or equivalently, **int**) is the assumed default. However, in C++ there are some situations where a missing type specifier leads to syntactic ambiguity, so C++ practice uses the explicit entry of all **int** type specifiers.

## Type taxonomy

---

There are four basic type categories: *void*, *scalar*, *function*, and *aggregate*. The scalar and aggregate types can be further divided as follows:

- Scalar: arithmetic, enumeration, pointer, and, in C++, reference types
- Aggregate: array, structure, union, and, in C++, class types

Types can also be divided into *fundamental* and *derived* types. The fundamental types are **void**, **char**, **int**, **float**, and **double**, together with **short**, **long**, **signed**, and **unsigned** variants of some of these. The derived types include pointers and references to other types, arrays of other types, function types, class types, structures, and unions.



A class object, for example, can hold a number of objects of different types together with functions for manipulating these objects, plus a mechanism to control access and inheritance from other classes.

Given any nonvoid type **type** (with some provisos), you can declare derived types as follows:

Table 1.13  
Declaring types

<b>type</b> <i>t</i> ;	An object of type <b>type</b>
<b>type</b> <i>array</i> [10];	Ten <b>types</b> : <i>array</i> [0] – <i>array</i> [9]
<b>type</b> * <i>ptr</i> ;	<i>ptr</i> is a pointer to <b>type</b>
<b>type</b> & <i>ref</i> = <i>t</i> ;	<i>ref</i> is a reference to <b>type</b> (C++)
<b>type</b> <i>func</i> ( <b>void</b> );	<b>func</b> returns value of type <b>type</b>
<b>void</b> <i>func1</i> ( <b>type</b> <i>t</i> );	<b>func1</b> takes a type <b>type</b> parameter
<b>struct</b> <i>st</i> { <b>type</b> <i>t1</i> ; <b>type</b> <i>t2</i> };	structure <i>st</i> holds two <b>types</b>

And here's how you could declare derived types in a class:

```
class ct { // class ct holds ptr to type plus a function
        // taking a type parameter (C++)
    type *ptr;
    public:
    void func(type*);
```

### Type void

**void** is a special type specifier indicating the absence of any values. It is used in the following situations:

*C++ handles **func()** in a special manner. See "Declarations and prototypes" on page 60 and code examples on page 61.*

- An empty parameter list in a function declaration:

```
int func(void); // func takes no arguments
```

- When the declared function does not return a value:

```
void func(int n); // return value
```

- As a generic pointer: A pointer to **void** is a generic pointer to anything:

```
void *ptr; // ptr can later be set to point to any object
```

- In *typecasting* expressions:

```
extern int errfunc(); // returns an error code
```

```
...
```

```
(void) errfunc(); // discard return value
```

### The fundamental types

*signed and unsigned are modifiers that can be applied to the integral types.*

The fundamental type specifiers are built from the following keywords:

<b>char</b>	<b>int</b>	<b>signed</b>
<b>double</b>	<b>long</b>	<b>unsigned</b>
<b>float</b>	<b>short</b>	

From these keywords, you can build the integral and floating-point types, which are together known as the *arithmetic* types. The include

file `limits.h` contains definitions of the value ranges for all the fundamental types.

Integral types **char**, **short**, **int**, and **long**, together with their unsigned variants, are all considered *integral* data types. The integral type specifiers are as follows, with synonyms listed on the same line:

Table 1.14  
Integral types

char, signed char	Synonyms if default char set to signed
unsigned char	
char, unsigned char	Synonyms if default char set to unsigned
signed char	
int, signed int	
unsigned, unsigned int	
short, short int, signed short int	
unsigned short, unsigned short int	
long, long int, signed long int	
unsigned long, unsigned long int	

At most, one of **signed** and **unsigned** can be used with **char**, **short**, **int**, or **long**. If you use the keywords **signed** and **unsigned** on their own, they mean **signed int** and **unsigned int**, respectively.

In the absence of **unsigned**, **signed** is usually assumed. An exception arises with **char**. Turbo C++ lets you set the default for **char** to be **signed** or **unsigned**. (The default, if you don't set it yourself, is **signed**.) If the default is set to **unsigned**, then the declaration `char ch` declares `ch` as **unsigned**. You would need to use `signed char ch` to override the default. Similarly, with a **signed** default for **char**, you would need an explicit `unsigned char ch` to declare an **unsigned char**.

At most, one of **long** and **short** can be used with **int**. The keywords **long** and **short** used on their own mean **long int** and **short int**.

ANSI C does not dictate the sizes or internal representations of these types, except to insist that **short**, **int**, and **long** form a non-decreasing sequence with "**short** <= **int** <= **long**." All three types can legally be the same. This is important if you want to write portable code aimed at other platforms.

In Turbo C++, the types **int** and **short** are equivalent, both being 16 bits. **long** is a 32-bit object. The signed varieties are all stored in 2's complement format using the MSB (most significant bit) as a sign bit: 0 for positive, 1 for negative (which explains the ranges shown in Table 1.9 on page 19). In the unsigned versions, all bits are used to give a range of  $0 - (2^n - 1)$ , where  $n$  is 8, 16, or 32.

Floating-point types The representations and sets of values for the floating-point types are implementation dependent; that is, each implementation of C is free to define them. Turbo C++ uses the IEEE floating-point formats. (Appendix A, “ANSI implementation-specific standards,” tells more about implementation-specific items.)

**float** and **double** are 32- and 64-bit floating-point data types, respectively. **long** can be used with **double** to declare an 80-bit precision floating-point identifier: **long double test\_case**, for example.

Table 1.9 on page 19 indicates the storage allocations for the floating-point types.

Standard conversions When you use an arithmetic expression, such as  $a + b$ , where  $a$  and  $b$  are different arithmetic types, Turbo C++ performs certain internal conversions before the expression is evaluated. These standard conversions include promotions of “lower” types to “higher” types in the interests of accuracy and consistency.

Here are the steps Turbo C++ uses to convert the operands in an arithmetic expression:

1. Any small integral types are converted as shown in Table 1.15. After this, any two values associated with an operator are either **int** (including the **long** and **unsigned** modifiers, **double**, **float**, or **long double**).
2. If either operand is of type **long double**, the other operand is converted to **long double**.
3. Otherwise, if either operand is of type **double**, the other operand is converted to **double**.
4. Otherwise, if either operand is of type **float**, the other operand is converted to **float**.
5. Otherwise, if either operand is of type **unsigned long**, the other operand is converted to **unsigned long**.
6. Otherwise, if either operand is of type **long**, then the other operand is converted to **long**.
7. Otherwise, if either operand is of type **unsigned**, then the other operand is converted to **unsigned**.
8. Otherwise, both operands are of type **int**.

The result of the expression is the same type as that of the two operands.

Table 1.15  
Methods used in standard  
arithmetic conversions

Type	Converts to	Method
char	int	Zero or sign-extended (depends on default char type)
unsigned char	int	Zero-filled high byte (always)
signed char	int	Sign-extended (always)
short	int	Same value
unsigned short	unsigned int	Same value
enum	int	Same value

### Special char, int, and enum conversions

*The conversions discussed in this section are specific to Turbo C++.*

Assigning a signed character object (such as a variable) to an integral object results in automatic sign extension. Objects of type **signed char** always use sign extension; objects of type **unsigned char** always set the high byte to zero when converted to **int**.

Converting a longer integral type to a shorter type truncates the higher order bits and leaves low-order bits unchanged. Converting a shorter integral type to a longer type either sign extends or zero fills the extra bits of the new value, depending on whether the shorter type is **signed** or **unsigned**, respectively.

## Initialization

*Initializers* set the initial value that is stored in an object (variables, arrays, structures, and so on). If you don't initialize an object, and it has static duration, it will be initialized by default in the following manner:

*If it has automatic storage duration, its value is indeterminate.*

- to zero if it is of an arithmetic type
- to null if it is a pointer type

The syntax for initializers is as follows:

```
initializer
= expression
= {initializer-list} <,>
(expression list)
```



```
initializer-list
expression
initializer-list, expression
{initializer-list} <,>
```

Rules governing initializers are:

1. The number of initializers in the initializer list cannot be larger than the number of objects to be initialized.

2. The item to be initialized must be an object type or an array of unknown size.
3. All expressions must be constants if they appear in one of these places:
  - a. in an initializer for an object that has static duration (not required for C++)
  - b. in an initializer list for an array, structure, or union (expressions using **sizeof** are also allowed)
4. If a declaration for an identifier has block scope, and the identifier has external or internal linkage, the declaration cannot have an initializer for the identifier.
5. If there are fewer initializers in a brace-enclosed list than there are members of a structure, the remainder of the structure is initialized implicitly in the same way as objects with static storage duration.

Scalar types are initialized with a single expression, which can optionally be enclosed in braces. The initial value of the object is that of the expression; the same constraints for type and conversions apply as for simple assignments.

For unions, a brace-enclosed initializer initializes the member that first appears in the union's declaration list. For structures or unions with automatic storage duration, the initializer must be one of the following:

- an initializer list as described in the following section
- a single expression with compatible union or structure type. In this case, the initial value of the object is that of the expression.

#### Arrays, structures, and unions

You initialize arrays and structures (at declaration time, if you like) with a brace-enclosed list of initializers for the members or elements of the object in question. The initializers are given in increasing array subscript or member order. You initialize unions with a brace-enclosed initializer for the first member of the union. For example, you could declare an array *days*, intended to count how many times each day of the week appears in a month (and assuming that each day will appear at least once), as follows:

```
int days[7] = { 1, 1, 1, 1, 1, 1, 1 }
```

Use these rules to initialize character arrays and wide character arrays:

1. You can initialize arrays of character type with a literal string, optionally enclosed in braces. Each character in the string, including the null terminator, initializes successive elements in the array. For example, you could declare

```
char name[] = { "Unknown" };
```

which sets up an eight-element array, whose elements are 'U' (for *name*[0]), 'n' (for *name*[1]), and so on (and including a null terminator).

2. You can initialize a wide character array (one that is compatible with **wchar\_t**) by using a wide string literal, optionally enclosed in braces. As with character arrays, the codes of the wide string literal initialize successive elements of the array.

Here is an example of a structure initialization:

```
struct mystruct {
    int i;
    char str[21];
    double d;
} s = { 20, "Borland", 3.141 };
```

Complex members of a structure, such as arrays or structures, can be initialized with suitable expressions inside nested braces. You can eliminate the braces, but you must follow certain rules, and it isn't recommended practice.

---

## Simple declarations

Simple declarations of variable identifiers have the following pattern:

```
data-type var1 <=init1>, var2 <=init2>, ...;
```

where *var1*, *var2*,... are any sequence of distinct identifiers with optional initializers. Each of the variables is declared to be of type *data-type*. For example,

```
int x = 1, y = 2;
```

creates two integer variables called *x* and *y* (and initializes them to the values 1 and 2, respectively).

These are all defining declarations; storage is allocated and any optional initializers are applied.

The initializer for an automatic object can be any legal expression that evaluates to an assignment-compatible value for the type of the variable involved. Initializers for static objects must be constants or constant expressions.



In C++, an initializer for a static object can be any expression involving constants and previously declared variables and functions.

---

## Storage class specifiers

A *storage class specifier*, or a type specifier, must be present in a declaration. The storage class specifiers can be one of the following:

<b>auto</b>	<b>register</b>	<b>typedef</b>
<b>extern</b>	<b>static</b>	

- Use of storage class specifier **auto**

The storage class specifier **auto** is used only with local scope variable declarations. It conveys local (automatic) duration, but since this is the default for all local scope variable declarations, its use is rare.
- Use of storage class specifier **extern**

The storage class specifier **extern** can be used with function and variable file scope and local scope declarations to indicate external linkage. With file scope variables, the default storage class specifier is **extern**. When used with variables, **extern** indicates that the variable has static duration. (Remember that functions always have static duration.)
- Use of storage class specifier **register**

The storage class specifier **register** is allowed only for local variable and function parameter declarations. It is equivalent to **auto**, with the added excitement that a request is made to the compiler that the variable should be allocated to a register if possible. The allocation of a register can significantly reduce the size and improve the performance of programs in many situations. However, since Turbo C++ does a good job of placing variables in registers, it is rarely necessary to use the **register** keyword.

Turbo C++ lets you select register variable options from the **Options | Compiler | Optimizations** dialog box. If you check **Automatic**, Turbo C++ will try to allocate registers even if you have not used the **register** storage class specifiers.
- Use of storage class specifier **static**

The storage class specifier **static** can be used with function and variable file scope and local scope declarations to indicate internal linkage. **static** also indicates that the variable has static duration. In the absence of constructors or explicit initializers, static variables are initialized with 0 or null.



In C++, a static data member of a class has the same value for all instances of a class. A static function member of a class can be invoked independently of any class instance.

#### Use of storage class specifier **typedef**

The keyword **typedef** indicates that you are defining a new data type specifier rather than declaring an object. **typedef** is included as a storage class specifier because of syntactical rather than functional similarities.

```
static long int biggy;  
typedef long int BIGGY;
```

The first declaration creates a 32-bit, **long int**, static-duration object called *biggy*. The second declaration establishes the identifier *BIGGY* as a new type specifier, but does not create any run-time object. *BIGGY* can be used in any subsequent declaration where a type specifier would be legal. For example,

```
extern BIGGY salary;
```

has the same effect as

```
extern long int salary;
```

Although this simple example can be achieved by `#define BIGGY long int`, more complex **typedef** applications achieve more than is possible with textual substitutions.

#### **Important!**

**typedef** does not create new data types; it merely creates useful mnemonic synonyms or aliases for existing types. It is especially valuable in simplifying complex declarations:

```
typedef double (*PFD)();  
PFD array_pfd[10];  
/* array_pfd is an array of 10 pointers to functions  
   returning double */
```

You can't use **typedef** identifiers with other data-type specifiers:

```
unsigned BIGGY pay;      /* ILLEGAL */
```

---

## Modifiers

In addition to the storage class specifier keywords, a declaration can use certain *modifiers* to alter some aspect of the identifier/object mapping. The modifiers available with Turbo C++ are summarized in the next table.

Table 1.16  
Turbo C++ modifiers

*C++ extends **const** and **volatile** to include classes and member functions.*

<b>Modifier</b>	<b>Use with</b>	<b>Usage</b>
const	Variables only	Prevents changes to object.
volatile	Variables only	Prevents register allocation and some optimization. Warns compiler that object may be subject to outside change during evaluation.
<b><i>Turbo C++ extensions</i></b>		
cdecl	Functions	Forces C argument-passing convention.
cdecl	Variables	Forces global identifier case-sensitivity and leading underscores.
pascal	Functions	Forces Pascal argument-passing convention.
pascal	Variables	Forces global identifier case-insensitivity with no leading underscores.
interrupt	Functions	Function compiles with the additional register-housekeeping code needed when writing interrupt handlers.
near, far, huge	Pointer variables	Overrides the default pointer type specified by the current memory model.
_cs, _ds, _es, _seg, _ss	Pointer variables	Segment pointers; see page 199.
near, far, huge	Functions	Overrides the default function type specified by the current memory model.
near, far	Variables	Directs the placement of the object in memory.
_export	Functions	OS/2 only. Ignored by Turbo C++.
_loadds	Functions	Sets DS to point to the current data segment.
_saveregs	Functions	Preserves all register values (except for return values) during execution of the function.

## The `const` modifier

The **`const`** modifier prevents any assignments to the object or any other side effects, such as increment or decrement. A **`const`** pointer cannot be modified, though the object to which it points can be. Consider the following examples:

The modifier **`const`** used by itself is equivalent to **`const int`**.

```
const float pi = 3.1415926;
const maxint = 32767;
char *const str = "Hello, world"; // A constant pointer
char const *str2 = "Hello, world"; /* A pointer to a constant char
*/
```

Given these, the following statements are illegal:

```
pi = 3.0;           /* Assigns a value to a const */
i = maxint++;      /* Increments a const */
str = "Hi, there!"; /* Points str to something else */
```

Note, however, that the function call `strcpy(str, "Hi, there!")` is legal, since it does a character-by-character copy from the string literal "Hi, there!" into the memory locations pointed to by `str`.



In C++, **`const`** also hides the **`const`** object and prevents external linkage. You need to use **`extern const`**. A pointer to a **`const`** can't be assigned to a pointer to a non-**`const`** (otherwise, the **`const`** value could be assigned to using the non-**`const`** pointer). For example,

```
char *str3 = str2 /* disallowed */
```

## The interrupt function modifier

The **`interrupt`** modifier is specific to Turbo C++. **`interrupt`** functions are designed to be used with the 8086/8088 interrupt vectors. Turbo C++ will compile an **`interrupt`** function with extra function entry and exit code so that registers AX, BX, CX, DX, SI, DI, ES, and DS are preserved. The other registers (BP, SP, SS, CS, and IP) are preserved as part of the C-calling sequence or as part of the interrupt handling itself. The function will use an **`iret`** instruction to return, so that the function can be used to service hardware or software interrupts. Here is an example of a typical **`interrupt`** definition:

```
void interrupt myhandler()
{
    ...
}
```

You should declare interrupt functions to be of type **`void`**. Interrupt functions can be declared in any memory model. For all memory models except huge, DS is set to the program data segment. For the huge model, DS is set to the module's data segment.

## The volatile modifier

In C++, **volatile** has a special meaning for class member functions. If you've declared a volatile object, you can only use its volatile member functions.

The **volatile** modifier indicates that the object may be modified; not only by you, but also by something outside of your program, such as an interrupt routine or an I/O port. Declaring an object to be **volatile** warns the compiler not to make assumptions concerning the value of the object while evaluating expressions containing it, since the value could (in theory) change at any moment. It also prevents the compiler from making the variable a register variable.

```
volatile int ticks;
interrupt timer()
{
    ticks++;
}

wait(int interval)
{
    ticks = 0;
    while (ticks < interval);    // Do nothing
}
```

These routines (assuming **timer** has been properly associated with a hardware clock interrupt) implement a timed wait of ticks specified by the argument *interval*. A highly optimizing compiler might not load the value of *ticks* inside the test of the **while** loop, since the loop doesn't change the value of *ticks*.

## The cdecl and pascal modifiers

Turbo C++ allows your programs to easily call routines written in other languages, and vice versa. When you mix languages like this, you have to deal with two important issues: identifiers and parameter passing.

In Turbo C++, all global identifiers are saved in their original case (lower, upper, or mixed) with an underscore (**\_**) prepended to the front of the identifier, unless you have selected the **-u-** option (Generate Underbars...Off in the Options | Compiler | Code Generation dialog box).

Page 32 tells how to use **extern**, which allows C names to be referenced from a C++ program.

### **pascal**

In Pascal, global identifiers are not saved in their original case, nor are underscores prepended to them. Turbo C++ lets you declare any identifier to be of type **pascal**; the identifier is converted to uppercase, and

no underscore is prepended. (If the identifier is a function, this also affects the parameter-passing sequence used; see "Function type modifiers," page 51, for more details.)

*The **-p** compiler option (Calling Convention... Pascal in the Options I Compiler I Code Generation dialog box) causes all functions (and pointers to those functions) to be treated as if they were of type **pascal**.*

The **pascal** modifier is specific to Turbo C++; it is intended for functions (and pointers to functions) that use the Pascal parameter-passing sequence. Also, functions declared to be of type **pascal** can still be called from C routines, so long as the C routine sees that the function is of type **pascal**.

```
pascal putnums(int i, int j, int k)
{
    printf("And the answers are: %d, %d, and %d\n",i,j,k);
}
```

Functions of type **pascal** cannot take a variable number of arguments, unlike functions such as **printf**. For this reason, you cannot use an ellipsis (...) in a **pascal** function definition.

### **cdecl**

***main** must be declared as **cdecl**; this is because the C start-up code always tries to call **main** with the C calling convention.*

Once you have compiled with the **-p** option, you may want to ensure that certain identifiers have their case preserved and keep the underscore on the front, especially if they're C identifiers from another file. You can do so by declaring those identifiers to be **cdecl**. (This also has an effect on parameter passing for functions).

Like **pascal**, the **cdecl** modifier is specific to Turbo C++. It is used with functions and pointers to functions. It overrides the **-p** compiler directive and allows a function to be called as a regular C function. For example, if you were to compile the previous program with the **-p** option set but wanted to use **printf**, you might do something like this:

```
extern cdecl printf();
putnums(int i, int j, int k);

cdecl main()
{
    putnums(1,4,9);
}

putnums(int i, int j, int k)
{
    printf("And the answers are: %d, %d, and %d\n",i,j,k);
}
```

If you compile a program with the **-p** option, all functions used from the run-time library will need to have **cdecl** declarations. If you look

at the header files (such as `stdio.h`), you'll see that every function is explicitly defined as **cdecl** in anticipation of this.

The pointer modifiers Turbo C++ has eight modifiers that affect the indirection operator (\*); that is, they modify pointers to data. These are **near**, **far**, **huge**, **\_cs**, **\_ds**, **\_es**, **\_seg**, and **\_ss**.

C lets you compile using one of several memory models. The model you use determines (among other things) the internal format of pointers. For example, if you use a small data model (tiny, small, medium), all data pointers contain a 16-bit offset from the data segment (DS) register. If you use a large data model (compact, large, huge), all pointers to data are 32 bits long and give both a segment address and an offset.

Sometimes, when using one size of data model, you want to declare a pointer to be of a different size or format than the current default. You do so using the pointer modifiers.

See the discussion starting on page 192 in Chapter 4 for an indepth explanation of **near**, **far**, and **huge** pointers, and page 193 for a description of normalized pointers. Also see the discussion starting on page 199 for more on **\_cs**, **\_ds**, **\_es**, **\_seg**, and **\_ss**.

Function type modifiers The **near**, **far**, and **huge** modifiers can also be used as function type modifiers; that is, they can modify functions and function pointers as well as data pointers. In addition, you can use the **\_export**, **\_loadregs**, and **\_saveregs** modifiers to modify functions.

The **near**, **far**, and **huge** function modifiers can be combined with **cdecl** or **pascal**, but not with **interrupt**.

Functions of type **huge** are useful when interfacing with code in assembly language that doesn't use the same memory allocation as Turbo C++.

A non-**interrupt** function can be declared to be **near**, **far**, or **huge** in order to override the default settings for the current memory model.

A **near** function uses **near** calls; a **far** or **huge** function uses **far** call instructions.

In the tiny, small, and compact memory models, an unqualified function defaults to type **near**. In the medium and large models, an unqualified function defaults to type **far**. In the huge memory model, it defaults to type **huge**.

A **huge** function is the same as a **far** function, except that the DS register is set to the data segment address of the source module when a **huge** function is entered, but left unset for a **far** function.

The **\_export** modifier is parsed, but ignored. It provides compatibility with source code written for OS/2. The **\_export** modifier has no significance for DOS programs.

The **\_loadds** modifier indicates that a function should set the DS register, just as a huge function does, but does not imply **near** or **far** calls. Thus, **\_loadds far** is equivalent to **huge**.

The **\_saveregs** modifier causes the function to preserve all register values and restore them before returning (except for explicit return values passed in registers such as AX or DX).

The **\_loadds** and **\_saveregs** modifiers are useful for writing low-level interface routines, such as mouse support routines.

## Complex declarations and declarators

See Table 1.9 on page 35 for the declarator syntax. The definition covers both identifier and function declarators.

---

Simple declarations have a list of comma-delimited identifiers following the optional storage class specifiers, type specifiers, and other modifiers.

A complex declaration uses a comma-delimited list of declarators following the various specifiers and modifiers. Within each declarator, there exists just one identifier, namely the identifier being declared. Each of the declarators in the list is associated with the leading storage class and type specifier.

The format of the declarator indicates how the declared *dname* is to be interpreted when used in an expression. If **type** is any type, and *storage class specifier* is any storage class specifier, and if *D1* and *D2* are any two declarators, then the declaration

```
storage class specifier type D1, D2;
```

indicates that each occurrence of *D1* or *D2* in an expression will be treated as an object of type **type** and storage class *storage class specifier*. The type of the *dname* embedded in the declarator will be some phrase containing **type**, such as "**type**," "pointer to **type**," "array of **type**," "function returning **type**," or "pointer to function returning **type**," and so on.

For example, in the declarations

```
int n, nao[], naf[3], *pn, *apn[], (*pan)[], &nr=n;  
int f(void), *fnp(void), (*pfn)(void);
```

each of the declarators could be used as rvalues (or possibly lvalues in some cases) in expressions where a single **int** object would be appropriate. The types of the embedded identifiers are derived from their declarators as follows:

Table 1.17: Complex declarations

Declarator syntax	Implied <i>type</i> of <i>name</i>	Example
<b>type</b> <i>name</i> ;	<b>type</b>	int count;
<b>type</b> <i>name</i> [];	(open) array of <b>type</b>	int count[];
<b>type</b> <i>name</i> [3];	Fixed array of three elements, all of <b>type</b> ( <i>name</i> [0], <i>name</i> [1], and <i>name</i> [2])	int count[3];
<b>type</b> * <i>name</i> ;	Pointer to <b>type</b>	int *count;
<b>type</b> * <i>name</i> [];	(open) array of pointers to <b>type</b>	int *count[];
<b>type</b> *( <i>name</i> []);	Same as above	int *(count []);
<b>type</b> (* <i>name</i> ) [];	Pointer to an (open) array of <b>type</b>	int (*count) [];
<b>type</b> & <i>name</i> ;	Reference to <b>type</b> (C++ only)	int &count;
<b>type</b> <i>name</i> ();	Function returning <b>type</b>	int count();
<b>type</b> * <i>name</i> ();	Function returning pointer to <b>type</b>	int *count();
<b>type</b> *( <i>name</i> ());	Same as above	int *(count());
<b>type</b> (* <i>name</i> ) ();	Pointer to function returning <b>type</b>	int (*count) ();

Note the need for parentheses in (\**name*)[] and (\**name*)(), since the precedence of both the array declarator [] and the function declarator () is higher than the pointer declarator \*. The parentheses in \*(*name*[]) are optional.

## Pointers

See page 80 for a discussion of referencing and de-referencing.

Pointers fall into two main categories: pointers to objects and pointers to functions. Both types of pointers are special objects for holding memory addresses.

The two pointer classes have distinct properties, purposes, and rules for manipulation, although they do share certain Turbo C++ operations. Generally speaking, pointers to functions are used to access functions and to pass functions as arguments to other functions; performing arithmetic on pointers to functions is not

allowed. Pointers to objects, on the other hand, are regularly incremented and decremented as you scan arrays or more complex data structures in memory.

Although pointers contain numbers with most of the characteristics of unsigned integers, they have their own rules and restrictions for assignments, conversions, and arithmetic. The examples in the next few sections illustrate these rules and restrictions.

---

## Pointers to objects

A pointer of type “pointer to object of *type*” holds the address of (that is, points to) an object of *type*. Since pointers are objects, you can have a pointer pointing to a pointer (and so on). Other objects commonly pointed at include arrays, structures, unions, and classes.

The size of pointers to objects is dependent on the memory model and the size and disposition of your data segments, possibly influenced by the optional pointer modifiers (discussed starting on page 51).

---

## Pointers to functions

A pointer to a function is best thought of as an address, usually in a code segment, where that function’s executable code is stored; that is, the address to which control is transferred when that function is called. The size and disposition of your code segments is determined by the memory model in force, which in turn dictates the size of the function pointers needed to call your functions.

A pointer to a function has a type called “pointer to function returning *type*,” where *type* is the function’s return type.



Under C++, which has stronger type checking, a pointer to a function has type “pointer to function taking argument types *type* and returning *type*.” In fact, under C, a function defined with argument types will also have this narrower type. For example,

```
void (*func)();
```

In C, this is a pointer to a function returning nothing. In C++, it’s a pointer to a function taking no arguments and returning nothing. In this example,

```
void (*func)(int);
```

*\*func* is a pointer to a function taking an **int** argument and returning nothing.

## Pointer

### declarations

See page 39 for details on **void**.

A pointer must be declared as pointing to some particular type, even if that type is **void** (which really means a pointer to anything). Once declared, though, a pointer can usually be reassigned to point to an object of another type. Turbo C++ lets you reassign pointers like this without typecasting, but the compiler will warn you unless the pointer was originally declared to be of type pointer to **void**. And in C, but not C++, you can assign a **void\*** pointer to a non-**void\*** pointer.

If **type** is any predefined or user-defined type, including **void**, the declaration

```
type *ptr; /* Danger--uninitialized pointer */
```

declares *ptr* to be of type “pointer to **type**.” All the scoping, duration, and visibility rules apply to the *ptr* object just declared.

A null pointer value is an address that is guaranteed to be different from any valid pointer in use in a program. Assigning the integer constant 0 to a pointer assigns a null pointer value to it.

The mnemonic **NULL** (defined in the standard library header files, such as `stdio.h`) can be used for legibility. All pointers can be successfully tested for equality or inequality to **NULL**.

The pointer type “pointer to void” must not be confused with the null pointer. The declaration

```
void *vptr;
```

declares that *vptr* is a generic pointer capable of being assigned to by any “pointer to **type**” value, including null, without complaint. Assignments without proper casting between a “pointer to **type1**” and a “pointer to **type2**,” where **type1** and **type2** are different types, can invoke a compiler warning or error. If **type1** is a function and **type2** isn’t (or vice versa), pointer assignments are illegal. If **type1** is a pointer to **void**, no cast is needed. If **type2** is a pointer to **void**, under C, no cast is needed.

Assignment restrictions also apply to pointers of different sizes (**near**, **far**, and **huge**). You can assign a smaller pointer to a larger one without error, but you can’t assign a larger pointer to a smaller one unless you are using an explicit cast. For example,

```
char near *ncp;  
char far *fcp;  
char huge *hcp;
```

```

fcp = ncp;           // legal
hcp = fcp;          // legal
fcp = hcp;          // not legal
ncp = fcp;          // not legal
ncp = (char near*)fcp; // now legal

```

## Pointers and constants

A pointer or the pointed-at object can be declared with the **const** modifier. Anything declared as a **const** cannot be assigned to. It is also illegal to create a pointer that might violate the nonassignability of a constant object. Consider the following examples:

```

int i;               // i is an int
int * pi;            // pi is a pointer to int (uninitialized)
int * const cp = &i; // cp is a constant pointer to int.
const int ci = 7;    // ci is a constant int
const int * pci;     // pci is a pointer to constant int
const int * const cpc = &ci; // cpc is a constant pointer to a
                          // constant int

```

The following assignments are legal:

```

i = ci;              // Assign const-int to int
*cp = ci;            // Assign const-int to
                    // object-pointed-at-by-a-const-pointer
++pci;              // Increment a pointer-to-const
pci = cpc;          // Assign a const-pointer-to-a-const to a
                    // pointer-to-const

```

The following assignments are illegal:

```

ci = 0;             // NO--cannot assign to a const-int
ci--;               // NO--cannot change a const-int
*pci = 3;           // NO--cannot assign to an object
                    // pointed at by pointer-to-const
cp = &ci;           // NO--cannot assign to a const-pointer,
                    // even if value would be unchanged
cpc++;              // NO--cannot change const-pointer
pi = pci;           // NO--if this assignment were allowed,
                    // you would be able to assign to *pci
                    // (a const value) by assigning to *pi.

```

Similar rules apply to the **volatile** modifier. Note that **const** and **volatile** can both appear as modifiers to the same identifier.

## Pointer arithmetic

*The internal arithmetic performed on pointers depends on the memory model in force and the presence of any overriding pointer modifiers. The difference between two pointers only has meaning if both pointers point into the same array.*

Pointer arithmetic is limited to addition, subtraction, and comparison. Arithmetical operations on object pointers of type “pointer to **type**” automatically take into account the size of **type**; that is, the number of bytes needed to store a **type** object.

When performing arithmetic with pointers, it is assumed that the pointer points to an array of objects. Thus, if a pointer is declared to point to **type**, adding an integral value to the pointer advances the pointer by that number of objects of **type**. If **type** has size 10 bytes, then adding an integer 5 to a pointer to **type** advances the pointer 50 bytes in memory. The difference has as its value the number of array elements separating the two pointer values. For example, if *ptr1* points to the third element of an array, and *ptr2* points to the tenth element, then the result of *ptr2* - *ptr1* would be 7.

When an integral value is added to or subtracted from a “pointer to **type**,” the result is also of type “pointer to **type**.” If **type** is a nonarray object, a pointer operand is treated as though it were a pointer to the first element of an “array of **type**” of length **sizeof (type)**.

There is no such element as “pointer to one past the last element”, of course, but a pointer is allowed to assume such a value. If *P* points to the last array element, *P* + 1 is legal, but *P* + 2 is undefined. If *P* points to one past the last array element, *P* - 1 is legal, giving a pointer to the last element. However, applying the indirection operator \* to a “pointer to one past the last element” leads to undefined behavior.

Informally, you can think of *P* + *n* as advancing the pointer by (*n* \* **sizeof(type)**) bytes, as long as the pointer remains within the legal range (first element to one beyond the last element).

Subtracting two pointers to elements of the same array object gives an integral value of type *ptrdiff\_t* defined in `stddef.h` (**signed long** for huge and far pointers; **signed int** for all others). This value represents the difference between the subscripts of the two referenced elements, provided it is in the range of *ptrdiff\_t*. In the expression *P1* - *P2*, where *P1* and *P2* are of type pointer to **type** (or pointer to qualified **type**), *P1* and *P2* must point to existing elements or to one past the last element. If *P1* points to the *i*-th element, and *P2* points to the *j*-th element, *P1* - *P2* has the value (*i* - *j*).

## Pointer conversions

---

Pointer types can be converted to other pointer types using the typecasting mechanism:

```
char *str;
int *ip;
str = (char *)ip;
```

More generally, the cast (*type\**) will convert a pointer to type “pointer to *type*.”

---

## C++ reference declarations

C++ reference types are closely related to pointer types. *Reference types* create aliases for objects and let you pass arguments to functions by reference. Traditional C passes arguments only by *value*. In C++ you can pass arguments by value or by reference. See page 98, “Referencing,” for complete details.

---

## Arrays

*This section starts the phrase-structure grammar part of this chapter; see page 4 for a description of the difference between lexical and phrase-structure grammars.*

The declaration

```
type declarator [<constant-expression>]
```

declares an array composed of elements of **type**. An array in C consists of a contiguous region of storage exactly large enough to hold all of its elements.

If an expression is given in an array declarator, it must evaluate to a positive constant integer. The value is the number of elements in the array. Each of the elements of an array is numbered from 0 through the number of elements minus one.

Multidimensional arrays are constructed by declaring arrays of array type. Thus, a two-dimensional array of five rows and seven columns called *alpha* is declared as

```
type alpha [5] [7];
```

In certain contexts, the first array declarator of a series may have no expression inside the brackets. Such an array is of indeterminate size. The contexts where this is legitimate are ones in which the size of the array is not needed to reserve space. For example, an **extern** declara-

tion of an array object does not need the exact dimension of the array, nor does an array function parameter. As a special extension to ANSI C, Turbo C++ also allows an array of indeterminate size as the final member of a structure. Such an array does not increase the size of the structure, except that padding can be added to ensure that the array is properly aligned. These structures are normally used in dynamic allocation, and the size of the actual array needed must be explicitly added to the size of the structure in order to properly reserve space.

Except when it is the operand of a **sizeof** or **&** operator, an array type expression is converted to a constant pointer to the first element of the array.

## Functions

---

Functions are central to Turbo C++ programming. Languages such as Pascal distinguish between procedure and function. Turbo C++ functions play both roles.

### Declarations and definitions

---

Each program must have a single external function named **main** marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions are **external** by default and are normally accessible from any file in the program. They can be restricted by using the **static** storage class specifier (see page 32).

Functions are defined in your source files or made available by linking precompiled libraries.

*In C++ you must always use function prototypes. We recommend that you also always use them in C.*

A given function can be declared several times in a program, provided the declarations are compatible. Nondefining function declarations using the function prototype format provide Turbo C++ with detailed parameter information, allowing better control over argument number and type checking, and type conversions.

Excluding C++ function overloading, only one definition of any given function is allowed. The declarations, if any, must also match this definition. (The essential difference between a definition and a declaration is that the definition has a function body.)

## Declarations and prototypes

In the original Kernighan and Ritchie style of declaration, a function could be implicitly declared by its appearance in a function call, or explicitly declared as follows:

In C++, this declaration means `<type> func(void)`

```
<type> func()
```

where **type** is the optional return type defaulting to **int**. A function can be declared to return any type except an array or function type. This approach does not allow the compiler to check that the type or number of arguments used in a function call match the declaration.

You can enable a warning within the IDE or with the command-line compiler:  
"Function called without a prototype."

This problem was eased by the introduction of function prototypes with the following declaration syntax:

```
<type> func(parameter-declarator-list);
```

Declarators specify the type of each function parameter. The compiler uses this information to check function calls for validity. The compiler is also able to coerce arguments to the proper type. Suppose you have the following code fragment:

```
long lmax(long v1, long v2); /* prototype */

main()
{
    int limit = 32;
    char ch = 'A';

    long mval;

    mval = lmax(limit,ch);    /* function call */
}
```

Since it has the function prototype for **lmax**, this program converts *limit* and *ch* to **long**, using the standard rules of assignment, before it places them on the stack for the call to **lmax**. Without the function prototype, *limit* and *ch* would have been placed on the stack as an integer and a character, respectively; in that case, the stack passed to **lmax** would not match in size or content what **lmax** was expecting, leading to problems. The classic declaration style does not allow any checking of parameter type or number, so using function prototypes aids greatly in tracking down programming errors.

Function prototypes also aid in documenting code. For example, the function **strcpy** takes two parameters: a source string and a destination string. The question is, which is which? The function prototype

```
char *strcpy(char *dest, char *source);
```

makes it clear. If a header file contains function prototypes, then you can print that file to get most of the information you need for writing programs that call those functions. If you include an identifier in a prototype parameter, it is only used for any later error messages involving that parameter; it has no other effect.

A function declarator with parentheses containing the single word **void** indicates a function that takes no arguments at all:

```
func(void);
```



In C++, **func()** also declares a function taking no arguments.

*stdarg.h contains macros that you can use in user-defined functions with variable numbers of parameters.*

A function prototype normally declares a function as accepting a fixed number of parameters. For C functions that accept a variable number of parameters (such as **printf**), a function prototype can end with an ellipsis (...), like this:

```
f(int *count, long total, ...)
```

With this form of prototype, the fixed parameters are checked at compile time, and the variable parameters are passed with no type checking.

Here are some more examples of function declarators and prototypes:

```
int f();          /* In C, a function returning an int with no
                  information about parameters. This is the K&R
                  "classic style." */

int f();          /* In C++, a function taking no arguments */

int f(void);      /* A function returning an int that takes no
                  parameters. */

int p(int, long); /* A function returning an int that accepts two
                  parameters: the first, an int; the second, a long.
                  */

int pascal q(void); /* A pascal function returning an int that takes no
                  parameters at all. */

char far *s(char *source, int kind); /* A function returning a far
                  pointer to a char and accepting two parameters: the
                  first, a pointer to a char; the second, an int. */

int printf(char *format, ...); /* A function returning an int and
                  accepting a pointer to a char fixed parameter and any
                  number of additional parameters of unknown type. */

int (*fp)(int);  /* A pointer to a function returning an int and
                  accepting a single int parameter. */
```

## Definitions

---

The general syntax for external function definitions is given in the following table:

Table 1.18  
External function definitions

---

<i>file</i>
<i>external-definition</i>
<i>file external-definition</i>
<i>external-definition:</i>
<i>function-definition</i>
<i>declaration</i>
<i>asm-statement</i>
<i>function-definition:</i>
<i>&lt;declaration-specifiers&gt; declarator &lt;declaration-list&gt; compound-statement</i>

---

In general, a function definition consists of the following sections (the grammar allows for more complicated cases):

You can intermix elements from 1 and 2.

1. Optional storage class specifiers: **extern** or **static**. The default is **extern**.
2. A return type, possibly **void**. The default is **int**.
3. Optional modifiers: **pascal**, **cdecl**, **interrupt**, **near**, **far**, **huge**. The defaults depend on the memory model and compiler option settings.
4. The name of the function.
5. A parameter declaration list, possibly empty, enclosed in parentheses. In C, the preferred way of showing an empty list is **func(void)**. The old style of **func()** is legal in C but antiquated and possibly unsafe. In C++, you'll get a warning.
6. A function body representing the code to be executed when the function is called.

---

## Formal parameter declarations

The formal parameter declaration list follows a similar syntax to that of the declarators found in normal identifier declarations. Here are a few examples:

```
int func(void) { // no args
int func(T1 t1, T2 t2, T3 t3=1) { // three simple parameters, one
// with default argument
int func(T1* ptr1, T2& tref) { // a pointer and a reference arg
```

```

int func(register int i) {           // request register for arg
int func(char *str,...) {           /* one string arg with a variable
    number of other args, or with a fixed number of args with
    varying types */

```



In C++, you can give default arguments as shown. Parameters with default values must be the last arguments in the parameter list. The arguments' types can be scalars, structures, unions, enumerations; pointers or references to structures and unions; or pointers to functions or classes.

The ellipsis (...) indicates that the function will be called with different sets of arguments on different occasions. The ellipsis can follow a sublist of known argument declarations. This form of prototype reduces the amount of checking the compiler can make.

The parameters declared all enjoy automatic scope and duration for the duration of the function. The only legal storage class specifier is **register**.

The **const** and **volatile** modifiers can be used with formal argument declarators.

## Function calls and argument conversions

A function is called with actual arguments placed in the same sequence as their matching formal arguments. The actual arguments are converted as if by initialization to the declared types of the formal arguments.

Here is a summary of the rules governing how Turbo C++ deals with language modifiers and formal parameters in function calls, both with and without prototypes:

1. The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function.
2. A function may modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine, except for reference arguments in C++.

When a function prototype has not been previously declared, Turbo C++ converts integral arguments to a function call according to the integral widening (expansion) rules described in the section "Standard conversions," starting on page 41. When a function prototype is

in scope, Turbo C++ converts the given argument to the type of the declared parameter as if by assignment.

When a function prototype includes an ellipsis (...), Turbo C++ converts all given function arguments as in any other prototype (up to the ellipsis). The compiler widens any arguments given beyond the fixed parameters, according to the normal rules for function arguments without prototypes.

If a prototype is present, the number of arguments must match (unless an ellipsis is present in the prototype). The types need only be compatible to the extent that an assignment can legally convert them. You can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.

**Important!** If your function prototype does not match the actual function definition, Turbo C++ will detect this if and only if that definition is in the same compilation unit as the prototype. If you create a library of routines with a corresponding header file of prototypes, consider including that header file when you compile the library, so that any discrepancies between the prototypes and the actual definitions will be caught. C++ provides type-safe linkage, so differences between expected and actual parameters will be caught by the linker.

## Structures

---

*Structure initialization is discussed on page 42.*

A *structure* is a derived type usually representing a user-defined collection of named members (or components). The members can be of any type, either fundamental or derived (with some restrictions to be noted later), in any sequence. In addition, a structure member can be a bit field type not allowed elsewhere. The Turbo C++ structure type lets you handle complex data structures almost as easily as single variables.



In C++, a structure type is treated as a class type (with certain differences: Default access is public, and the default for the base class is also public). This allows more sophisticated control over access to structure members by using the C++ access specifiers: **public** (the default), **private**, and **protected**. Apart from these optional access mechanisms, and from exceptions as noted, the following discussion on structure syntax and usage applies equally to C and C++ structures.

Structures are declared using the keyword **struct**. For example,

```
struct mystruct { ... }; // mystruct is the structure tag
```

```

...
struct mystruct s, *ps, arrs[10];
/* s is type struct mystruct; ps is type pointer to struct mystruct;
   arrs is array of struct mystruct. */

```

## Untagged structures and typedefs

*Untagged structure and union members are ignored during initialization.*

If you omit the structure tag, you can get an *untagged* structure. You can use untagged structures to declare the identifiers in the comma-delimited *struct-id-list* to be of the given structure type (or derived from it), but you cannot declare additional objects of this type elsewhere:

```

struct { ... } s, *ps, arrs[10]; // untagged structure

```

It is possible to create a **typedef** while declaring a structure, with or without a tag:

```

typedef struct mystruct { ... } MYSTRUCT;
MYSTRUCT s, *ps, arrs[10]; // same as struct mystruct s, etc.
typedef struct { ... } YRSTRUCT; // no tag
YRSTRUCT y, *yp, array[20];

```

You don't usually need both a tag and a **typedef**: Either can be used in structure declarations.

## Structure member declarations

The *member-decl-list* within the braces declares the types and names of the structure members using the declarator syntax shown in Table 1.11 on page 36.

A structure member can be of any type, with two exceptions:

1. The member type cannot be same as the **struct** type being currently declared:

```

struct mystruct { mystruct s } s1, s2; // illegal

```

A member can be a pointer to the structure being declared, as in the following example:

```

struct mystruct { mystruct *ps } s1, s2; // OK

```

Also, a structure can contain previously defined structure types when declaring an instance of a declared structure.

2. Except in C++, a member cannot have the type "function returning..." but the type "pointer to function returning..." is allowed. In C++, a **struct** can have member functions.

*You can omit the **struct** keyword in C++.*

## Structures and functions

A function can return a structure type or a pointer to a structure type:

```
mystruct func1(void); // func1() returns a structure
mystruct *func2(void); // func2() returns pointer to structure
```

A structure can be passed as an argument to a function in the following ways:

```
void func1(mystruct s); // directly
void func2(mystruct *sptr); // via a pointer
void func3(mystruct &sref); // as a reference (C++ only)
```

## Structure member access

Structure and union members are accessed using the selection operators `.` and `->`. Suppose that the object `s` is of struct type **S**, and `sptr` is a pointer to **S**. Then if `m` is a member identifier of type **M** declared in **S**, the expressions `s.m` and `sptr->m` are of type **M**, and both represent the member object `m` in `s`. The expression `s->sptr` is a convenient synonym for `(*sptr).m`.

The operator `.` is called the direct member selector; the operator `->` is called the indirect (or pointer) member selector; for example,

```
struct mystruct {
    int i;
    char str[21];
    double d;
} s, *sptr=&s;
...
s.i = 3; // assign to the i member of mystruct s
sptr->d = 1.23; // assign to the d member of mystruct s
```

The expression `s.m` is an lvalue, provided that `s` is not an lvalue and `m` is not an array type. The expression `sptr->m` is an lvalue unless `m` is an array type.

If structure `B` contains a field whose type is structure `A`, the members of `A` can be accessed by two applications of the member selectors:

```
struct A {
    int j;
    double x;
};

struct B {
    int i;
```

```

    struct A a;
    double d;
} s, *sptr;
...
s.i = 3;           // assign to the i member of B
s.a.j = 2;        // assign to the j member of A
sptr->d = 1.23;    // assign to the d member of B
(sptr->a).x = 3.14 // assign to x member of A

```

Each structure declaration introduces a unique structure type, so that in

```

struct A {
    int i,j;
    double d;
} a, a1;

struct B {
    int i,j;
    double d;
} b;

```

the objects *a* and *a1* are both of type *struct A*, but the objects *a* and *b* are of different structure types. Structures can be assigned only if the source and destination have the same type:

```

a = a1; // OK: same type, so member by member assignment
a = b;  // ILLLEGAL: different types
a.i = b.i; a.j = b.j; a.d = b.d /* but you can assign
                                member-by-member */

```

## Structure word alignment

Memory is allocated to a structure member-by-member from left to right, from low to high memory address. In this example,

```

struct mystruct {
    int i;
    char str[21];
    double d;
} s;

```

the object *s* occupies sufficient memory to hold a 2-byte integer, a 21-byte string, and an 8-byte double. The format of this object in memory is determined by the Turbo C++ word alignment option. With this option off (the default), *s* will be allocated 31 contiguous bytes. If you turn on word alignment with the **-a** compiler option (or with the **Options | Compiler | Code Generation** dialog box), Turbo C++ pads the structure with bytes to ensure the structure is aligned as follows:

1. The structure will start on a word boundary (even address).
2. Any non-**char** member will have an even byte offset from the start of the structure.
3. A final byte is added (if necessary) at the end to ensure that the whole structure contains an even number of bytes.

With word alignment on, the structure would therefore have a byte added before the **double**, making a 32-byte object.

---

## Structure name spaces

Structure tag names share the same name space with union tags and enumeration tags (but **enums** within a structure are in a different name space in C++). This means that such tags must be uniquely named within the same scope. However, tag names need not differ from identifiers in the other three name spaces: the label name space, the member name space(s), and the single name space (which consists of variables, functions, **typedef** names, and enumerators).

Member names within a given structure or union must be unique, but they can share the names of members in other structures or unions. For example,

```
goto s;
...
s:
struct s { // OK: tag and label name spaces different
    int s; // OK: label, tag and member name spaces different
    float s; // ILLEGAL: member name duplicated
} s; // OK: var name space different. In C++, this can only be
// done if s does not have a constructor.

union s { // ILLEGAL: tag space duplicate
    int s; // OK: new member space
    float f;
} f; // OK: var name space

struct t {
    int s; // OK: different member space
    ...
} s; // ILLEGAL: var name duplicate
```

---

## Incomplete declarations

A pointer to a structure type *A* can legally appear in the declaration of another structure *B* before the structure *A* has been declared:



would leave binary  $10 = -2$  in *a.i* with no warning. The signed **int** field *k* of width 1 can hold only the values  $-1$  and  $0$ , since the bit pattern 1 is interpreted as  $-1$ .

**Note** Bit fields can be declared only in structures, unions, and classes. They are accessed with the same member selectors (`.` and `->`) used for non-bit field members. Also, bit fields pose several problems when writing portable code, since the organization of bits-within-bytes and bytes-within-words is machine dependent.

The expression `&mystruct.x` is illegal if *x* is a bit field identifier, since there is no guarantee that *mystruct.x* lies at a byte address.

## Unions

---

*Unions correspond to the variant record types of Pascal and Modula-2.*

Union types are derived types sharing many of the syntactical and functional features of structure types. The key difference is that a union allows only one of its members to be “active” at any one time. The size of a union is the size of its largest member. The value of only one of its members can be stored at any time. In the following simple case,

```
union myunion {      /* union tag = myunion */
    int i;
    double d;
    char ch;
} mu, *muptr=&mu;
```

the identifier *mu*, of type **union myunion**, can be used to hold a 2-byte **int**, an 8-byte **double**, or a single-byte **char**, but only one of these at the same time.

`sizeof(union myunion)` and `sizeof(mu)` both return 8, but 6 bytes are unused (padded) when *mu* holds an **int** object, and 7 bytes are unused when *mu* holds a **char**. You access union members with the structure member selectors (`.` and `->`), but care is needed:

```
mu.d = 4.016;
printf("mu.d = %f\n",mu.d); // OK: displays mu.d = 4.016
printf("mu.i = %d\n",mu.i); // peculiar result
mu.ch = 'A';
printf("mu.ch = %c\n",mu.ch); // OK: displays mu.ch = A
printf("mu.d = %f\n",mu.d); // peculiar result
muptr->i = 3;
printf("mu.i = %d\n",mu.i); // OK: displays mu.i = 3
```

The second **printf** is legal, since *mu.i* is an integer type. However, the bit pattern in *mu.i* corresponds to parts of the **double** previously assigned, and will not usually provide a useful integer interpretation.

When properly converted, a pointer to a union points to each of its members, and vice versa.

## Union declarations

---

The general declaration syntax for unions is pretty much the same as that for structures. Differences are

1. Unions can contain bit fields, but only one can be active. They all start at the beginning of the union.
-  2. Unlike C++ structures, C++ union types cannot use the class access specifiers: **public**, **private**, and **protected**. All fields of a union are public.
3. Unions can be initialized only through their first declared member:

```
union local87 {
    int i;
    double d;
} a = { 20 };
```
-  4. A union can't participate in a class hierarchy. It can't be derived from any class, nor can it be a base class. A union *can* have a constructor.
-  5. Anonymous unions can't have member functions.

## Enumerations

---

An enumeration data type is used to provide mnemonic identifiers for a set of integer values. For example, the following declaration,

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
```

establishes a unique integral type, **enum days**, a variable *anyday* of this type, and a set of enumerators (*sun, mon,...*) with constant integer values.

Turbo C++ is free to store enumerators in a single byte when the **-b** flag is off (default is on, meaning **enums** are always **ints**) if the range of values permits, but the value is always promoted to an **int** when used in expressions. The identifiers used in an enumerator list are implicitly of type **unsigned char** or **int**, depending on the values of the

enumerators. If all values can be represented in an **unsigned char**, that is the type of each enumerator.



In C, a variable of an enumerated type can be assigned any value of type **int**—no type checking beyond that is enforced. In C++, a variable of an enumerated type can be assigned only one of its enumerators. That is,

```
anyday = mon;      // OK
anyday = 1;        // illegal, even though mon == 1
```

The identifier *days* is the optional enumeration tag that can be used in subsequent declarations of enumeration variables of type **enum days**:

```
enum days payday, holiday; // declare two variables
```



In C++, you can omit the **enum** keyword if *days* is not the name of anything else in the same scope.

As with **struct** and **union** declarations, you can omit the tag if no further variables of this **enum** type are required:

```
enum { sun, mon, tues, wed, thur, fri, sat } anyday;
/* anonymous enum type */
```

See page 17 for more on enumeration constants.

The enumerators listed inside the braces are also known as *enumeration constants*. Each is assigned a fixed integral value. In the absence of explicit initializers, the first enumerator (*sun*) is set to zero, and each succeeding enumerator is set to one more than its predecessor (*mon* = 1, *tues* = 2, and so on).

With explicit integral initializers, you can set one or more enumerators to specific values. Any subsequent names without initializers will then increase by one. For example, in the following declaration,

```
/* initializer expression can include previously declared
   enumerators */
enum coins { penny = 1, tuppence, nickel = penny + 4, dime = 10,
            quarter = nickel * nickel } smallchange;
```

*tuppence* would acquire the value 2, *nickel* the value 5, and *quarter* the value 25.

The initializer can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

**enum** types can appear wherever **int** types are permitted.

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
enum days payday;
```

```

typedef enum days DAYS;
DAYS *daysptr;
int i = tues;
anyday = mon;           // OK
*daysptr = anyday;    // OK
mon = tues;             // ILLEGAL: mon is a constant

```

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers:

```

int mon = 11;
{
    enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
    /* enumerator mon hides outer declaration of int mon */
    struct days { int i, j;}; // ILLEGAL: days duplicate tag
    double sat;              // ILLEGAL: redefinition of sat
}
mon = 12;                    // back in int mon scope

```



In C++, enumerators declared within a class are in the scope of that class.

## Expressions

---

*Table 1.19 shows how identifiers and operators are combined to form grammatically legal "phrases."*

An *expression* is a sequence of operators, operands, and punctuators that specifies a computation. The formal syntax, listed in Table 1.19, indicates that expressions are defined recursively: Subexpressions can be nested without formal limit. (However, the compiler will report an out-of-memory error if it can't compile an expression that is too complex.)

Table 1.19: Turbo C++ expressions

---

<p><b>primary-expression:</b>  <i>literal</i>  <i>pseudo-variable</i>  <i>(expression)</i>  <b>this</b> (C++ specific)  <b>:: identifier</b> (C++ specific)  <b>:: operator-function-name</b> (C++ specific)  <i>name</i></p> <p><b>literal:</b>  <i>integer-constant</i>  <i>character-constant</i>  <i>floating-constant</i>  <i>string</i></p> <p><b>name:</b>  <i>identifier</i>  <i>operator-function-name</i> (C++ specific)  <i>conversion-function-name</i> (C++ specific)  <i>qualified-name</i> (C++ specific)</p> <p><b>qualified-name:</b> (C++ specific)  <i>class-name</i> :: <i>identifier</i>  <i>class-name</i> :: <i>operator-function-name</i>  <i>class-name</i> :: <i>conversion-function-name</i>  <i>class-name</i> :: <i>class-name</i>  <i>class-name</i> :: ~ <i>class-name</i></p> <p><b>postfix-expression:</b>  <i>primary-expression</i>  <i>postfix-expression</i> [ <i>expression</i> ]  <i>postfix-expression</i> ( &lt;<i>expression-list</i>&gt; )  <i>simple-type-name</i> ( &lt;<i>expression-list</i>&gt; ) (C++ specific)  <i>postfix-expression</i> . <i>name</i>  <i>postfix-expression</i> -&gt; <i>name</i>  <i>postfix-expression</i> ++  <i>postfix-expression</i> --</p> <p><b>expression-list:</b>  <i>assignment-expression</i>  <i>expression-list</i> , <i>assignment-expression</i></p> <p><b>unary-expression:</b>  <i>postfix-expression</i>  ++ <i>unary-expression</i>  -- <i>unary-expression</i>  <i>unary-operator</i> <i>cast-expression</i>  <b>sizeof</b> <i>unary-expression</i>  <b>sizeof</b> ( <i>type-name</i> )  <i>allocation-expression</i> (C++ specific)  <i>deallocation-expression</i> (C++ specific)</p> <p><b>unary-operator:</b> one of  &amp; * + - ~ !</p> <p><b>allocation-expression:</b> (C++ specific)  &lt;&lt;::&gt; <b>new</b> &lt;<i>placement</i>&gt; <i>restricted-type-name</i> &lt;<i>initializer</i>&gt;  &lt;&lt;::&gt; <b>new</b> &lt;<i>placement</i>&gt; ( <i>type-name</i> ) &lt;<i>initializer</i>&gt;</p> <p><b>placement:</b> (C++ specific)  ( <i>expression-list</i> )</p> <p><b>restricted-type-name:</b> (C++ specific)  <i>type-specifiers</i> &lt;<i>restricted-declarator</i>&gt;</p> <p><b>restricted-declarator:</b> (C++ specific)  <i>ptr-operator</i> &lt;<i>restricted-declarator</i>&gt;  <i>restricted-declarator</i> [ &lt;<i>expression</i>&gt; ]</p> <p><b>deallocation-expression:</b> (C++ specific)  &lt;&lt;::&gt; <b>delete</b> <i>cast-expression</i>  &lt;&lt;::&gt; <b>delete</b> [ <i>expression</i> ] <i>cast-expression</i></p>	<p><b>cast-expression:</b>  <i>unary-expression</i>  ( <i>type-name</i> ) <i>cast-expression</i></p> <p><b>pm-expression:</b>  <i>cast-expression</i>  <i>pm-expression</i> . * <i>cast-expression</i> (C++ specific)  <i>pm-expression</i> -&gt; * <i>cast-expression</i> (C++ specific)</p> <p><b>multiplicative-expression:</b>  <i>pm-expression</i>  <i>multiplicative-expression</i> * <i>pm-expression</i>  <i>multiplicative-expression</i> / <i>pm-expression</i>  <i>multiplicative-expression</i> % <i>pm-expression</i></p> <p><b>additive-expression:</b>  <i>multiplicative-expression</i>  <i>additive-expression</i> + <i>multiplicative-expression</i>  <i>additive-expression</i> - <i>multiplicative-expression</i></p> <p><b>shift-expression:</b>  <i>additive-expression</i>  <i>shift-expression</i> &lt;&lt; <i>additive-expression</i>  <i>shift-expression</i> &gt;&gt; <i>additive-expression</i></p> <p><b>relational-expression:</b>  <i>shift-expression</i>  <i>relational-expression</i> &lt; <i>shift-expression</i>  <i>relational-expression</i> &gt; <i>shift-expression</i>  <i>relational-expression</i> &lt;= <i>shift-expression</i>  <i>relational-expression</i> &gt;= <i>shift-expression</i></p> <p><b>equality-expression:</b>  <i>relational-expression</i>  <i>equality-expression</i> == <i>relational-expression</i>  <i>equality-expression</i> != <i>relational-expression</i></p> <p><b>AND-expression:</b>  <i>equality-expression</i>  <i>AND-expression</i> &amp; <i>equality-expression</i></p> <p><b>exclusive-OR-expression:</b>  <i>AND-expression</i>  <i>exclusive-OR-expression</i> ^ <i>AND-expression</i></p> <p><b>inclusive-OR-expression:</b>  <i>exclusive-OR-expression</i>  <i>inclusive-OR-expression</i>   <i>exclusive-OR-expression</i></p> <p><b>logical-AND-expression:</b>  <i>inclusive-OR-expression</i>  <i>logical-AND-expression</i> &amp;&amp; <i>inclusive-OR-expression</i></p> <p><b>logical-OR-expression:</b>  <i>logical-AND-expression</i>  <i>logical-OR-expression</i>    <i>logical-AND-expression</i></p> <p><b>conditional-expression:</b>  <i>logical-OR-expression</i>  <i>logical-OR-expression</i> ? <i>expression</i> : <i>conditional-expression</i></p> <p><b>assignment-expression:</b>  <i>conditional-expression</i>  <i>unary-expression</i> <i>assignment-operator</i> <i>assignment-expression</i></p> <p><b>assignment-operator:</b> one of  = * = / = % = + = - =  &lt;&lt;= &gt;&gt;= &amp;= ^=  =</p> <p><b>expression:</b>  <i>assignment-expression</i>  <i>expression</i> , <i>assignment-expression</i></p> <p><b>constant-expression:</b>  <i>conditional-expression</i></p>
---	---

---

The standard conversions are detailed in Table 1.15 on page 42.

Expressions are evaluated according to certain conversion, grouping, associativity, and precedence rules which depend on the operators used, the presence of parentheses, and the data types of the operands. The way operands and subexpressions are grouped does not necessarily specify the actual order in which they are evaluated by Turbo C++ (see "Evaluation order" on page 76). Expressions can produce an lvalue, an rvalue, or no value. Expressions may cause side effects whether they produce a value or not.

The grammar in Table 1.19 on page 74 completely defines the precedence and associativity of the operators.

We've summarized the precedence and associativity rules in Table 1.20. Each category has an associativity rule: left to right, or right to left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

Table 1.20  
Associativity and precedence of Turbo C++ operators  
There are sixteen precedence categories. The first category (the first line) has the highest precedence. Operators in the same category have equal precedence. Where there are duplicates of operators in the table, the first occurrence is unary, the second binary.

Operators	Associativity
( ) [ ] -> :: .	Left to right
! ~ + - ++ -- & * (typecast) sizeof new delete	Right to left
.* ->*	Left to right
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
?: (conditional expression)	Right to left
= *= /= %= += -= &= ^=  = <<= >>=	Right to left
,	Left to right

## Expressions and C++

C++ allows the overloading of certain standard C operators, as explained starting on page 125. An overloaded operator is defined to behave in a special way when applied to expressions of class type. For instance, the relational operator == might be defined in the class **complex** to test the equality of two complex numbers without changing its normal usage with non-class data types. Overloaded operators are implemented as functions; the function determines the operand type, lvalue, and evaluation order to be applied when the overloaded operator is used. However,

overloading cannot change the precedence of an operator. Similarly, C++ allows user-defined conversions between class objects and fundamental types. Keep in mind, then, that some of the rules for operators and conversions discussed in this section may not apply to expressions in C++.

## Evaluation order

---

The order in which Turbo C++ evaluates the operands of an expression is not specified, except where an operator specifically states otherwise. The compiler will try to rearrange the expression in order to improve the quality of the generated code. Care is therefore needed with expressions in which a value is modified more than once. In general, avoid writing expressions that both modify and use the value of the same object. Consider the expression

```
i = v[i++]; // i is undefined
```

The value of *i* depends on whether *i* is incremented before or after the assignment. Similarly,

```
int total = 0;
sum = (total = 3) + (++total); // sum = 4 or sum = 7 ??
```

is ambiguous for *sum* and *total*. The solution is to revamp the expression, using a temporary variable:

```
int temp, total = 0;
temp = ++total;
sum = (total = 3) + temp;
```

Where the syntax does enforce an evaluation sequence, it is safe to have multiple evaluations:

```
sum = (i = 3, i++, i++); // OK: sum = 4, i = 5
```

Each subexpression of the comma expression is evaluated from left to right, and the whole expression evaluates to the rightmost value.

Turbo C++ regroups expressions, rearranging associative and commutative operators regardless of parentheses, in order to create an efficiently compiled expression; in no case will the rearrangement affect the value of the expression.

You can use parentheses to force the order of evaluation in expressions. For example, if you have the variables *a*, *b*, *c*, and *f*, then the expression  $f = a + (b + c)$  forces  $(b + c)$  to be evaluated before adding the result to *a*.

---

## Errors and overflows

During the evaluation of an expression, Turbo C++ can encounter many problematic situations, such as division by zero or out-of-range floating-point values. Integer overflow is ignored (C uses modulo  $2^n$  arithmetic on  $n$ -bit registers), but errors detected by math library functions can be handled by standard or user-defined routines. See **matherr** and **signal** in the *Library Reference*.

---

## Operator semantics

*The Turbo C++ operators described here are the standard ANSI C operators.*

Unless the operators are overloaded, the following information is true in both C and C++. In C++ you can overload all of these operators with the exception of `.` (member operator) and `?:` (conditional operator) (and you also can't overload the C++ operators `::` and `.*`).

If an operator is overloaded, the discussion may not be true for it anymore. Table 1.19 on page 74 gives the syntax for all operators and operator expressions.

---

## Postfix and prefix operators

The six postfix operators `[]` `()` `.` `->` `++` and `--` are used to build postfix expressions as shown in the expressions syntax table (Table 1.19). The increment and decrement operators (`++` and `--`) are also prefix and unary operators; they are discussed starting on page 79.

### Array subscript operator `[]`

In the expression

*postfix-expression* [*expression*]

In C, but not necessarily in C++, the expression *exp1*[*exp2*] is defined as

\* ((*exp1*) + (*exp2*))

where either *exp1* is a pointer and *exp2* is an integer, or *exp1* is an integer and *exp2* is a pointer. (The punctuators `[]`, `*`, and `+` can be individually overloaded in C++.)

Function call operators ( )      The expression  
*postfix-expression(<arg-expression-list>)*

is a call to the function given by the postfix expression. The *arg-expression-list* is a comma-delimited list of expressions of any type representing the actual (or real) function arguments. The value of the function call expression, if any, is determined by the return statement in the function definition. See "Function calls and argument conversions," page 63, for more on function calls.

Structure/union member operator . (dot)      In the expression  
*postfix-expression . identifier*

*lvalues are defined on page 28.*

the postfix expression must be of type structure or union; the identifier must be the name of a member of that structure or union type. The expression designates a member of a structure or union object. The value of the expression is the value of the selected member; it will be an lvalue if and only if the postfix expression is an lvalue. Detailed examples of the use of . and → for structures are given on page 66.

Structure/union pointer operator →      In the expression  
*postfix-expression → identifier*

the postfix expression must be of type pointer to structure or pointer to union; the identifier must be the name of a member of that structure or union type. The expression designates a member of a structure or union object. The value of the expression is the value of the selected member; it will be an lvalue if and only if the postfix expression is an lvalue.

Postfix increment operator ++      In the expression  
*postfix-expression ++*

the postfix expression is the operand; it must be of scalar type (arithmetic or pointer types) and must be a modifiable lvalue (see page 28 for more on modifiable lvalues). The postfix ++ is also known as the *postincrement* operator. The value of the whole expression is the value of the postfix expression *before* the increment is applied. After the postfix expression is evaluated, the operand is incremented by 1.

The increment value is appropriate to the type of the operand. Pointer types are subject to the rules for pointer arithmetic.

Postfix decrement operator -- The postfix decrement, also known as the *postdecrement*, operator follows the same rules as the postfix increment, except that 1 is subtracted from the operand *after* the evaluation.

---

## Increment and decrement operators

The first two unary operators are ++ and --. These are also postfix and prefix operators, so they are discussed here. The remaining six unary operators are covered following this discussion.

Prefix increment operator

In the expression

*++ unary-expression*

the unary expression is the operand; it must be of scalar type and must be a modifiable lvalue. The prefix increment operator is also known as the *preincrement* operator. The operand is incremented by 1 *before* the expression is evaluated; the value of the whole expression is the incremented value of the operand. The 1 used to increment is the appropriate value for the type of the operand. Pointer types follow the rules of pointer arithmetic.

Prefix decrement operator

The prefix decrement, also known as the *predecrement*, operator has the following syntax:

*-- unary-expression*

It follows the same rules as the prefix increment operator, except that the operand is decremented by 1 before the whole expression is evaluated.

---

## Unary operators

The six unary operators (aside from ++ and --) are & \* + - ~ and !. The syntax is

*unary-operator cast-expression*

*cast-expression:*

*unary-expression*

*(type-name) cast-expression*

Address operator &  
The symbol & is also used in  
C++ to specify reference  
types; see page 98.

The & operator and \* operator (the \* operator is described in the next section) work together as the *referencing* and *dereferencing* operators. In the expression

**&** *cast-expression*

the *cast-expression* operand must be either a function designator or an lvalue designating an object that is not a bit field and is not declared with the **register** storage class specifier. If the operand is of type **type**, the result is of type pointer to **type**.

Note that some non-lvalue identifiers, such as function names and array names, are automatically converted into “pointer to X” types when appearing in certain contexts. The & operator can be used with such objects, but its use is redundant and therefore discouraged.

Consider the following extract:

```
type t1 = 1, t2 = 2;  
type *ptr = &t1;    // initialized pointer  
*ptr = t2;         // same effect as t1 = t2
```

Note that **type** \*ptr = &t1 is treated as

```
T *ptr;  
ptr = &t1;
```

so it is *ptr*, not *\*ptr*, that gets assigned. Once *ptr* has been initialized with the address &t1, it can be safely dereferenced to give the lvalue *\*ptr*.

Indirection operator \* In the expression

**\*** *cast-expression*

the *cast-expression* operand must have type “pointer to **type**,” where **type** is any type. The result of the indirection is of type **type**. If the operand is of type “pointer to function,” the result is a function designator; if the operand is a pointer to an object, the result is an lvalue designating that object. In the following situations, the result of indirection is undefined:

1. The *cast-expression* is a null pointer.
2. The *cast-expression* is the address of an automatic variable and execution of its block has terminated.

Unary plus operator + In the expression  
+ *cast-expression*  
the *cast-expression* operand must be of arithmetic type. The result is the value of the operand after any required integral promotions.

Unary minus operator - In the expression  
- *cast-expression*  
the *cast-expression* operand must be of arithmetic type. The result is the negative of the value of the operand after any required integral promotions.

Bitwise complement operator ~ In the expression  
~ *cast-expression*  
the *cast-expression* operand must be of integral type. The result is the bitwise complement of the operand after any required integral promotions. Each 0 bit in the operand is set to 1, and each 1 bit in the operand is set to 0.

Logical negation operator ! In the expression  
! *cast-expression*  
the *cast-expression* operand must be of scalar type. The result is of type **int** and is the logical negation of the operand: 0 if the operand is non-zero; 1 if the operand is zero. The expression *!E* is equivalent to (0 == E).

---

## The sizeof operator

There are two distinct uses of the **sizeof** operator:

**sizeof** *unary-expression*  
**sizeof** (*type-name*)

*How much space is set aside for each type depends on the machine.*

The result in both cases is an integer constant that gives the size in bytes of how much memory space is used by the operand (determined by its type, with some exceptions). In the first use, the type of the operand expression is determined without evaluating the expression (and therefore without side effects). When the operand is of type **char** (**signed** or **unsigned**), **sizeof** gives the result 1. When the operand is a

non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is *not* converted to a pointer type). The number of elements in an array equals **sizeof array/sizeof array[0]**.

If the operand is a parameter declared as array type or function type, **sizeof** gives the size of the pointer. When applied to structures and unions, **sizeof** gives the total number of bytes, including any padding.

**sizeof** cannot be used with expressions of function type, incomplete types, parenthesized names of such types, or with an lvalue that designates a bit field object.

The integer type of the result of **sizeof** is **size\_t**, defined as **unsigned int** in `stddef.h`.

You can use **sizeof** in preprocessor directives; this is specific to Turbo C++.



In C++, **sizeof(class type)**, where *class type* is derived from some base class, returns the base class size.

## Multiplicative operators

There are three multiplicative operators: **\***, **/** and **%**. The syntax is

*multiplicative-expression:*

*cast-expression*

*multiplicative-expression \* cast-expression*

*multiplicative-expression / cast-expression*

*multiplicative-expression % cast-expression*

The operands for **\*** (multiplication) and **/** (division) must be of arithmetical type. The operands for **%** (modulus, or remainder) must be of integral type. The usual arithmetic conversions are made on the operands (see page 41).

The result of (*op1 \* op2*) is the product of the two operands. The results of (*op1 / op2*) and (*op1 % op2*) are the quotient and remainder, respectively, when *op1* is divided by *op2*, provided that *op2* is nonzero. Use of **/** or **%** with a zero second operand results in an error.

When *op1* and *op2* are integers and the quotient is not an integer, the results are as follows:

*Rounding is always toward zero.*

1. If *op1* and *op2* have the same sign, *op1 / op2* is the largest integer less than the true quotient, and *op1 % op2* has the sign of *op1*.
2. If *op1* and *op2* have opposite signs, *op1 / op2* is the smallest integer greater than the true quotient, and *op1 % op2* has the sign of *op1*.

## Additive operators

---

There are two additive operators: `+` and `-`. The syntax is

*additive-expression:*  
*multiplicative-expression*  
*additive-expression + multiplicative-expression*  
*additive-expression - multiplicative-expression*

### The addition operator `+`

The legal operand types for *op1* `+` *op2* are

1. Both *op1* and *op2* are of arithmetic type.
2. *op1* is of integral type, and *op2* is of pointer to object type.
3. *op2* is of integral type, and *op1* is of pointer to object type.

In case 1, the operands are subjected to the standard arithmetical conversions, and the result is the arithmetical sum of the operands. In cases 2 and 3, the rules of pointer arithmetic apply. (Pointer arithmetic is covered on page 57.)

### The subtraction operator `-`

The legal operand types for *op1* `-` *op2* are

1. Both *op1* and *op2* are of arithmetic type.
2. Both *op1* and *op2* are pointers to compatible object types. (**Note:** The unqualified type **type** is considered to be compatible with the qualified types **const type**, **volatile type**, and **const volatile type**.)
3. *op1* is of pointer to object type, and *op2* is integral type.

In case 1, the operands are subjected to the standard arithmetic conversions, and the result is the arithmetic difference of the operands. In cases 2 and 3, the rules of pointer arithmetic apply.

## Bitwise shift operators

---

There are two bitwise shift operators: `<<` and `>>`. The syntax is

*shift-expression:*  
*additive-expression*  
*shift-expression << additive-expression*  
*shift-expression >> additive-expression*

Bitwise left-shift operator <<

In the expression  $E1 \ll E2$ , the operands  $E1$  and  $E2$  must be of integral type. The normal integral promotions are performed on  $E1$  and  $E2$ , and the type of the result is the type of the promoted  $E1$ . If  $E2$  is negative or is greater than or equal to the width in bits of  $E1$ , the operation is undefined.

*The constants `ULONG_MAX` and `UINT_MAX` are defined in `limits.h`.*

The result of  $E1 \ll E2$  is the value of  $E1$  left-shifted by  $E2$  bit positions, zero-filled from the right if necessary. Left shifts of an **unsigned long**  $E1$  are equivalent to multiplying  $E1$  by  $2^{E2}$ , reduced modulo  $ULONG\_MAX + 1$ ; left shifts of **unsigned ints** are equivalent to multiplying by  $2^{E2}$  reduced modulo  $UINT\_MAX + 1$ . If  $E1$  is a signed integer, the result must be interpreted with care, since the sign bit may change.

Bitwise right-shift operator >>

In the expression  $E1 \gg E2$ , the operands  $E1$  and  $E2$  must be of integral type. The normal integral promotions are performed on  $E1$  and  $E2$ , and the type of the result is the type of the promoted  $E1$ . If  $E2$  is negative or is greater than or equal to the width in bits of  $E1$ , the operation is undefined.

The result of  $E1 \gg E2$  is the value of  $E1$  right-shifted by  $E2$  bit positions. If  $E1$  is of **unsigned** type, zero-fill occurs from the left if necessary. If  $E1$  is of **signed** type, the fill from the left uses the sign bit (0 for positive, 1 for negative  $E1$ ). This sign-bit extension ensures that the sign of  $E1 \gg E2$  is the same as the sign of  $E1$ . Except for signed types, the value of  $E1 \gg E2$  is the integral part of the quotient  $E1 / 2^{E2}$ .

---

## Relational operators

There are four relational operators: `<` `>` `<=` and `>=`. The syntax for these operators is:

*relational-expression:*

*shift-expression*

*relational-expression < shift-expression*

*relational-expression > shift-expression*

*relational-expression <= shift-expression*

*relational-expression >= shift-expression*

The less-than operator < In the expression  $E1 < E2$ , the operands must conform to one of the following sets of conditions:

*Qualified names are defined on page 108.*

1. Both  $E1$  and  $E2$  are of arithmetic type.
2. Both  $E1$  and  $E2$  are pointers to qualified or unqualified versions of compatible object types.
3. Both  $E1$  and  $E2$  are pointers to qualified or unqualified versions of compatible incomplete types.

In case 1, the usual arithmetic conversions are performed. The result of  $E1 < E2$  is of type **int**. If the value of  $E1$  is less than the value of  $E2$ , the result is 1 (true); otherwise, the result is zero (false).

In cases 2 and 3, where  $E1$  and  $E2$  are pointers to compatible types, the result of  $E1 < E2$  depends on the relative locations (addresses) of the two objects being pointed at. When comparing structure members within the same structure, the “higher” pointer indicates a later declaration. Within arrays, the “higher” pointer indicates a larger subscript value. All pointers to members of the same union object compare as equal.

Normally, the comparison of pointers to different structure, array, or union objects, or the comparison of pointers outside the range of an array object give undefined results; however, an exception is made for the “pointer beyond the last element” situation as discussed under “Pointer arithmetic” on page 57. If  $P$  points to an element of an array object, and  $Q$  points to the last element, the expression  $P < Q + 1$  is allowed, evaluating to 1 (true), even though  $Q + 1$  does not point to an element of the array object.

The greater-than operator > The expression  $E1 > E2$  gives 1 (true) if the value of  $E1$  is greater than the value of  $E2$ ; otherwise, the result is 0 (false), using the same interpretations for arithmetic and pointer comparisons, as defined for the less-than operator. The same operand rules and restrictions also apply.

The less-than or equal-to operator <= Similarly, the expression  $E1 <= E2$  gives 1 (true) if the value of  $E1$  is less than or equal to the value of  $E2$ . Otherwise, the result is 0 (false), using the same interpretations for arithmetic and pointer comparisons, as defined for the less-than operator. The same operand rules and restrictions also apply.

The greater-than or equal-to operator >=

Finally, the expression  $E1 \geq E2$  gives 1 (true) if the value of  $E1$  is greater than or equal to the value of  $E2$ . Otherwise, the result is 0 (false), using the same interpretations for arithmetic and pointer comparisons, as defined for the less-than operator. The same operand rules and restrictions also apply.

## Equality operators

---

There are two equality operators: `==` and `!=`. They test for equality and inequality between arithmetic values or between pointer values, following rules very similar to those for the relational operators. Note, however, that `==` and `!=` have a lower precedence than the relational operators `<`, `>`, `<=`, and `>=`. Also, `==` and `!=` can compare certain pointer types for equality and inequality where the relational operators would not be allowed.

The syntax is

```
equality-expression:
    relational-expression
equality-expression == relational-expression
equality-expression != relational-expression
```

The equal-to operator ==

In the expression  $E1 == E2$ , the operands must conform to one of the following sets of conditions:

1. Both  $E1$  and  $E2$  are of arithmetic type.
2. Both  $E1$  and  $E2$  are pointers to qualified or unqualified versions of compatible types.
3. One of  $E1$  and  $E2$  is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of **void**.
4. One of  $E1$  or  $E2$  is a pointer and the other is a null pointer constant.

If  $E1$  and  $E2$  have types that are valid operand types for a relational operator, the same comparison rules just detailed for  $E1 < E2$ ,  $E1 <= E2$ , and so on, apply.

In case 1, for example, the usual arithmetic conversions are performed, and the result of  $E1 == E2$  is of type **int**. If the value of  $E1$  is equal to the value of  $E2$ , the result is 1 (true); otherwise, the result is zero (false).

In case 2,  $E1 == E2$  gives 1 (true) if  $E1$  and  $E2$  point to the same object, or both point “one past the last element” of the same array object, or both are null pointers.

If  $E1$  and  $E2$  are pointers to function types,  $E1 == E2$  gives 1 (true) if they are both null or if they both point to the same function. Conversely, if  $E1 == E2$  gives 1 (true), then either  $E1$  and  $E2$  point to the same function, or they are both null.

In case 4, the pointer to an object or incomplete type is converted to the type of the other operand (pointer to a qualified or unqualified version of **void**).

The inequality operator  $!=$  The expression  $E1 != E2$  follows the same rules as those for  $E1 == E2$ , except that the result is 1 (true) if the operands are unequal, and 0 (false) if the operands are equal.

## Bitwise AND operator &

The syntax is

*AND-expression:*  
*equality-expression*  
*AND-expression & equality-expression*

In the expression  $E1 \& E2$ , both operands must be of integral type. The usual arithmetical conversions are performed on  $E1$  and  $E2$ , and the result is the bitwise AND of  $E1$  and  $E2$ . Each bit in the result is determined as shown in Table 1.21.

Table 1.21  
Bitwise operators truth table

Bit value in E1	Bit value in E2	$E1 \& E2$	$E1 \wedge E2$	$E1   E2$
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

## Bitwise exclusive OR operator ^

The syntax is

*exclusive-OR-expression:*  
*AND-expression*  
*exclusive-OR-expression ^ AND-expression*

In the expression  $E1 \wedge E2$ , both operands must be of integral type. The usual arithmetic conversions are performed on  $E1$  and  $E2$ , and the result is the bitwise exclusive OR of  $E1$  and  $E2$ . Each bit in the result is determined as shown in Table 1.21.

---

## Bitwise inclusive OR operator |

The syntax is

*inclusive-OR-expression:*  
*exclusive-OR-expression*  
*inclusive-OR-expression | exclusive-OR-expression*

In the expression  $E1 | E2$ , both operands must be of integral type. The usual arithmetic conversions are performed on  $E1$  and  $E2$ , and the result is the bitwise inclusive OR of  $E1$  and  $E2$ . Each bit in the result is determined as shown in Table 1.21.

---

## Logical AND operator &&

The syntax is

*logical-AND-expression:*  
*inclusive-OR-expression*  
*logical-AND-expression && inclusive-OR-expression*

In the expression  $E1 \&\& E2$ , both operands must be of scalar type. The result is of type **int**, the result is 1 (true) if the values of  $E1$  and  $E2$  are both nonzero; otherwise, the result is 0 (false).

Unlike the bitwise **&** operator, **&&** guarantees left-to-right evaluation.  $E1$  is evaluated first; if  $E1$  is zero,  $E1 \&\& E2$  gives 0 (false), and  $E2$  is not evaluated.

---

## Logical OR operator ||

The syntax is

*logical-OR-expression:*  
*logical-AND-expression*  
*logical-OR-expression || logical-AND-expression*

In the expression  $E1 || E2$ , both operands must be of scalar type. The result is of type **int**, and the result is 1 (true) if either of the values of  $E1$  and  $E2$  are nonzero. Otherwise, the result is 0 (false).

Unlike the bitwise `|` operator, `||` guarantees left-to-right evaluation. *E1* is evaluated first; if *E1* is nonzero, *E1* `||` *E2* gives 1 (true), and *E2* is not evaluated.

## Conditional operator ? :

The syntax is

```
conditional-expression  
logical-OR-expression  
logical-OR-expression ? expression : conditional-expression
```

In the expression *E1* ? *E2* : *E3*, the operand *E1* must be of scalar type. The operands *E2* and *E3* must obey one of the following sets of rules:

1. Both of arithmetic type
2. Both of compatible structure or union types
3. Both of void type
4. Both of type pointer to qualified or unqualified versions of compatible types
5. One operand of pointer type, the other a null pointer constant
6. One operand of type pointer to an object or incomplete type, the other of type pointer to a qualified or unqualified version of void

First, *E1* is evaluated; if its value is nonzero (true), then *E2* is evaluated and *E3* is ignored. If *E1* evaluates to zero (false), then *E3* is evaluated and *E2* is ignored. The result of *E1* ? *E2* : *E3* will be the value of whichever of *E2* and *E3* is evaluated.

In case 1, both *E2* and *E3* are subject to the usual arithmetic conversions, and the type of the result is the common type resulting from these conversions.

In case 2, the type of the result is the structure or union type of *E2* and *E3*.

In case 3, the result is of type **void**.

In cases 4 and 5, the type of the result is pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.

In case 6, the type of the result is that of the nonpointer-to-void operand.

## Assignment operators

There are eleven assignment operators. The = operator is the simple assignment operator; the other ten are known as compound assignment operators.

The syntax is

*assignment-expression:*  
*conditional-expression*  
*unary-expression assignment-operator assignment-expression*

*assignment-operator:* one of

= \*= /= %= += -=  
<<= >>= &= ^= |=

The simple assignment operator =

In the expression  $E1 = E2$ ,  $E1$  must be a modifiable lvalue. The value of  $E2$ , after conversion to the type of  $E1$ , is stored in the object designated by  $E1$  (replacing  $E1$ 's previous value). The value of the assignment expression is the value of  $E1$  after the assignment. The assignment expression is not itself an lvalue.

The operands  $E1$  and  $E2$  must obey one of the following sets of rules:

1.  $E1$  is of qualified or unqualified arithmetic type and  $E2$  is of arithmetic type.
2.  $E1$  has a qualified or unqualified version of a structure or union type compatible with the type of  $E2$ .
3.  $E1$  and  $E2$  are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right.
4. One of  $E1$  or  $E2$  is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**. The type pointed to by the left has all the qualifiers of the type pointed to by the right.
5.  $E1$  is a pointer and  $E2$  is a null pointer constant.

The compound assignment operators

The compound assignments  $op=$ , where  $op$  can be any one of the ten operator symbols \* / % + - << >> & ^ |, are interpreted as follows:

$E1\ op = E2$

has the same effect as

$$E1 = E1 \text{ op } E2$$

except that the lvalue  $E1$  is evaluated only once. For example,  $E1 += E2$  is the same as  $E1 = E1 + E2$ .

The rules for compound assignment are therefore covered in the previous section (on the simple assignment operator  $=$ ).

## Comma operator

---

The syntax is

*expression:*

*assignment-expression*

*expression , assignment-expression*

In the comma expression

$E1, E2$

the left operand  $E1$  is evaluated as a **void** expression, then  $E2$  is evaluated to give the result and type of the comma expression. By recursion, the expression

$E1, E2, \dots, En$

results in the left-to-right evaluation of each  $Ei$ , with the value and type of  $En$  giving the result of the whole expression. To avoid ambiguity with the commas used in function argument and initializer lists, parentheses must be used. For example,

```
func(i, (j = 1, j + 4), k);
```

calls **func** with three arguments, not four. The arguments are  $i$ , 5, and  $k$ .

## Statements

---

*Statements* specify the flow of control as a program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code. The following table lays out the syntax for statements:

Table 1.22: Turbo C++ statements

<p><i>statement:</i>  <i>labeled-statement</i>  <i>compound-statement</i>  <i>expression-statement</i>  <i>selection-statement</i>  <i>iteration-statement</i>  <i>jump-statement</i>  <i>asm-statement</i>  <i>declaration (C++ specific)</i></p> <p><i>asm-statement:</i>  <b>asm</b> <i>tokens</i> <i>newline</i>  <b>asm</b> <i>tokens</i>;  <b>asm</b> { <i>tokens</i>; &lt;<i>tokens</i>&gt;;=              &lt;<i>tokens</i>&gt;          }</p> <p><i>labeled-statement:</i>  <i>identifier</i> : <i>statement</i>  <b>case</b> <i>constant-expression</i> : <i>statement</i>  <b>default</b> : <i>statement</i></p> <p><i>compound-statement:</i>          { &lt;<i>declaration-list</i>&gt; &lt;<i>statement-list</i>&gt; }</p> <p><i>declaration-list:</i>  <i>declaration</i></p>	<p><i>declaration-list declaration</i></p> <p><i>statement-list:</i>  <i>statement</i>  <i>statement-list statement</i></p> <p><i>expression-statement:</i>          &lt;<i>expression</i>&gt; ;</p> <p><i>selection-statement:</i>  <b>if</b> ( <i>expression</i> ) <i>statement</i>  <b>if</b> ( <i>expression</i> ) <i>statement</i> <b>else</b> <i>statement</i>  <b>switch</b> ( <i>expression</i> ) <i>statement</i></p> <p><i>iteration-statement:</i>  <b>while</b> ( <i>expression</i> ) <i>statement</i>  <b>do</b> <i>statement</i> <b>while</b> ( <i>expression</i> );  <b>for</b> ( <i>for-init-statement</i> &lt;<i>expression</i>&gt; ; &lt;<i>expression</i>&gt; ) <i>statement</i></p> <p><i>for-init-statement</i>  <i>expression-statement</i>  <i>declaration (C++ specific)</i></p> <p><i>jump-statement:</i>  <b>goto</b> <i>identifier</i> ;  <b>continue</b> ;  <b>break</b> ;  <b>return</b> &lt;<i>expression</i>&gt; ;</p>
--	---

## Blocks

A compound statement, or *block*, is a list (possibly empty) of statements enclosed in matching braces ( { } ). Syntactically, a block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth.

## Labeled statements

A statement can be labeled in the following ways:

1. *label-identifier* : *statement*

The label identifier serves as a target for the unconditional **goto** statement. Label identifiers have their own name space and enjoy function scope. Note that in C++ you can label both declaration and non-declaration statements.

2. **case** *constant-expression* : *statement*  
**default** : *statement*

Case and default labeled statements are used only in conjunction with switch statements.

## Expression statements

---

Any expression followed by a semicolon forms an *expression statement*:

*<expression>;*

Turbo C++ executes an expression statement by evaluating the expression. All side effects from this evaluation are completed before the next statement is executed. Most expression statements are assignment statements or function calls.

A special case is the *null statement*, consisting of a single semicolon (;). The null statement does nothing. It is nevertheless useful in situations where the Turbo C++ syntax expects a statement but your program does not need one.

## Selection statements

---

Selection or flow-control statements select from alternative courses of action by testing certain values. There are two types of selection statements: the **if...else** and the **switch**.

### if statements

The basic **if** statement has the following pattern:

*The parentheses around cond-expression are essential.*

**if** (*cond-expression*) *t-st* **<else** *f-st***>**

The *cond-expression* must be of scalar type. The expression is evaluated. If the value is zero (or null for pointer types), we say that the *cond-expression* is false; otherwise, it is true.

If there is no **else** clause and *cond-expression* is true, *t-st* is executed; otherwise, *t-st* is ignored.

If the optional **else** *f-st* is present and *cond-expression* is true, *t-st* is executed; otherwise, *t-st* is ignored and *f-st* is executed.

### Note

Unlike, say, Pascal, Turbo C++ does not have a specific Boolean data type. Any expression of integer or pointer type can serve a Boolean role in conditional tests. The relational expression (*a > b*) (if legal) evaluates to **int** 1 (true) if (*a > b*), and to **int** 0 (false) if (*a <= b*). Pointer conversions are such that a pointer can always be correctly compared to a constant expression evaluating to 0. That is, the test for null pointers can be written **if** (!ptr) ... or **if** (ptr == 0) ...

The *f-st* and *t-st* statements can themselves be **if** statements, allowing for a series of conditional tests nested to any depth. Care is needed with nested **if...else** constructs to ensure that the correct statements

are selected. There is no **endif** statement: Any "else" ambiguity is resolved by matching an **else** with the last encountered **if**-without-an-**else** at the same block level. For example,

```
if (x == 1)
    if (y == 1) puts("x=1 and y=1");
    else puts("x != 1");
```

draws the wrong conclusion! The **else** matches with the second **if**, despite the indentation. The correct conclusion is that  $x = 1$  and  $y \neq 1$ . Note the effect of braces:

```
if (x == 1)
{
    if (y == 1) puts("x = 1 and y = 1");
}
else puts("x != 1"); // correct conclusion
```

switch statements The **switch** statement uses the following basic format:

**switch** (*sw-expression*) *case-st*

A **switch** statement lets you transfer control to one of several case-labeled statements, depending on the value of *sw-expression*. The latter must be of integral type (in C++, it can be of class type, provided that there is an unambiguous conversion to integral type available). Any statement in *case-st* (including empty statements) can be labeled with one or more case labels:

*It is illegal to have duplicate case constants in the same switch statement.*

**case** *const-exp-i* : *case-st-i*

where each case constant, *const-exp-i*, is a constant expression with a unique integer value (converted to the type of the controlling expression) within its enclosing **switch** statement.

There can also be at most one **default** label:

**default** : *default-st*

After evaluating *sw-expression*, a match is sought with one of the *const-exp-i*. If a match is found, control passes to the statement *case-st-i* with the matching case label.

If no match is found and there is a **default** label, control passes to *default-st*. If no match is found and there is no **default** label, none of the statements in *case-st* is executed. Program execution is not affected when **case** and **default** labels are encountered. Control simply passes through the labels to the following statement or switch. To stop

execution at the end of a group of statements for a particular case, use **break**.

---

## Iteration statements

Iteration statements let you loop a set of statements. There are three forms of iteration in Turbo C++: **while**, **do**, and **for** loops.

### while statements

The general format for this statement is

*The parentheses are essential.*

```
while (cond-exp) t-st
```

The loop statement, *t-st*, will be executed repeatedly until the conditional expression, *cond-exp*, compares equal to zero (false).

The *cond-exp* is evaluated and tested first (as described on page 93). If this value is nonzero (true), *t-st* is executed; if no jump statements that exit from the loop are encountered, *cond-exp* is evaluated again. This cycle repeats until *cond-exp* is zero.

As with **if** statements, pointer type expressions can be compared with the null pointer, so that `while (ptr)...` is equivalent to

```
while (ptr != NULL)...
```

The **while** loop offers a concise method for scanning strings and other null-terminated data structures:

```
char str[10]="Borland";  
char *ptr=&str[0];  
int count=0;  
//...  
while (*ptr++) // loop until end of string  
    count++;
```

In the absence of jump statements, *t-st* must affect the value of *cond-exp* in some way, or *cond-exp* itself must change during evaluation in order to prevent unwanted endless loops.

### do while statements

The general format is

```
do do-st while (cond-exp);
```

The *do-st* statement is executed repeatedly until *cond-exp* compares equal to zero (false). The key difference from the **while** statement is that *cond-exp* is tested *after*, rather than before, each execution of the loop statement. At least one execution of *do-st* is assured. The same restrictions apply to the type of *cond-exp* (scalar).

for statements    The **for** statement format in C is

For C++, *<init-exp>* can be an expression or a declaration.

**for** (*<init-exp>*; *<test-exp>*; *<increment-exp>*) *statement*

The sequence of events is as follows:

1. The initializing expression *init-exp*, if any, is executed. As the name implies, this usually initializes one or more loop counters, but the syntax allows an expression of any degree of complexity (including declarations in C++). Hence the claim that any C program can be written as a single **for** loop. (But don't try this at home. Such stunts are performed by trained professionals.)
2. The expression *test-exp* is evaluated following the rules of the **while** loop. If *test-exp* is nonzero (true), the loop statement is executed. An empty expression here is taken as **while** (1), that is, always true. If the value of *test-exp* is zero (false), the **for** loop terminates.
3. *increment-exp* advances one or more counters.
4. The expression *statement* (possibly empty) is evaluated and control returns to step 2.

If any of the optional elements are empty, appropriate semicolons are required:

```
for (;;) {           // same as for (; 1;)
    // loop forever
}
```



The C rules for **for** statements apply in C++. However, the *init-exp* in C++ can also be a declaration. The scope of a declared identifier extends to the end of the controlled statement, not beyond. For example,

```
for (int i = 1; i < j; ++i)
{
    if (i ...) ...           // ok to refer to i here
}
if (i...)                   // illegal; i is now out of scope
```

---

## Jump statements

A jump statement, when executed, transfers control unconditionally. There are four such statements: **break**, **continue**, **goto**, and **return**.

break statements    The syntax is

**break;**

A **break** statement can be used only inside an iteration (**while**, **do**, and **for** loops) or a **switch** statement. It terminates the iteration or **switch** statement. Since iteration and **switch** statements can be intermixed and nested to any depth, take care to ensure that your **break** exits from the correct loop or switch. The rule is that a **break** terminates the *nearest* enclosing iteration or **switch** statement.

continue statements    The syntax is

**continue;**

A **continue** statement can be used only inside an iteration statement; it transfers control to the test condition for **while** and **do** loops, and to the increment expression in a **for** loop.

With nested iteration loops, a **continue** statement is taken as belonging to the *nearest* enclosing iteration.

goto statements    The syntax is

**goto label;**

The **goto** statement transfers control to the statement labeled *label* (see “Labeled statements,” page 92), which must be in the same function.



In C++, it is illegal to bypass a declaration having an explicit or implicit initializer unless that declaration is within an inner block that is also bypassed.

return statements    Unless the function return type is **void**, a function body must contain at least one **return** statement with the following format:

**return return-expression;**

where *return-expression* must be of type **type** or of a type that is convertible to **type** by assignment. The value of the *return-expression* is the value returned by the function. An expression that calls the function, such as `func(actual-arg-list)`, is an rvalue of type **type**, not an lvalue:

```
t = func(arg);        // OK
func(arg) = t;        /* illegal in C; legal in C++ if return type of
                      func is a reference */
```

```
(func(arg))++; /* illegal in C; legal in C++ if return type of
               func is a reference */
```

The execution of a function call terminates if a **return** statement is encountered; if no **return** is met, execution “falls through,” ending at the final closing brace of the function body.

If the return type is **void**, the **return** statement can be written as

```
{
    ...
    return;
}
```

with no return expression; alternatively, the **return** statement can be omitted.

## C++

---

C++ is basically a superset of C. This means that, generally speaking, you can compile C programs under C++, but you can't compile a C++ program under C if the program uses any constructs peculiar to C++. Some situations need special care. The same function **func** declared twice in C with different argument types will invoke a duplicated name error. Under C++, however, **func** will be interpreted as an overloaded function—whether this is legal or not will depend on other circumstances. For a general discussion of programming in C++, see Chapter 5, “A C++ primer,” in *Getting Started*. Chapter 6, “Hands-on C++,” also in *Getting Started*, gives you a quick feeling for how C++ constructs work.

Although C++ introduces new keywords and operators to handle classes, some of the capabilities of C++ have applications outside of any class context. We first review these aspects of C++ that can be used independently of classes, then get into the specifics of classes and class mechanisms.

---

### Referencing

*Pointer referencing and dereferencing is discussed on page 80.*

C++ reference types are closely related to pointer types. C++ *reference types* create aliases for objects and let you pass arguments to functions by reference. Traditional C passes arguments only by *value*. In C++ you can pass arguments by value or by reference.

Simple references    The reference declarator can be used to declare references outside functions:

```
int i = 0;
int &ir = i; // ir is an alias for i
ir = 2;     // same effect as i = 2
```

This creates the lvalue *ir* as an alias for *i*, provided that the initializer is the same type as the reference. Any operations on *ir* have precisely the same effect as operations on *i*. For example, *ir = 2* assigns 2 to *i*, and *&ir* returns the address of *i*.

Reference arguments    The reference declarator can also be used to declare reference type parameters within a function:

```
void func1 (int i);
void func2 (int &ir); // ir is type "reference to int"
...
int sum=3;
func1(sum); // sum passed by value
func2(sum); // sum passed by reference
```

The *sum* argument passed by reference can be changed directly by **func2**. **func1**, on the other hand, gets a copy of the *sum* argument (passed by value), so *sum* itself cannot be altered by **func1**.

When an actual argument *x* is passed by value, the matching formal argument in the function receives a copy of *x*. Any changes to this copy within the function body are not reflected in the value of *x* itself. Of course, the function can *return* a value that could be used later to change *x*, but the function cannot directly alter a parameter passed by value.

The traditional C method for changing *x* uses the actual argument *&x*, the address of *x*, rather than *x* itself. Although *&x* is passed by value, the function can access *x* through the copy of *&x* it receives. Even if the function does not need to change *x*, it is still useful (though subject to possibly dangerous side effects) to pass *&x*, especially if *x* is a large data structure. Passing *x* directly by value involves the wasteful copying of the data structure.

Compare the three implementations of the function **treble**:

```
Implementation 1    int treble_1(n)
                   {
                   return 3*n;
                   }
```

```

...
int x, i = 4;
x = treble_1(i);      // x now = 12, i = 4
...
Implementation 2 void treble_2(int* np)
{
    *np = (*np)*3;
}
...
treble_2(int &i);      // i now = 12
Implementation 3 void treble_3(int& n) // n is a reference type
{
    n = 3*n;
}
...
treble_3(i);          // i now = 36

```

The formal argument declaration **type& t** (or equivalently, **type &t**) establishes *t* as type “reference to **type**.” So, when **treble\_3** is called with the real argument *i*, *i* is used to initialize the formal reference argument *n*. *n* therefore acts as an alias for *i*, so that *n* = 3\**n* also assigns 3 \* *i* to *i*.

If the initializer is a constant or an object of a different type than the reference type, Turbo C++ creates a temporary object for which the reference acts as an alias:

```

int& ir = 6; /* temporary int object created, aliased by ir, gets value
             6 */
float f;
int& ir2 = f; /* creates temporary int object aliased by ir2; f converted
              before assignment */
ir2 = 2.0 // ir2 now = 2, but f is unchanged

```

The automatic creation of temporary objects permits the conversion of reference types when formal and actual arguments have different (but assignment-compatible) types. When passing by value, of course, there are fewer conversion problems, since the copy of the actual argument can be physically changed before assignment to the formal argument.

---

## Scope access operator

The scope access (or resolution) operator **::** (two semicolons) lets you access a global (or file duration) name even if it is hidden by a local redeclaration of that name (see page 29):

This code also works if the "global" `i` is a file-level static.

```
int i; // global i
...
void func(void);
{
    int i=0; // local i hides global i
    i = 3; // this i is the local i
    ::i = 4; // this i is the global i
    printf ("%d", i); // prints out 3
}
```

The `::` operator has other uses with class types, as discussed throughout this chapter.

---

## The new and delete operators

The **new** and **delete** operators offer dynamic storage allocation and deallocation, similar but superior to the standard library functions in the **malloc** and **free** families (see the *Library Reference*).

A simplified syntax is

```
pointer-to-name = new name <name-initializer>;
delete pointer-to-name;
```

*name* can be of any type except "function returning..." (however, pointers to functions are allowed).

**new** tries to create an object of type *name* by allocating (if possible) **sizeof(name)** bytes in *free store* (also called the heap). The storage duration of the new object is from the point of creation until the operator **delete** kills it by deallocating its memory, or until the end of the program.

If successful, **new** returns a pointer to the new object. A null pointer indicates a failure (such as insufficient or fragmented heap memory). As with **malloc**, you need to test for null before trying to access the new object. However, unlike **malloc**, **new** calculates the size of *name* without the need for an explicit **sizeof** operator. Further, the pointer returned is of the correct type, "pointer to *name*," without the need for explicit casting.

*new*, being a keyword, doesn't need a prototype.

```
name *nameptr; // name is any non-function type
...
if (!(nameptr = new name)) {
    errmsg("Insufficient memory for name");
    exit (1);
}
// use *nameptr to initialize new name object
```

```
...
delete nameptr; // destroy name and deallocate sizeof(name) bytes
```

The operator new with arrays

If *name* is an array, the pointer returned by **new** points to the first element of the array. When creating multidimensional arrays with **new**, all array sizes must be supplied:

```
mat_ptr = new int[3][10][12]; // OK
mat_ptr = new int[3][][12]; // illegal
mat_ptr = new int[][10][12]; // illegal
```

The ::operator new

When used with non-class objects, **new** works by calling a standard library routine, the global **::operator new**. With class objects of type *name*, a specific operator called *name::operator new* can be defined. **new** applied to class *name* objects invokes the appropriate *name::operator new* if present; otherwise, the standard **::operator new** is used.

Initializers with the new operator

The optional initializer is another advantage **new** has over **malloc** (although **calloc** does clear its allocations to zero). In the absence of explicit initializers, the object created by **new** contains unpredictable data (garbage). The objects allocated by **new**, other than arrays, can be initialized with a suitable expression between parentheses:

```
int_ptr = new int(3);
```

Arrays of classes with constructors are initialized with the *default constructor* (see page 115). The user-defined **new** operator with customized initialization plays a key role in C++ constructors for class-type objects.

---

## Classes

C++ classes offer extensions to the predefined type system. Each class type represents a unique set of objects and the operations (methods) and conversions available to create, manipulate, and destroy such objects. Derived classes can be declared that *inherit* the members of one or more *base* (or parent) classes.

In C++, structures and unions are considered as classes with certain access defaults.

A simplified, "first-look" syntax for class declarations is

```
class-key class-name <: base-list> { <member-list> }
```

*class-key* is one of **class**, **struct**, or **union**.

The optional *base-list* lists the base class or classes from which the class *class-name* will derive (or *inherit*) objects and methods. If any base classes are specified, the class *class-name* is called a derived class (see page 110, “Base and derived class access”). The *base-list* has default and optional overriding *access specifiers* that can modify the access rights of the derived class to members of the base classes (see page 108, “Member access control”).

The optional *member-list* declares the class members (data and functions) of *class-name* with default and optional overriding access specifiers that may affect which functions can access which members.

**Class names** *class-name* is any identifier unique within its scope. With structures, classes, and unions, *class-name* can be omitted (see “Untagged structures and typedefs,” page 65.)

**Class types** The declaration creates a unique type, class type *class-name*. This lets you declare further *class objects* (or *instances*) of this type, and objects derived from this type (such as pointers to, references to, arrays of *class-name*, and so on):

```
class X { ... };
X x, &xr, *xpтр, xarray[10];
/* four objects: type X, reference to X, pointer to X and array of X*/

struct Y { ... };
Y y, &yr, *ypтр, yarray[10];
// C would have
// struct Y y, &yr, *ypтр, yarray[10];

union Z { ... };
Z z, &zr, *zpтр, zarray[10];
// C would have
// union Z z, &zr, *zpтр, zarray[10];
```

Note the difference between C and C++ structure and union declarations: The keywords **struct** and **union** are essential in C, but in C++ they are needed only when the class names, **Y** and **Z**, are hidden (see the following section).

**Class name scope** The scope of a class name is local, with some tricks peculiar to classes. Class name scope starts at the point of declaration and ends with the enclosing block. A class name hides any class, object, enumerator, or function with the same name in the enclosing scope. If a class name is declared in a scope containing the declaration of an object, function, or enumerator of the same name, the class can only be referred to

using the *elaborated type specifier*. This means that the class key, **class**, **struct**, or **union** must be used with the class name. For example,

```
struct S { ... };  
  
int S(struct S *Sptr);  
  
void func(void)  
{  
    S t;           // ILLEGAL declaration: no class key  
                  // and function S in scope  
    struct S s;   // OK: elaborated with class key  
    S(&s);        // OK: this is a function call  
}
```

C++ also allows an incomplete class declaration:

```
class X; // no members, yet!  
struct Y;  
union Z;
```

Incomplete declarations permit certain references to the class names **X**, **Y**, or **Z** (usually references to pointers to class objects) before the classes have been fully defined (see “Structure member declarations,” page 65). Of course, you must make a complete class declaration with members before you can declare and use class objects.

**Class objects** Class objects can be assigned (unless copying has been restricted), passed as arguments to functions, returned by functions (with some exceptions), and so on. Other operations on class objects and members can be user-defined in many ways, including member and friend functions, and the redefinition of standard functions and operators when used with objects of a certain class. Redefined functions and operators are said to be *overloaded*. Operators and functions that are restricted to objects of a certain class (or related group of classes) are called *member functions* for that class. C++ offers a mechanism whereby the same function or operator name can be called to perform different tasks, depending on the type or number of arguments or operands.

**Class member list** The optional *member-list* is a sequence of data declarations (of any type, including enumerations, bit fields and other classes) and function declarations and definitions, all with optional storage class specifiers and access modifiers. The objects thus defined are called *class members*. The storage class specifiers **auto**, **extern**, and **register** are not allowed. Members can be declared with the **static** storage class specifiers.

Member functions A function declared without the **friend** specifier is known as a *member function* of the class. Functions declared with the **friend** modifier are called *friend functions*.

The same name can be used to denote more than one function, provided that they differ in argument type or number of arguments.

The keyword **this** Nonstatic member functions operate on the class type object with which they are called. For example, if  $x$  is an object of class **X** and **f** is a member function of **X**, the function call  $x.f()$  operates on  $x$ . Similarly, if  $xptr$  is a pointer to an **X** object, the function call  $xptr->f()$  operates on  $*xptr$ . But how does **f** know which  $x$  it is operating on? C++ provides **f** with a pointer to  $x$  called **this**. **this** is passed as a hidden argument in all calls to non-static member functions.

The keyword **this** is a local variable available in the body of any nonstatic member function. **this** does not need to be declared and is rarely referred to explicitly in a function definition. However, it is used implicitly within the function for member references. If  $x.f(y)$  is called, for example, where  $y$  is a member of **X**, **this** is set to  $\&x$  and  $y$  is set to **this->y**, which is equivalent to  $x.y$ .

Inline functions You can declare a member function within its class and define it elsewhere. Alternatively, you can both declare and define a member function within its class, in which case it is called an *inline* function. (Chapter 5, "A C++ primer," in *Getting Started* gives some examples of inline functions.)

Turbo C++ can sometimes reduce the normal function call overhead by substituting the function call directly with the compiled code of the function body. This process, called an *inline expansion* of the function body, does not affect the scope of the function name or its arguments. Inline expansion is not always possible or feasible. The *inline* specifier is a request (or hint) to the compiler that you would welcome an inline expansion. As with the **register** storage class specifier, the compiler may or may not take the hint!

Explicit and implicit **inline** requests are best reserved for small, frequently used functions, such as the *operator functions* that implement overloaded operators. For example, the following class declaration:

```
int i;                // global int
class X {
public:
```

```

        char* func(void) { return i; } // inline by default
        char* i;
    };

```

is equivalent to:

```

inline char* X::func(void) { return i; }

```

**func** is defined “outside” the class with an explicit inline specifier. The *i* returned by **func** is the **char\*** *i* of class **X**—see the section on member scope starting on page 107.

**Static members** The storage class specifier **static** can be used in class declarations of data and function members. Such members are called *static members* and have distinct properties from nonstatic members. With nonstatic members, a distinct copy “exists” for each object in its class; with static members, only one copy exists, and it can be accessed without reference to any particular object in its class. If *x* is a static member of class **X**, it can be referenced as **X::x** (even if objects of class **X** haven’t been created yet). It is still possible to access *x* using the normal member access operators. For example, *y.x* and *yptr->x*, where *y* is an object of class **X** and *yptr* is a pointer to an object of class **X**, although the expressions *y* and *yptr* are *not* evaluated. In particular, a static member function can be called with or without the special member function syntax:

```

class X {
    int member_int;
public:
    static void func(int i, X* ptr);
};

void g(void);
{
    X obj;
    func(1, &obj); // error unless there is a global func()
                  // defined elsewhere

    X::func(1, &obj); // calls the static func() in X
                    // OK for static functions only
    obj.func(1, &obj); // so does this (OK for static and
                      // nonstatic functions)
}

```

Since a static member function can be called with no particular object in mind, it has no **this** pointer. A consequence of this is that a static member function cannot access nonstatic members without explicitly specifying an object with **.** or **->**. For example, with the declarations of the previous example, **func** might be defined as follows:

```

void X::func(int i, X* ptr)
{
    member_int = i;           // which object does member_int
                              // refer to? Error
    ptr->member_int = i;     // OK: now we know!
}

```

Apart from inline functions, static member functions of global classes have external linkage. Static member functions cannot be virtual functions. It is illegal to have a static and nonstatic member function with the same name and argument types.

The declaration of a static data member in its class declaration is not a definition, so a definition must be provided elsewhere to allocate storage and provide initialization. The definition of a static data member can be omitted if “default initialization to all zeros” is in operation.

Static members of a class declared local to some function have no linkage and cannot be initialized. Static members of a global class can be initialized like ordinary global objects, but only in file scope. Static members obey the usual class member access rules, except they can be initialized.

```

class X {
    ...
    static int x;
    ...
};

int X::x = 1;

```

The main use for static members is to keep track of data common to all objects of a class, such as the number of objects created, or the last-used resource from a pool shared by all such objects. Static members are also used to

- reduce the number of visible global names
- make obvious which static objects logically belong to which class
- permit access control to their names

**Member scope** The expression **X::func()** in the example on page 106 uses the class name **X** with the scope access modifier to signify that **func**, although defined “outside” the class, is indeed a member function of **X**, and it exists within the scope of **X**. The influence of **X::** extends into the body of the definition. This explains why the *i* returned by **func** refers to **X::i**, the *char\** *i* of **X**, rather than the global **int i**. Without the **X::** modi-

fier, the function **func** would represent an ordinary non-class function, returning the global **int i**.

All member functions, then, are in the scope of their class, even if defined outside the class.

Data members of class **X** can be referenced using the selection operators **.** and **->** (as with C structures). Member functions can also be called using the selection operators (see also "The keyword **this**," page 105). For example,

```
class X {
public:
    int i;
    char name[20];
    X *ptr1;
    X *ptr2;
    void Xfunc(char*data, X* left, X* right); // define elsewhere
};
void f(void);
{
    X x1, x2, *xptr=&x1;
    x1.i = 0;
    x2.i = x1.i;
    xptr->i = 1;
    x1.Xfunc("stan", &x2, xptr);
}
```

If *m* is a member or base member of class **X**, the expression **X::m** is called a *qualified name*; it has the same type as *m*, and it is an lvalue only if *m* is an lvalue. A key point is that even if the class name **X** is hidden by a non-type name, the qualified name **X::m** will access the correct class member, *m*.

Class members cannot be added to a class by another section of your program. The class **X** cannot contain objects of class **X**, but can contain pointers or references to objects of class **X** (note the similarity with C's structure and union types).

### Member access control

Members of a class acquire access attributes either by default (depending on class key and declaration placement) or by the use of one of the three access specifiers: **public**, **private**, and **protected**. The significance of these attributes is as follows:

**public**      The member can be used by any function.

*Friend function declarations are not affected by access specifiers (see "Friends of classes," page 112).*

- private** The member can be used only by member functions and friends of the class in which it is declared.
- protected** Same as for **private**, but additionally, the member can be used by member functions and friends of classes *derived* from the declared class, but only in objects of the derived type. (Derived classes are explained in the next section.)

Members of a class are **private** by default, so you need explicit **public** or **protected** access specifiers to override the default.

Members of a **struct** are **public** by default, but you can override this with the **private** or **protected** access specifier.

Members of a **union** are **public** by default; this cannot be changed. All three access specifiers are illegal with union members.

A default or overriding access modifier remains effective for all subsequent member declarations until a different access modifier is encountered. For example,

```
class X {
    int i;    // X::i is private by default
    char ch; // so is X::ch
public:
    int j;    // next two are public
    int k;
protected:
    int l;    // X::l is protected
};

struct Y {
    int i;    // Y::i is public by default
private:
    int j;    // Y::j is private
public:
    int k;    // Y::k is public
};

union Z {
    int i;    // public by default; no other choice
    double d;
};
```

The access specifiers can be listed and grouped in any convenient sequence. You can save a little typing effort by declaring all the private members together, and so on.

Base and derived class access

When you declare a derived class **D**, you list the base classes **B1**, **B2**, ... in a comma-delimited *base-list*:

```
class-key D : base-list { <member-list> }
```

*Since a base class can itself be a derived class, the access attribute question is recursive: You backtrack until you reach the basest of the base classes, those that do not inherit.*

**D** inherits all the members of these base classes. (Redefined base class members are inherited and can be accessed using scope overrides, if needed.) **D** can use only the **public** and **protected** members of its base classes. But, what will be the access attributes of the inherited members as viewed by **D**? **D** may want to use a **public** member from a base class, but make it **private** as far as outside functions are concerned. The solution is to use access specifiers in the *base-list*.

When declaring **D**, you can use the access specifier **public** or **private** in front of the classes in the *base-list*:

*protected cannot be used in a base list. Unions cannot have base classes, and unions cannot be used as base classes.*

```
class D : public B1, private B2, ... {  
    ...  
}
```

These modifiers do not alter the access attributes of base members as viewed by the base class, though they *can* alter the access attributes of base members as viewed by the derived class.

The default is **private** if **D** is a **class** declaration, and **public** if **D** is a **struct** declaration.

The derived class inherits access attributes from a base class as follows:

**public** base class: **public** members of the base class are **public** members of the derived class. **Protected** members of the base class are **protected** members of the derived class. **Private** members of the base class remain **private** to the base class.

**private** base class: Both **public** and **protected** members of the base class are **private** members of the derived class. **Private** members of the base class remain **private** to the base class.

In both cases, note carefully that **private** members of a base class are, and remain, inaccessible to member functions of the derived class *unless friend* declarations are explicitly declared in the base class granting access. For example,

```
class X : A { // default for class is private A  
    ...
```

```

}
/* class X is derived from class A */
class Y : B, public C {    // override default for C
...
}
/* class Y is derived (multiple inheritance) from B and C
   B defaults to private B */

struct S : D {           // default for struct is public D
...                       /* struct S is derived from D */
}
struct T : private D, E { // override default for D
                        // E is public by default
...
}
/* struct T is derived (multiple inheritance) from D and E
   E defaults to public E */

```

The effect of access specifiers in the base list can be adjusted by using a *qualified-name* in the public or protected declarations in the derived class. For example,

```

class B {
    int a;                // private by default
public:
    int b, c;
    int Bfunc(void);
};

class X : private B {    // a, b, c, Bfunc are now private in X
    int d;                // private by default, NOTE: a is not
                        // accessible in X
public:
    B::c;                 // c was private, now is public
    int e;
    int Xfunc(void);
};

int Efunc(X& x);        // external to B and X

```

The function **Efunc** can use only the public names *c*, *e*, and **Xfunc**.

The function **Xfunc** is in **X**, which is derived from **private B**, so it has access to

- The “adjusted-to-public” *c*
- The “private-to-**X**” members from **B**: *b* and **Bfunc**
- **X**’s own private and public members: *d*, *e*, and **Xfunc**

However, **Xfunc** cannot access the “private-to-**B**” member, *a*.

## Virtual base classes

---

With multiple inheritance, a base class can't be specified more than once in a derived class:

```
class B { ...};
class D : B, B { ... }; // Illegal
```

However, a base class can be indirectly passed to the derived class more than once:

```
class X : public B { ... }
class Y : public B { ... }

class Z : public X, public Y { ... } // OK
```

In this case, each object of class **Z** will have two sub-objects of class **B**. If this causes problems, the keyword **virtual** can be added to a base class specifier. For example,

```
class X : virtual public B { ... }
class Y : virtual public B { ... }
class Z : public X, public Y { ... }
```

**B** is now a virtual base class, and class **Z** has only one sub-object of class **B**.

## Friends of classes

---

A **friend F** of a class **X** is a function or class that, although not a member function of **X**, has full access rights to the private and protected members of **X**. In all other respects, **F** is a normal function with respect to scope, declarations, and definitions.

Since **F** is not a member of **X**, it is not in the scope of **X** and it cannot be called with the *x.F* and *xptr->F* selector operators (where *x* is an **X** object, and *xptr* is a pointer to an **X** object).

If the specifier **friend** is used with a function declaration or definition within the class **X**, it becomes a friend of **X**.

Friend functions defined within a class obey the same inline rules as member functions (see "Inline functions," page 105). Friend functions are not affected by their position within the class or by any access specifiers. For example,

```
class X {
    int i; // private to X
    friend void friend_func(X*, int);
```

```

/* friend_func is not private, even though it's declared in the private
   section */
public:
    void member_func(int);
};

/* definitions; note both functions access private int i */
void friend_func(X* xptr, int a) { xptr->i = a; }
void X::member_func(int a) { i = a; }

X xobj;

/* note difference in function calls */
friend_func(&xobj, 6);
xobj.member_func(6);

```

You can make all the functions of class **Y** into friends of class **X** with a single declaration:

```

class Y; // incomplete declaration
class X {
    friend Y;
    int i;
    void member_funcX();
};

class Y; { // complete the declaration
    void friend_X1(X&);
    void friend_X2(X*);
    ...
};

```

The functions declared in **Y** are friends of **X**, although they have no **friend** specifiers. They can access the private members of **X**, such as *i* and **member\_funcX**.

It is also possible for an individual member function of class **X** to be a friend of class **Y**:

```

class X {
    ...
    void member_funcX();
}

class Y {
    int i;
    friend void X::member_funcX();
    ...
};

```

Class friendship is not transitive: **X** friend of **Y** and **Y** friend of **Z** does not imply **X** friend of **Z**. However, friendship *is* inherited.

## Constructors and destructors

---

There are several special member functions that determine how the objects of a class are created, initialized, copied, and destroyed. Constructors and destructors are the most important of these. They have many of the characteristics of normal member functions—you declare and define them within the class, or declare them within the class and define them outside—but they have some unique features.

1. They do not have return value declarations (not even **void**).
2. They cannot be inherited, though a derived class can call the base class' constructors and destructors.
3. Constructors, like most C++ functions, can have default arguments or use member initialization lists.
4. Destructors can be **virtual**, but constructors cannot.
5. You can't take their addresses.

```
main()
{
    ...
    void *ptr = base::base;    // illegal
    ...
}
```

6. Constructors and destructors can be generated by Turbo C++ if they haven't been explicitly defined; they are also invoked on many occasions without explicit calls in your program. Any constructor or destructor generated by the compiler will be public.
7. You cannot call constructors the way you call a normal function. Destructors can be called if you use their fully qualified name.

8.

```
{
    ...
    X *p;
    ...
    p->X::~~X();           // legal call of destructor
    X::X();                // illegal call of constructor
    ...
}
```

9. The compiler automatically calls constructors and destructors when defining and destroying objects.
10. Constructors and destructors can make implicit calls to operator **new** and operator **delete** if allocation is required for an object.

11. An object with a constructor or destructor cannot be used as a member of a union.

If a class **X** has one or more constructors, one of them is invoked each time you define an object *x* of class **X**. The constructor creates *x* and initializes it. Destructors reverse the process by destroying the class objects created by constructors.

Constructors are also invoked when local or temporary objects of a class are created; destructors are invoked when these objects go out of scope.

## Constructors

---

Constructors are distinguished from all other member functions by having the same name as the class they belong to. When an object of that class is created or is being copied, the appropriate constructor is called implicitly.

Constructors for global variables are called before function **main** is called. When the `pragma startup` directive is used to install a function prior to **main**, global variable constructors are called prior to the startup functions.

Local objects are created as the scope of the variable becomes active. A constructor is also invoked when a temporary object of the class is created.

```
class X
{
public:
    X(); // class X constructor
};
```

A class **X** constructor cannot take **X** as an argument:

```
class X {
...
public:
    X(X); // illegal
}
```

The parameters to the constructor can be of any type except that of the class of which it is a member. The constructor can accept a reference to its own class as a parameter; when it does so, it is called the *copy constructor*. A constructor which accepts no parameters is called the *default constructor*. We discuss the default constructor next; the description of the copy constructor starts on page 116.

The default constructor The default constructor for class **X** is one that takes no arguments: `X::X()`. If no user-defined constructors exist for a class, Turbo C++ generates a default constructor. On a declaration such as `X x`, the default constructor creates the object `x`.

**Important!** Like all functions, constructors can have default arguments. For example, the constructor

```
X::X(int, int = 0)
```

can take one or two arguments. When presented with one argument, the missing second argument is assumed to be a zero **int**. Similarly, the constructor

```
X::X(int = 5, int = 6)
```

could take two, one, or no arguments, with appropriate defaults. However, the default constructor `X::X()` takes *no* arguments and must not be confused with, say, `X::X(int = 0)`, which takes one or no arguments.

Take care to avoid ambiguity in calling constructors. In the following case, the default constructor and the constructor accepting an integer could become ambiguous:

```
class X
{
public:
    X();
    X(int i = 0);
};

main()
{
    X one(10); // OK; uses X::X(int)
    X two;    // illegal; ambiguous whether to call X::X() or
              // X::X(int = 0)
    ...
    return 0;
}
```

The copy constructor A copy constructor for class **X** is one that can be called with a single argument of type `X`: `X::X(const X&)` or `X::X(const X&, int = 0)`. Default arguments are also allowed in a copy constructor. Copy constructors are invoked when copying a class object, typically when you declare with initialization by another class object: `X x = y`. Turbo C++

generates a copy constructor for class **X** if one is needed and none is defined in class **X**.

**Overloading constructors** Constructors can be overloaded, allowing objects to be created, depending on the values being used for initialization.

```
class X
{
    int    integer_part;
    double double_part;
public:
    X(int i)    { integer_part = i; }
    X(double d) { double_part = d; }
};

main()
{
    X one(10); // invokes X::X(int) and sets integer_part to 10
    X one(3.14); // invokes X::X(double) setting double_part
    ...
    return 0;
}
```

**Order of calling constructors** In the case where a class has one or more base classes, the base class constructors are invoked before the derived class constructor. The base class constructors are called in the order they are declared.

For example, in this setup,

```
class Y {...}
class X : public Y {...}
X one;
```

the constructors are called in this order:

```
Y(); // base class constructor
X(); // derived class constructor
```

For the case of multiple base classes:

```
class X : public Y, public Z
X one;
```

the constructors are called in the order of declaration:

```
Y(); // base class constructors come first
Z();
X();
```

Constructors for virtual base classes are invoked before any non-virtual base classes. If the hierarchy contains multiple virtual base classes, the virtual base class constructors are invoked in the order in which they were declared. Any non-virtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a non-virtual base, that non-virtual base will be first so that the virtual base class may be properly constructed. The code

```
class X : public Y, virtual public Z
X one;
```

produces this order:

```
Z(); // virtual base class initialization
Y(); // non-virtual base class
X(); // derived class
```

Or for a more complicated example:

```
class base;
class base2;
class level1 : public base2, virtual public base;
class level2 : public base2, virtual public base;
class toplevel : public level1, virtual public level2;
toplevel view;
```

The construction order of view would be as follows:

```
base(); // virtual base class highest in hierarchy
// base is only constructed once
base2(); // non-virtual base of virtual base level2
// must be called to construct level2
level2(); // virtual base class
base2(); // non-virtual base of level1
level1(); // other non-virtual base
toplevel();
```

In the event that a class hierarchy contains multiple instances of a virtual base class, that base class is only constructed once. If, however, there exist both virtual and non-virtual instances of the base class, the class constructor is invoked a single time for all virtual instances and then once for each non-virtual occurrence of the base class.

Constructors for elements of an array are called in increasing order of the subscript.

**Class initialization** An object of a class with only public members and no constructors or base classes (typically a structure) can be initialized with an initializer list. If a class has a constructor, its objects must be either initialized or have a default constructor. The latter is used for objects not explicitly initialized.

Objects of classes with constructors can be initialized with an expression list in parentheses. This list is used as an argument list to the constructor. An alternative is to use an equal sign followed by a single value. The single value can be of the type of the first argument accepted by a constructor of that class, in which case either there are no additional arguments, or the remaining arguments have default values. It could also be an object of that class type. In the former case, the matching constructor is called to create the object. In the latter case, the copy constructor is called to initialize the object.

```
class X
{
    int i;
public:
    X();           // function bodies omitted for clarity
    X(int x);
    X(const X&);
};

main()
{
    X one;        // default constructor invoked
    X two(1);     // constructor X::X(int) is used
    X three = 1; // calls X::X(int)
    X four = one; // invokes X::X(const X&) for copy
    X five(two); // calls X::X(const X&)
}
```

The constructor can assign values to its members in two ways. It can accept the values as parameters and make assignments to the member variables within the function body of the constructor:

```
class X
{
    int a, b;
public:
    X(int i, int j) { a = i; b = j }
};
```

Or it can use an initializer list prior to the function body:

```
class X
```

```

{
    int a, b;
public:
    X(int i, int j) : a(i), b(j) {}
};

```

In both cases, an initialization of `X x(1, 2)` assigns a value of 1 to `x::a` and 2 to `x::b`. The second method, the initializer list, provides a mechanism for passing values along to base class constructors.

*Base class constructors must be declared as either **public** or **protected** to be called from a derived class.*

```

class base1
{
    int x;
public:
    base1(int i) { x = i; }
};

class base2
{
    int x;
public:
    base2(int i) : x(i) {}
};

class top : public base1, public base2
{
    int a, b;
public:
    top(int i, int j) : base1(i*5), base2(j+1), a(i) { b = j;}
};

```

With this class hierarchy, a declaration of `top one(1, 2)` would result in the initialization of **base1** with the value 5 and **base2** with the value 3. The methods of initialization can be intermixed.

As described previously, the base classes are initialized in declaration order. Then the members are initialized, also in declaration order, independent of the initialization list.

```

class X
{
    int a, b;
public:
    X(int i, j) : a(i), b(a+j) {}
};

```

With this class, a declaration of `X x(1,1)` results in an assignment of 1 to `x::a` and 2 to `x::b`.

Base class constructors are called prior to the construction of any of the derived classes members. The values of the derived class can't be changed and then have an affect on the base class's creation.

```
class base
{
    int x;
public:
    base(int i) : x(i) {}
};

class derived : base
{
    int a;
public:
    derived(int i) : a(i*10), base(a) { } // Watch out! Base will be
                                        // passed an uninitialized a.
};
```

With this class setup, a call of derived **d(1)** will *not* result in a value of 10 for the base class member *x*. The value passed to the base class constructor will be undefined.

When you want an initializer list in a non-inline constructor, don't place the list in the class definition. Instead, put it at the point at which the function is defined.

```
derived::derived(int i) : a(i)
{
    ...
}
```

---

## Destructors

The destructor for a class is called to free members of an object before the object is itself destroyed. The destructor is a member function whose name is that of the class preceded by a tilde (~). A destructor cannot accept any parameters, nor will it have a return type or value declared.

```
class X
{
public:
    ~X(); // destructor for class X
};
```

If a destructor is not explicitly defined for a class, the compiler will generate one.

- When destructors are invoked
- A destructor is called implicitly when a variable goes out of its declared scope. Destructors for local variables are called when the block they are declared in is no longer active. In the case of global variables, destructors are called as part of the exit procedure after **main**.
- When pointers to objects go out of scope, a destructor is not implicitly called. This means that the **delete** operator must be called to destroy such an object.
- Destructors are called in the exact opposite order from which their corresponding constructors were called (see page 117).
- atexit**, **#pragma exit**, and destructors
- All global objects are active until the code in all exit procedures has executed. Local variables, including those declared in **main**, are destroyed as they go out of scope. The order of execution at the end of a Turbo C++ program in these regards is as follows:
- **atexit** functions are executed in the order they were inserted.
  - **#pragma** exit functions are executed in the order of their priority codes.
  - Destructors for global variables are called.
- exit** and destructors
- When you call **exit** from within a program, destructors are not called for any local variables in the current scope. Global variables are destroyed in their normal order.
- abort** and destructors
- If you call **abort** anywhere in a program, no destructors are called, not even for variables with a global scope.
- A destructor can also be invoked explicitly in one of two ways: indirectly through a call to **delete**, or directly by using the destructor's fully qualified name. You can use **delete** to destroy objects that have been allocated using **new**. Explicit calls to the destructor are only necessary for objects allocated a specific address through calls to **new**.

```

class X {
    ...
    ~X();
    ...
};

void* operator new(size_t size, void *ptr)
{

```

```

        return ptr;
    }
    char buffer[sizeof(X)];

    main()
    {
        X* pointer = new X;
        X* exact_pointer;

        exact_pointer = new(&buffer) X; // pointer initialized at
                                        // address of buffer

        ...

        delete pointer;                // delete used to destroy pointer
        exact_pointer->X::~~X();        // direct call used to deallocate
    }

```

**Virtual destructors** A destructor can be declared as virtual. This allows a pointer to a base class object to call the correct destructor in the event that the pointer actually refers to a derived class object. The destructor of a class derived from a class with a virtual destructor is itself virtual.

```

class color
{
public:
    virtual ~color();    // virtual destructor for color
};

class red : public color
{
public:
    ~red();              // destructor for red is also virtual
};

class brightred: public red
{
public:
    ~brightred();       // brightred's destructor also virtual
};

```

The previously listed classes and the following declarations

```

color *palette[3];

palette[0] = new red;
palette[1] = new brightred;
palette[2] = new color;

```

will produce these results

```

delete palette[0];

```

```

// The destructor for red is called followed by the
// destructor for color.

delete palette[1];
// The destructor for brightred is called, followed by ~red
// and ~color.

delete palette[2];
// The destructor for color is invoked.

```

However, in the event that no destructors were declared as virtual, **delete palette[0]**, **delete palette[1]**, and **delete palette[2]** would all call only the destructor for class **color**. This would incorrectly destruct the first two elements, which were actually of type **red** and **brightred**.

## Overloaded operators

C++ lets you redefine the action of most operators, so that they perform specified functions when used with objects of a particular class. As with overloaded C++ functions in general, the compiler distinguishes the different functions by noting the context of the call: the number and types of the arguments or operands.

All the operators on page 20 can be overloaded except for

. .\* :: ?:

The preprocessing symbols # and ## also cannot be overloaded.

The keyword **operator** followed by the operator symbol is called the *operator function name*; it is used like a normal function name when defining the new (overloaded) action of the operator.

A function operator called with arguments behaves like an operator working on its operands in an expression. The operator function can't alter the number of arguments or the precedence and associativity rules (Table 1.20 on page 75) applying to normal operator use. Consider the class *complex*:

*This class was invented for illustrative purposes only. It isn't the same as the class **complex** in the run-time library.*

```

class complex {
    double real, imag;           // private by default
public:
    ...
    complex() { real = imag = 0; } // inline constructor
    complex(double r, double i = 0) { // another one
        real = r; imag = i;
    }
    ...
}

```

We could easily devise a function for adding complex numbers, say,

```
complex AddComplex(complex c1, complex c2);
```

but it would be more natural to be able to write:

```
complex c1(0,1), c2(1,0), c3  
c3 = c1 + c2;
```

than

```
c3 = AddComplex(c1, c2);
```

The operator **+** is easily overloaded by including the following declaration in the class *complex*:

```
friend complex operator +(complex c1, complex c2);
```

and defining it (possibly inline) as:

```
complex operator +(complex c1, complex c2)  
{  
    return complex{c1.real + c2.real, c1.imag + c2.imag};  
}
```

---

## Operator functions

Operator functions can be called directly, although they are usually invoked indirectly by the use of the overload operator:

```
c3 = c1.operator + (c2); // same as c3 = c1 + c2
```

Apart from **new** and **delete**, which have their own rules (see the next sections), an operator function must either be a nonstatic member function or have at least one argument of class type. The operator functions **=**, **()**, **[]** and **->** must be nonstatic member functions.

### Overloaded operators and inheritance

With the exception of the assignment function operator **=()** (see “Overloading the assignment operator **=**” on page 127), all overloaded operator functions for class **X** are inherited by classes derived from **X**, with the standard resolution rules for overloaded functions. If **X** is a base class for **Y**, an overloaded operator function for **X** may possibly be further overloaded for **Y**.

### Overloading **new** and **delete**

The operators **new** and **delete** can be overloaded to provide alternative free storage (heap) memory-management routines. A user-defined operator **new** must return a **void\*** and must have a **size\_t** as its first argument. A user-defined operator **delete** must have a **void**

*The type **size\_t** is defined in **stdlib.h**.*

return type and **void\*** as its first argument; a second argument of type **size\_t** is optional. For example,

```
#include <stdlib.h>

class X {
    ...
public:
    void* operator new(size_t size) { return newalloc(size);}
    void operator delete(void* p) { newfree(p); }
    X() { /* initialize here */ }
    X(char ch) { /* and here */ }

    ~X() { /* clean up here */ }
    ...
};
```

The *size* argument gives the size of the object being created, and **newalloc** and **newfree** are user-supplied memory allocation and deallocation functions. Constructor and destructor calls for objects of class **X** (or objects of classes derived from **X** that do not have their own overloaded operators **new** and **delete**) will invoke the matching user-defined **X::operator new()** and **X::operator delete()**, respectively.

The **X::operator new** and **X::operator delete** operator functions are static members of **X** whether explicitly declared as **static** or not, so they cannot be virtual functions.

The standard, predefined (global) **new** and **delete** operators can still be used within the scope of **X**, either explicitly with the global scope operator (**::operator new** and **::operator delete**), or implicitly when creating and destroying non-**X** or non-**X**-derived class objects. For example, you could use the standard **new** and **delete** when defining the overloaded versions:

```
void* X::operator new(size_t s)
{
    void* ptr = new char[s]; // standard new called
    ...
    return ptr;
}

void X::operator delete(void* ptr)
{
    ...
    delete (void*) ptr; // standard delete called
}
```

The reason for the *size* argument is that classes derived from **X** inherit the **X::operator new**. The size of a derived class object may well differ from that of the base class.

### Overloading unary operators

You can overload a prefix or postfix unary operator by declaring a nonstatic member function taking no arguments, or by declaring a non-member function taking one argument. If @ represents a unary operator, @x and x@ can both be interpreted as either **x.operator@()** or **operator@(x)**, depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

Care is needed when overloading ++ and --, since postfix and prefix usage cannot be distinguished from within the overloading function. For example,

```
class X {
    ...
    X operator ++() { /* increment X routine here */ }
}
...
X x, y;
y = ++x; // same as y = x++ !
```

### Overloading binary operators

You can overload a binary operator by declaring a nonstatic member function taking one argument, or by declaring a non-member function (usually **friend**) taking two arguments. If @ represents a binary operator, x@y can be interpreted as either **x.operator@(y)** or **operator@(x,y)**, depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

### Overloading the assignment operator =

The assignment operator = can only be overloaded by declaring a nonstatic member function. For example,

```
class String {
    ...
    String& operator = (String& str);
    ...
    String (String&);
    ~String();
}
```

This code, with suitable definitions of **String::operator =()**, allows string assignments *str1 = str2*, just like other languages. Unlike the other operator functions, the assignment operator function cannot be

inherited by derived classes. If, for any class **X**, there is no user-defined operator =, the operator = is defined by default as a member-by-member assignment of the members of class **X**:

```
X& X::operator = (const X& source)
{
    // memberwise assignment
}
```

Overloading the  
function call  
operator ()

The function call

*primary-expression* ( <*expression-list*> )

is considered a binary operator with operands *primary-expression* and *expression-list* (possibly empty). The corresponding operator function is **operator()**. This function can be user-defined for a class **X** (and any derived classes) only by means of a nonstatic member function. A call *x*(*arg1*, *arg2*), where *x* is an object of class **X**, is interpreted as *x.operator()*(*arg1*, *arg2*).

Overloading the  
subscripting operator [ ]

Similarly, the subscripting operation

*primary-expression* [ *expression* ]

is considered a binary operator with operands *primary-expression* and *expression*. The corresponding operator function is **operator[]**; this can be user-defined for a class **X** (and any derived classes) only by means of a nonstatic member function. The expression *x*[*y*], where *x* is an object of class **X**, is interpreted as *x.operator[]* (*y*).

Overloading the class  
member access  
operator ->

Class member access using

*primary-expression* -> *expression*

is considered a unary operator. The function **operator->** must be a nonstatic member function. The expression *x->m*, where *x* is a class **X** object, is interpreted as (*x.operator->*())->*m*, so that the function **operator->()** must either return a pointer to a class object or return an object of a class for which **operator->** is defined.

---

## Virtual functions

Virtual functions allow derived classes to provide different versions of a base class function. You can declare a virtual function in a base class, then redefine it in any derived class, even if the number and type of arguments are the same. You can also declare the functions `int`

Base::Fun (int) and int Derived::Fun (int) even when they are not virtual. The base class version is available to derived class objects via scope override. If they are virtual, only the function associated with the actual type of the object is available.

With virtual functions, you cannot change just the function type. It is illegal, therefore, to redefine a virtual function so that it differs only in the return type. If two functions with the same name have different arguments, C++ considers them different, and the virtual function mechanism is ignored.

The redefined function is said to *override* the base class function. The **virtual** specifier is used to declare a virtual function. The **virtual** specifier implies membership, so a virtual function cannot be a global (nonmember) function.

If a base class **B** contains a virtual function **vf**, and class **D**, derived from **B**, contains a function **vf** of the same type, then if **vf** is called for an object *d* or **D**, the call made is **D::vf**, even if the access is via a pointer or reference to **B**. For example,

```
struct B {
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    void f();
};
class D : public B {
    virtual void vf1(); // virtual specifier is legal but redundant
    void vf2(int);     // not virtual, since it's using a different
                      // arg list
    char vf3();       // Illegal: return-type-only change!
    void f();
};
void extf()
{
    D d;           // declare a D object
    B* bp = &d;   // standard conversion from D* to B*
    bp->vf1();     // calls D::vf1
    bp->vf2();     // call B::vf2 since D's vf2 has different args
    bp->f();       // calls B::f (not virtual)
}
```

The overriding function **vf1** in **D** is automatically virtual. The **virtual** specifier *can* be used with an overriding function declaration in the derived class, but its use is redundant.

The interpretation of a virtual function call depends on the type of the object for which it is called; with non-virtual function calls, the interpretation depends only on the type of the pointer or reference denoting the object for which it is called.

**Note** Virtual functions must be members of some class, but they cannot be static members. A virtual function can be a **friend** of another class.

A virtual function in a base class, like all member functions of a base class, must be defined or, if not defined, declared *pure*:

```
class B {  
    virtual void vf(int) = 0;    // = 0 means 'pure'
```

In a class derived from such a base class, each pure function must be defined or redeclared as pure (see the next section, "Abstract classes").

If a virtual function is defined in the base it need not necessarily be redefined in the derived class. Calls will simply call the base function.

Virtual functions exact a price for their versatility: Each object in the derived class needs to carry a pointer to a table of functions in order to select the correct one at run time (late binding). See Chapter 5, "A C++ primer," in *Getting Started*.

---

## Abstract classes

Chapter 5, "A C++ primer," in *Getting Started* gives an example of an abstract class in action.

An *abstract class* is a class with at least one pure virtual function. A virtual function is specified as pure by using the pure-specifier.

An abstract class can be used only as a base class for other classes. No objects of an abstract class can be created. An abstract class cannot be used as an argument type or as a function return type. However, you can declare pointers to an abstract class. References to an abstract class are allowed, provided that a temporary object is not needed in the initialization. For example:

```
class shape {          // abstract class  
    point center;  
    ...  
public:  
    where() { return center; }  
    move(point p) { center = p; draw(); }  
    virtual void rotate(int) = 0; // pure virtual function  
    virtual void draw() = 0;      // pure virtual function  
    virtual void hilight() = 0;  // pure virtual function  
    ...  
}
```

```

shape x;           // ERROR: attempted creation of an object of
                  // an abstract class
shape* sptr;      // pointer to abstract class is OK
shape f();        // ERROR: abstract class cannot be a return
                  // type
int g(shape s);   // ERROR: abstract class cannot be a
                  //function argument type
shape& h(shape&); // reference to abstract class as return
                  // value or function argument is OK

```

Suppose that **D** is a derived class with the abstract class **B** as its immediate base class. Then for each pure virtual function **pvf** in **B**, **D** must either provide a definition for **pvf**, or **D** must declare **pvf** as pure.

For example, using the class `shape` outlined above,

```

class circle : public shape { // circle derived from
                              // abstract class
    int radius;               // private

public:
    void rotate(int) { }      // virtual function defined:
                              // no action to rotate a
                              // circle
    void draw();              // circle::draw must be
                              // defined somewhere
    void hilite() = 0;        // redeclare as pure
}

```

Member functions can be called from a constructor of an abstract class, but calling a pure virtual function directly or indirectly from such a constructor provokes a run-time error.

## C++ scope

---

The lexical scoping rules for C++, apart from class scope, follow the general rules for C, with the proviso that C++, unlike C, permits both data and function declarations to appear wherever a statement may appear. The latter flexibility means that care is needed when interpreting such phrases as “enclosing scope” and “point of declaration.”

**Class scope** The name *M* of a member of a class **X** has class scope “local to **X**,” it can only be used in the following situations:

- In member functions of **X**
- In expressions such as *x.M*, where *x* is an object of **X**
- In expressions such as *xptr->M*, where *xptr* is a pointer to an object of **X**

- In expressions such as **X::M** or **D::M**, where **D** is a derived class of **X**
- In forward references within the class of which it is a member.

Classes, enumerations, or **typedef** names declared within a class **X**, or names of functions declared as friends of **X**, are not members of **X**; their names simply have enclosing scope.

**Hiding** A name can be hidden by an explicit declaration of the same name in an enclosed block or in a class. A hidden class member is still accessible using the scope modifier with a class name: **X::M**. A hidden file scope (global) name can be referenced with the unary operator **::**; for example, **::g**. A class name **X** can be hidden by the name of an object, function, or enumerator declared within the scope of **X**, regardless of the order in which the names are declared. However, the hidden class name **X** can still be accessed by prefixing **X** with the appropriate keyword: **class**, **struct**, or **union**.

The point of declaration for a name *x* is immediately after its complete declaration but before its initializer, if one exists.

**C++ scoping rules summary** The following rules apply to all names, including **typedef** names and class names, provided that C++ allows such names in the particular context discussed:

1. The name itself is tested for ambiguity. If no ambiguities are detected within its scope, the access sequence is initiated.
2. If no access control errors occur, the type of the object, function, class, **typedef**, and so on, is tested.
3. If the name is used outside any function and class, or is prefixed by the unary scope access operator **::**, and if the name is not qualified by the binary **::** operator or the member selection operators **.** and **->**, then the name must be a global object, function, or enumerator.
4. If the name *n* appears in any of the forms **X::n**, *x.n* (where *x* is an object of **X** or a reference to **X**), or *ptr->n* (where *ptr* is a pointer to **X**), then *n* is the name of a member of **X** or the member of a class from which **X** is derived.
5. Any name not covered so far that is used in a static member function must be declared in the block in which it occurs or in an enclosing block, or be a global name. The declaration of a local name *n* hides declarations of *n* in enclosing blocks and global declarations of *n*. Names in different scopes are not overloaded.

6. Any name not covered so far that is used in a nonstatic member function of class **X** must be declared in the block in which it occurs or in an enclosing block, be a member of class **X** or a base class of **X**, or be a global name. The declaration of a local name *n* hides declarations of *n* in enclosing blocks, members of the function's class, and global declarations of *n*. The declaration of a member name hides declarations of the same name in base classes.
7. The name of a function argument in a function definition is in the scope of the outermost block of the function. The name of a function argument in a non-defining function declaration has no scope at all. The scope of a default argument is determined by the point of declaration of its argument, but it can't access local variables or nonstatic class members. Default arguments are evaluated at each point of call.
8. A constructor initializer (see *ctor-initializer* in the class declarator syntax, Table 1.12 on page 37) is evaluated in the scope of the outermost block of its constructor, so it can refer to the constructor's argument names.

## Turbo C++ preprocessor directives

---

Although Turbo C++ uses an integrated single-pass compiler for both its IDE and command-line-compiler versions, it is useful to retain the terminology associated with earlier multipass compilers. In the latter, a first pass of the source text would pull in any include files, test for any conditional-compilation directives, expand any macros, and produce an intermediate file for further compiler passes. Since both the IDE and command-line-compiler versions of Turbo C++ perform this first pass with no intermediate output, Turbo C++ provides an independent preprocessor, CPP.EXE, that does produce such an output file. CPP is useful as a debugging aid, letting you see the net result of include directives, conditional compilation directives, and complex macro expansions.

*CPP is documented online.*

The following discussion on preprocessor directives, their syntax and semantics, therefore, applies both to the CPP preprocessor and to the preprocessor functionality built into the Turbo C++ compilers.

*The preprocessor detects preprocessor directives (also known as control lines) and parses the tokens embedded in them.*

The Turbo C++ preprocessor includes a sophisticated macro processor that scans your source code before the compiler itself gets to work. The preprocessor gives you great power and flexibility in the following areas:

- Defining macros that reduce programming effort and improve your source code legibility. Some macros can also eliminate the overhead of function calls.
- Including text from other files, such as header files containing standard library and user-supplied function prototypes and manifest constants.
- Setting up conditional compilations for improved portability and for debugging sessions.

*Preprocessor directives are usually placed at the beginning of your source code, but they can legally appear at any point in a program.*

Any line with a leading # is taken as a preprocessing directive, unless the # is within a string literal, in a character constant, or embedded in a comment. The initial # can be preceded or followed by whitespace (excluding new lines).

The full syntax for Turbo C++'s preprocessor directives is given in the next table.

Table 1.23: Turbo C++ preprocessing directives syntax

<i>preprocessing-file:</i> group	# <i>newline</i>
<i>group:</i> group-part group group-part	<i>action:</i> one of + - .
<i>group-part:</i> <pp-tokens> <i>newline</i> <i>if-section</i> <i>control-line</i>	<i>abbreviation:</i> <i>abbreviation:</i> amb amp apt aus big cln cpt def dup eff mod par pia pro rch ret rng rpt rvl sig str stu stv sus ucp use voi zst
<i>if-section:</i> <i>if-group</i> < <i>elif-groups</i> > < <i>else-group</i> > <i>endif-line</i>	<i>lparen:</i> the left parenthesis character without preceding whitespace
<i>if-group:</i> #if <i>constant-expression</i> <i>newline</i> < <i>group</i> > #ifdef <i>identifier</i> <i>newline</i> < <i>group</i> > #ifndef <i>identifier</i> <i>newline</i> < <i>group</i> >	<i>replacement-list:</i> <pp-tokens>
<i>elif-groups:</i> elif-group elif-groups <i>elif-group</i>	<i>pp-tokens:</i> <i>preprocessing-token</i> <i>pp-tokens</i> <i>preprocessing-token</i>
<i>elif-group:</i> #elif <i>constant-expression</i> <i>newline</i> < <i>group</i> >	<i>preprocessing-token:</i> <i>header-name</i> (only within an #include directive) <i>identifier</i> (no keyword distinction) <i>constant</i> <i>string-literal</i> <i>operator</i> <i>punctuator</i> each non-whitespace character that cannot be one of the preceding
<i>else-group:</i> #else <i>newline</i> < <i>group</i> >	<i>header-name:</i> < <i>h-char-sequence</i> >
<i>endif-line:</i> #endif <i>newline</i>	<i>h-char-sequence:</i> <i>h-char</i> <i>h-char-sequence</i> <i>h-char</i>
<i>control-line:</i> #include <i>pp-tokens</i> <i>newline</i> #define <i>identifier</i> <i>replacement-list</i> <i>newline</i> #define <i>identifier</i> <i>lparen</i> < <i>identifier-list</i> > <i>replacement-list</i> <i>newline</i> #undef <i>identifier</i> <i>newline</i> #line <i>pp-tokens</i> <i>newline</i> #error < <i>pp-tokens</i> > <i>newline</i> #pragma < <i>pp-tokens</i> > <i>newline</i> #pragma warn <i>action</i> <i>abbreviation</i> <i>newline</i> #pragma inline <i>newline</i>	<i>h-char:</i> any character in the source character set except the newline (\n) or greater than (>) character
	<i>newline:</i> the <i>newline</i> character

## Null directive #

---

The null directive consists of a line containing the single character #. This directive is always ignored.

## The #define and #undef directives

---

The **#define** directive defines a *macro*. Macros provide a mechanism for token replacement with or without a set of formal, function-like parameters.

### Simple #define macros

In the simple case with no parameters, the syntax is as follows:

```
#define macro_identifier <token_sequence>
```

Each occurrence of *macro\_identifier* in your source code following this control line will be replaced *in situ* with the possibly empty *token\_sequence* (there are some exceptions, which are noted later). Such replacements are known as *macro expansions*. The token sequence is sometimes called the *body* of the macro.

Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code are not expanded.

An empty token sequence results in the effective removal of each affected macro identifier from the source code:

```
#define HI "Have a nice day!"
#define empty
#define NIL ""
...
puts(HI);          /* expands to puts("Have a nice day!"); */
puts(NIL);         /* expands to puts(""); */
puts("empty");     /* NO expansion of empty! */
/* NOR any expansion of the empty within comments! */
```

After each individual macro expansion, a further scan is made of the newly expanded text. This allows for the possibility of *nested macros*: The expanded text may contain macro identifiers that are subject to replacement. However, if the macro expands into what looks like a preprocessing directive, such a directive will not be recognized by the preprocessor:

```
#define GETSTD #include <stdio.h>
...
GETSTD    /* compiler error */
```

GETSTD will expand to `#include <stdio.h>`. However, the preprocessor itself will not obey this apparently legal directive, but will pass it verbatim to the compiler. The compiler will reject `#include <stdio.h>` as illegal input. A macro won't be expanded during its own expansion. So `#define A A` won't expand indefinitely.

The `#undef` directive    You can undefine a macro using the `#undef` directive:

```
#undef macro_identifier
```

This line detaches any previous token sequence from the macro identifier; the macro definition has been forgotten, and the macro identifier is undefined.

No macro expansion occurs within `#undef` lines.

The state of being *defined* or *undefined* turns out to be an important property of an identifier, regardless of the actual definition. The `#ifdef` and `#ifndef` conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier has been undefined, it can be redefined with `#define`, using the same or a different token sequence.

```
#define BLOCK_SIZE 512
...
buff = BLOCK_SIZE*blks; /* expands as 512*blks *
...
#undef BLOCK_SIZE
/* use of BLOCK_SIZE now would be illegal "unknown" identifier */
...
#define BLOCK_SIZE 128 /* redefinition */
...
buf = BLOCK_SIZE*blks; /* expands as 128*blks */
...
```

Attempting to redefine an already defined macro identifier will result in a warning unless the new definition is *exactly* the same, token-by-token definition as the existing one. The preferred strategy where definitions may exist in other header files is as follows:

```
#ifndef BLOCK_SIZE
    #define BLOCK_SIZE 512
#endif
```

The middle line is bypassed if `BLOCK_SIZE` is currently defined; if `BLOCK_SIZE` is not currently defined, the middle line is invoked to define it.

Note that no semicolon (;) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in the macro expansion. The token sequence terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single space character.

Assembly language programmers must resist the temptation to write:

```
#define BLOCK_SIZE = 512 /* ?? token sequence includes the = */
```

The -D and -U options

Identifiers can be defined and undefined using the command-line compiler options **-D** and **-U** (see Chapter 4, “The command-line compiler,” in the *User’s Guide*). Identifiers can be defined, but not explicitly undefined, from the IDE **Options | Compiler | Defines** menu (see Chapter 1, “The IDE reference,” also in the *User’s Guide*).

The command line

```
tcc -Ddebug=1; paradox=0; X -Umysym myprog.c
```

is equivalent to placing

```
#define debug 1
#define paradox 0
#define X
#undef mysym
```

in the program.

Keywords and protected words

It is legal but ill-advised to use Turbo C++ keywords as macro identifiers:

```
#define int long /* legal but probably catastrophic */
#define INT long /* legal and possibly useful */
```

The following predefined global identifiers may *not* appear immediately following a **#define** or **#undef** directive:

*Note the double underscores, leading and trailing.*

```
__STDC__
__FILE__
__LINE__
```

```
__DATE__
__TIME__
```

Macros with parameters

The following syntax is used to define a macro with parameters:

**#define** *macro\_identifier*(*<arg\_list>*) *token\_sequence*

Any comma within parentheses in an argument list is treated as part of the argument, not as an argument delimiter.

Note that there can be no whitespace between the macro identifier and the (. The optional *arg\_list* is a sequence of identifiers separated by commas, not unlike the argument list of a C function. Each comma-delimited identifier plays the role of a *formal argument* or *place holder*.

Such macros are called by writing

```
macro_identifier<whitespace>(<actual_arg_list>)
```

in the subsequent source code. The syntax is identical to that of a function call; indeed, many standard library C “functions” are implemented as macros. However, there are some important semantic differences and potential pitfalls (see page 140).

The optional *actual\_arg\_list* must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the formal *arg\_list* of the **#define** line: There must be an actual argument for each formal argument. An error will be reported if the number of arguments in the two lists is different.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in the *actual\_arg\_list*. For example,

```
#define CUBE(x) ((x)*(x)*(x))
...
int n,y;
n = CUBE(y);
```

results in the following replacement:

```
n = ((y) * (y) * (y));
```

Similarly, the last line of

```
#define SUM ((a) + (b))
...
int i,j,sum;
sum = SUM(i,j);
```

expands to *sum = ((i) + (j))*. The reason for the apparent glut of parentheses will be clear if you consider the call

```
n = CUBE(y+1);
```

Without the inner parentheses in the definition, this would expand as  $n = y+1*y+1 *y+1$ , which is parsed as

```
n = y + (1*y) + (1*y) + 1; // != (y+1) cubed unless y=0 or y = -3!
```

As with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

Note the following points when using macros with argument lists:

1. **Nested parentheses and commas:** The *actual\_arg\_list* may contain nested parentheses provided that they are balanced; also, commas appearing within quotes or parentheses are not treated like argument delimiters:

```
#define ERRMSG(x, str) showerr("Error",x,str)
#define SUM(x,y) ((x) + (y))
...
ERRMSG(2, "Press Enter, then Esc");
/* expands to showerr("Error",2,"Press Enter, then Esc");
return SUM(f(i,j), g(k,l));
/* expands to return ((f(i,j)) + (g(k,l))); */
```

2. **Token pasting with ##:** You can paste (or merge) two tokens together by separating them with ## (plus optional whitespace on either side). The preprocessor removes the whitespace and the ##, combining the separate tokens into one new token. You can use this to construct identifiers; for example, given the definition

```
#define VAR(i,j) (i##j)
```

then the call `VAR(x,6)` would expand to `(x6)`. This replaces the older (nonportable) method of using `i/**/j`.

3. **Converting to strings with #:** The # symbol can be placed in front of a formal macro argument in order to convert the actual argument to a string after replacement. So, given the following macro definition:

```
#define TRACE(flag) printf(#flag "%d\n",flag)
```

the code fragment

```
int highval = 1024;
TRACE(highval);
```

becomes

```
int highval = 1024;
printf("highval" "= %d\n", highval);
```

which, in turn, is treated as

```
int highval = 1024;
printf("highval=%d\n", highval);
```

4. **The backslash for line continuation:** A long token sequence can straddle a line by using a backslash (\). The backslash and the following newline are both stripped to provide the actual token sequence used in expansions:

```
#define WARN "This is really a single-\  
line warning"  
...  
puts(WARN);  
/* screen will show: This is really a single-line warning */
```

5. **Side effects and other dangers:** The similarities between function and macro calls often obscure their differences. A macro call has no built-in type checking, so a mismatch between formal and actual argument data types can produce bizarre, hard-to-debug results with no immediate warning. Macro calls can also give rise to unwanted side effects, especially when an actual argument is evaluated more than once. Compare **CUBE** and **cube** in the following example:

```
int cube(int x) {  
    return x*x*x;  
}  
#define CUBE(x) ((x)*(x)*(x))  
...  
int b = 0, a = 3;  
b = cube(a++);  
/* cube() is passed actual arg = 3; so b = 27; a now = 4 */  
a = 3;  
b = CUBE(a++);  
/* expands as ((a++)*(a++)*(a++)); a now = 6 */
```

*Final value of b depends on what your compiler does to the expanded expression.*

## File inclusion with `#include`

*The angle brackets are real tokens, not metasyms that imply that header\_name is optional.*

---

The **#include** directive pulls in other named files, known as *include files*, *header files*, or *headers*, into the source code. The syntax has three forms:

```
#include <header_name>  
#include "header_name"  
#include macro_identifier
```

The third variant assumes that neither < nor " appears as the first non-whitespace character following **#include**; further, it assumes that a macro definition exists that will expand the macro identifier into a valid delimited header name with either of the <header\_name> or "header\_name" formats.

The first and second variant imply that no macro expansion will be attempted; in other words, *header\_name* is never scanned for macro identifiers. *header\_name* must be a valid DOS file name with an extension (traditionally .h for header) and optional path name and path delimiters.

The preprocessor removes the **#include** line and conceptually replaces it with the entire text of the header file at that point in the source code. The source code itself is not changed, but the compiler “sees” the enlarged text. The placement of the **#include** may therefore influence the scope and duration of any identifiers in the included file.

If you place an explicit path in the *header\_name*, only that directory will be searched.

The difference between the *<header\_name>* and “*header\_name*” formats lies in the searching algorithm employed in trying to locate the include file; these algorithms are described in the following two sections.

Header file search with  
*<header\_name>*

The *<header\_name>* variant specifies a standard include file; the search is made successively in each of the include directories in the order they are defined. If the file is not located in any of the default directories, an error message is issued.

Header file search with  
“*header\_name*”

The “*header\_name*” variant specifies a user-supplied include file; the file is sought first in the current directory (usually the directory holding the source file being compiled). If the file is not found there, the search continues in the include directories as in the *<header\_name>* situation.

The following example clarifies these differences:

```
#include <stdio.h>
/* header in standard include directory */

#define myinclude "c:\tc\include\mystuff.h"
/* Note: Single backslashes OK here; within a C statement you would
   need "c:\tc\include\mystuff.h" */

#include myinclude
/* macro expansion */

#include "myinclude.h"
/* no macro expansion */
```

After expansion, the second **#include** statement causes the preprocessor to look in C:\TC\INCLUDE\mstuff.h and nowhere else. The

third **#include** causes it to look for `myinclude.h` in the current directory, then in the default directories.

## Conditional compilation

Turbo C++ supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those beginning with `#` (except the **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, and **#endif** directives), as well as any lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

The **#if**, **#elif**, **#else**, and **#endif** conditional directives

The conditional directives **#if**, **#elif**, **#else**, and **#endif** work like the normal C conditional operators. They are used as follows:

```
#if constant-expression-1
<section-1>
<#elif constant-expression-2 newline section-2>
...
<#elif constant-expression-n newline section-n>
<#else final-section>
#endif
...
```

If the *constant-expression-1* (subject to macro expansion) evaluates to nonzero (true), the lines of code (possibly empty) represented by *section-1*, whether preprocessor command lines or normal source lines, are preprocessed and, as appropriate, passed to the Turbo C++ compiler. Otherwise, if *constant-expression-1* evaluates to zero (false), *section-1* is ignored (no macro expansion and no compilation).

In the *true* case, after *section-1* has been preprocessed, control passes to the matching **#endif** (which ends this conditional interlude) and continues with *next-section*. In the *false* case, control passes to the next **#elif** line (if any) where *constant-expression-2* is evaluated. If true, *section-2* is processed, after which control moves on to the matching **#endif**. Otherwise, if *constant-expression-2* is false, control passes to the next **#elif**, and so on, until either **#else** or **#endif** is reached. The optional **#else** is used as an alternative condition for which all previous tests have proved false. The **#endif** ends the conditional sequence.

The processed section can contain further conditional clauses, nested to any depth; each **#if** must be carefully balanced with a closing **#endif**.

The net result of the above scenario is that only one section (possibly empty) is passed on for further processing. The bypassed sections are relevant only for keeping track of any nested conditionals, so that each **#if** can be matched with its correct **#endif**.

The constant expressions to be tested must evaluate to a constant integral value.

### The operator **defined**

The **defined** operator offers an alternative, more flexible way of testing whether combinations of identifiers are defined or not. It is valid only in **#if** and **#elif** expressions.

The expression **defined**(*identifier*) or **defined** *identifier* (parentheses are optional) evaluates to 1 (true) if the symbol has been previously defined (using **#define**) and has not been subsequently undefined (using **#undef**); otherwise, it evaluates to 0 (false). So the directive

```
#if defined(mysym)
```

is the same as

```
#ifdef mysym
```

The advantage is that you can use **defined** repeatedly in a complex expression following the **#if** directive, such as

```
#if defined(mysym) && !defined(yoursym)
```

The **#ifdef** and **#ifndef** conditional directives

The **#ifdef** and **#ifndef** conditional directives let you test whether an identifier is currently defined or not, that is, whether a previous **#define** command has been processed for that identifier and is still in force. The line

```
#ifdef identifier
```

has exactly the same effect as

```
#if 1
```

if *identifier* is currently defined, and the same effect as

```
#if 0
```

if *identifier* is currently undefined.

**#ifndef** tests true for the “not-defined” condition, so the line

```
#ifndef identifier
```

has exactly the same effect as

```
#if 0
```

if *identifier* is currently defined, and the same effect as

```
#if 1
```

if *identifier* is currently undefined.

The syntax thereafter follows that of the **#if**, **#elif**, **#else**, and **#endif** given in the previous section.

An identifier defined as NULL is considered to be defined.

---

## The #line line control directive

You can use the **#line** command to supply line numbers to a program for cross-reference and error reporting. If your program consists of sections derived from some other program file, it is often useful to mark such sections with the line numbers of the original source rather than the normal sequential line numbers derived from the composite program. The syntax is

```
#line integer_constant <"filename">
```

indicating that the following source line originally came from line number *integer\_constant* of *filename*. Once the *filename* has been registered, subsequent **#line** commands relating to that file can omit the explicit *filename* argument. For example,

*The inclusion of stdio.h means that the preprocessor output will be somewhat large.*

```
/* TEMP.C: An example of the #line directive */
#include <stdio.h>
#line 4 "junk.c"
void main()
{
    printf(" in line %d of %s", __LINE__, __FILE__);
#line 12 "temp.c"
    printf("\n");
    printf(" in line %d of %s", __LINE__, __FILE__);
#line 8
    printf("\n");
    printf(" in line %d of %s", __LINE__, __FILE__);
}
```

If you run TEMP.C through CPP (`cpp temp`), you'll get an output file TEMP.I; it should look like this:

```
temp.c 1:
c:\borland\tc\cpp\include\stdio.h 1:
c:\borland\tc\cpp\include\stdio.h 2:
```

We've eliminated most of the *stdio.h* portion.

```
c:\borland\tc\cpp\include\stdio.h 3:
...
c:\borland\tc\cpp\include\stdio.h 212:
c:\borland\tc\cpp\include\stdio.h 213:
temp.c 2:
temp.c 3:
junk.c 4: void main()
junk.c 5: {
junk.c 6: printf(" in line %d of %s",6,"junk.c");
junk.c 7:
temp.c 12: printf("\n");
temp.c 13: printf(" in line %d of %s",13,"temp.c");
temp.c 14:
temp.c 8: printf("\n");
temp.c 9: printf(" in line %d of %s",9,"temp.c");
temp.c 10: }
temp.c 11:
```

If you then compile TEMP.C, you'll get the output shown here:

```
in line 6 of junk.c
in line 13 of temp.c
in line 9 of temp.c
```

Macros are expanded in **#line** arguments as they are in the **#include** directive.

The **#line** directive is primarily used by utilities that produce C code as output, and not in human-written code.

---

## The **#error** directive

The **#error** directive has the following syntax:

```
#error errmsg
```

This generates the message:

```
Error: filename line# : Error directive: errmsg
```

This directive is usually embedded in a preprocessor conditional that catches some undesired compile-time condition. In the normal case, that condition will be false. If the condition is true, you want the compiler to print an error message and stop the compile. You do this by putting an **#error** directive within a conditional that is true for the undesired case.

For example, suppose you **#define** MYVAL, which must be either 0 or 1. You could then include the following conditional in your source code to test for an incorrect value of MYVAL:

```
#if (MYVAL != 0 && MYVAL != 1)
#error MYVAL must be defined to either 0 or 1
#endif
```

## The #pragma directive

The **#pragma** directive permits implementation-specific directives of the form:

**#pragma** *directive-name*

With **#pragma**, Turbo C++ can define whatever directives it desires without interfering with other compilers that support **#pragma**. If the compiler doesn't recognize *directive-name*, it ignores the **#pragma** directive without any error or warning message.

Turbo C++ supports the following **#pragma** directives:

- #pragma argsused
- #pragma exit
- #pragma inline
- #pragma option
- #pragma saveregs
- #pragma startup
- #pragma warn

### #pragma argsused

The **argsused** pragma is only allowed between function definitions, and it affects only the next function. It disables the warning message:

```
"Parameter name is never used in function func-name"
```

### #pragma exit and #pragma startup

These two pragmas allow the program to specify function(s) that should be called either upon program startup (before **main** is called), or program exit (just before the program terminates through **\_exit**).

The syntax is as follows:

```
#pragma exit function-name <priority>
#pragma startup function-name <priority>
```

The specified *function-name* must be a previously declared function taking no arguments and returning **void**; in other words, it should be declared as

```
void func(void);
```

The optional *priority* parameter should be an integer in the range 64 to 255. The highest priority is 0. (Priorities from 0 to 63 are used by the C libraries, and should not be used by the user.) Functions with higher priorities are called first at startup and last at exit. If you don't specify a priority, it defaults to 100. For example,

```
#include <stdio.h>

void startFunc(void)
{
    printf("Startup function.\n");
}

#pragma startup startFunc 64
/* priority 64 --> called first at startup */

void exitFunc(void)
{
    printf("Wrapping up execution.\n");
}

#pragma exit exitFunc
/* default priority is 100 */

void main(void)
{
    printf("This is main.\n");
}
```

Note that the function name used in **pragma startup** or **exit** must be defined (or declared) before the pragma line is reached.

**#pragma inline** This directive is equivalent to the **-B** command-line compiler option or the integrated environment option. It tells the compiler that there is inline assembly language code in your program (see Chapter 6, "Interfacing with assembly language"). The syntax is

#### **#pragma inline**

This is best placed at the top of the file, since the compiler restarts itself with the **-B** option when it encounters **#pragma inline**. Actually, you can leave off both the **-B** option and the **#pragma inline** directive, and the compiler will restart itself anyway as soon as it encounters **asm** statements. The purpose of the option and the directive is to save some compilation time.

**#pragma option** Use **#pragma option** to include command-line options within your program code. The syntax is

**#pragma option** [*options...*]

*options* can be any command-line option (except those listed in the following paragraph). Any number of options can appear in one directive.

Options that cannot appear in a **pragma option** include:

- **-B** (compile using assembly)
- **-c** (compile, but don't link)
- **-dxxx** (define a macro)
- **-Dxxx = ccc** (define a macro with text)
- **-efff** (name .EXE file *fff*)
- **-lfff** (name include directory)
- **-Lfff** (name library directory)
- **-lxset** (linker option *x*)
- **-M** (create a .MAP file in link)
- **-o** overlays
- **-Q** EMS
- **-S** (create .ASM output and stop)
- **-Uxxx** (undefine a macro)
- **-V** (virtual)
- **-Y** (overlays)

The compile proceeds in two states. You can include more options in a **#pragma option** during the first state than during the second state. The first state is called parsing-only; the second is the coding state.

Using any macro name that begins with two underscores (and is therefore a possible built-in macro) in an **#if**, **#ifdef**, **#ifndef** or **#elif** directive changes the compiler to coding state.

The occurrence of the first real token (the first C declaration) also changes the state to coding.

In other words, you can use **#pragmas**, **#includes**, **#define**, and some **#ifs** during the parsing-only state. During this phase, you can use **#pragma option** to change the command-line options.

Options which can appear in **#pragma options** only during the parsing-only state include:

- **-Efff** (assembler name string)
- **-f\*** (any floating-point option except **-ff**)
- **-i#** (significant identifier chars)
- **-m\*** (any memory model option)
- **-nddd** (output directory)
- **-offf** (output file name *fff*)
- **-u** (use underbars on **cdecl** names)
- **-z\*** (any segment name option)

Other options can be changed anywhere. The following options will only affect the compiler if they get changed between functions or object declarations:

- 1** Instruction set control.
- 2** Instruction set control.
- a** Alignment control. (Note that alignment of structure members is determined at the point of the structure definition, not when later objects use the structure.)
- ff** Fast floating-point control.
- G** Generate code for speed.
- k** Standard stack frame control.
- N** Stack checking control.
- O** Optimization control.
- p** Pascal calling convention default.
- r** and **-rd** Register variable control.
- v** Verbose debugging control.
- y** Line information control.

The following options can be changed at any time and take effect immediately:

- A** Keyword control.
- C** Nested comment control.
- d** Merge duplicate strings.
- gn** Stop after *n* warnings.
- jn** Stop after *n* errors.
- K** **char** type is unsigned.
- wxxx** Warning (same as **#pragma warn**).

Any of the toggle options (such as **-a** or **-K**) can be turned on and off as on the command line. They can additionally appear followed by a dot (.) to reset the option to its command-line state.

`#pragma saveregs` The **saveregs** pragma guarantees that a **huge** function will not change the value of any of the registers when it is entered. This directive is sometimes needed for interfacing with assembly language code. The directive should be placed immediately before the function definition. It applies to that function alone.

`#pragma warn` The **warn** directive lets you override specific **-wxxx** command-line options (or check **Display Warnings** in the **Options | Compiler | Messages** dialog box).

For example, if your source code contains the directives

```
#pragma warn +xxx
#pragma warn -yyy
#pragma warn .zzz
```

the *xxx* warning will be turned on (even if on the **Options | Compiler | Messages** menu it was toggled to *Off*), the *yyy* warning will be turned off, and the *zzz* warning will be restored to the value it had when compilation of the file began.

A complete list of the three-letter abbreviations and the warnings to which they apply is given in Chapter 4, "The command-line compiler," in the *User's Guide*.

---

## Predefined macros

Turbo C++ predefines the following global identifiers. Except for `__cplusplus`, each of these starts and ends with two underscore characters (`__`). These macros are also known as *manifest constants*.

`__CDECL__` This macro is specific to Turbo C++. It signals that the **-p** flag was not used (**Calling Convention...C**): Set to the integer constant 1 if **-p** was not used; otherwise, undefined.

The following six symbols are defined based on the memory model chosen at compile time.

```
__COMPACT__      __MEDIUM__
__HUGE__         __SMALL__
__LARGE__        __TINY__
```

Only one is defined for any given compilation; the others, by definition, are undefined. For example, if you compile with the small

model, the `__SMALL__` macro is defined and the rest are not, so that the directive

```
#if defined( __SMALL__ )
```

will be true, while

```
#if defined( __HUGE__ )
```

(or any of the others) will be false. The actual value for any of these defined macros is 1.

- `__cplusplus` This macro is specific to Turbo C++. This allows you to write a module that will be compiled sometimes as C and sometimes as C++. Using conditional compilation, you can control which C and C++ parts are included.
  
- `__DATE__` This macro provides the date the preprocessor began processing the current source file (as a string literal).  
  
Each inclusion of `__DATE__` in a given file contains the same value, regardless of how long the processing takes. The date appears in the format *mmm dd yyyy*, where *mmm* equals the month (Jan, Feb, and so forth), *dd* equals the day (1 to 31, with the first character of *dd* a blank if the value is less than 10), and *yyyy* equals the year (1990, 1991, and so forth).
  
- `__FILE__` This macro provides the name of the current source file being processed (as a string literal). This macro changes whenever the compiler processes an **#include** directive or a **#line** directive, or when the include file is complete.
  
- `__LINE__` This macro provides the number of the current source-file line being processed (as a decimal constant). Normally, the first line of a source file is defined to be 1, through the **#line** directive can affect this. See page 144 for information on the **#line** directive.
  
- `__MSDOS__` This macro is specific to Turbo C++. It provides the integer constant 1 for all compilations.

- `__OVERLAY__` This macro is specific to Turbo C++. It is predefined to be 1 if you compile a module with the `-Y` option (enable overlay support). If you don't enable overlay support, this macro is undefined.
- `__PASCAL__` This macro is specific to Turbo C++. It signals that the `-p` flag has been used. The macro is set to the integer constant 1 if the `-p` flag is used; otherwise, it remains undefined.
- `__STDC__` This macro is defined as the constant 1 if you compile with the ANSI compatibility flag (`-A` or `ANSI Keywords Only...On`); otherwise, the macro is undefined.
- `__TIME__` This macro keeps track of the time the preprocessor began processing the current source file (as a string literal).  
As with `__DATE__`, each inclusion of `__TIME__` contains the same value, regardless of how long the processing takes. It takes the format `hh:mm:ss`, where *hh* equals the hour (00 to 23), *mm* equals minutes (00 to 59), and *ss* equals seconds (00 to 59).
- `__TURBOC__` This macro is specific to Turbo C++. It gives the current Turbo C++ version number, a hexadecimal constant. For example, version 1.0 is `0x0100`.

## *Run-time library cross-reference*

This chapter is an overview of the Turbo C++ library routines and include files.

In this chapter, we

- explain why you might want to obtain the source code for the Turbo C++ run-time library
- list and describe the header files
- summarize the different categories of tasks performed by the library routines

Turbo C++ comes equipped with over 450 functions and macros that you call from within your C programs to perform a wide variety of tasks, including low- and high-level I/O, string and file manipulation, memory allocation, process control, data conversion, mathematical calculations, and much more. These functions and macros, called library routines, are individually documented in the *Library Reference*.

Turbo C++'s routines are contained in the library files (Cx.LIB, CPx.LIB, MATHx.LIB, and GRAPHICS.LIB). Because Turbo C++ supports six distinct memory models, each model except the tiny model has its own library file and math file, containing versions of the routines written for that particular model. (The tiny model shares the small model's library and math files.)

*In C++, you **must** always use prototypes. See page 60 for more information on function prototypes.*

Turbo C++ implements the latest ANSI C standard which, among other things, allows (and strongly recommends) function prototypes to be given for the routines in your C programs. All of

Turbo C++'s library routines are declared with prototypes in one or more header files.

## Reasons to access the run-time library source code

---

The Turbo C++ run-time library contains over 450 functions, covering a broad range of areas: low-level control of your IBM PC, interfacing with DOS, input/output, process management, string and memory manipulations, math, sorting and searching, and so on. There are several good reasons why you may wish to obtain the source code for these functions:

- You may find that a particular Turbo C++ function you want to write is similar to, but not the same as, a function in the library. With access to the run-time library source code, you can tailor the library function to your own needs, and avoid having to write a separate function of your own.
- Sometimes, when you are debugging code, you may wish to know more about the internals of a library function. Having the source code to the run-time library would be of great help in this situation.
- When you can't figure out what a library function is really supposed to do, it's useful to be able to take a quick look at that function's source code.
- You may want to eliminate leading underscores on C symbols. Access to the run-time library source code will let you eliminate them.
- You can learn a lot from studying tight, professionally written library source code.

For all these reasons, and more, you will want to have access to the Turbo C++ run-time library source code. Because Borland believes strongly in the concept of "open architecture," we have made the Turbo C++ run-time library source code available for licensing. All you have to do is fill out the order form distributed with your Turbo C++ package, include your payment, and we'll ship you the Turbo C++ run-time library source code.

# The Turbo C++ header files

---

*Header files defined by ANSI C are marked as such with a comment in the margin. C++ header files are also marked in the margin.*

Header files, also called include files, provide function prototype declarations for library functions. Data types and symbolic constants used with the library functions are also defined in them, along with global variables defined by Turbo C++ and by the library functions. The Turbo C++ library follows the ANSI C standard on names of header files and their contents.

	alloc.h	Declares memory management functions (allocation, deallocation, etc.).
ANSI C	assert.h	Defines the <b>assert</b> debugging macro.
	bcd.h	Declares the C++ class <b>bcd</b> and the overloaded operators for <b>bcd</b> and <b>bcd</b> math functions.
	bios.h	Declares various functions used in calling IBM-PC ROM BIOS routines.
	complex.h	Declares the C++ complex math functions.
	conio.h	Declares various functions used in calling the DOS console I/O routines.
ANSI C	ctype.h	Contains information used by the character classification and character conversion macros (such as <b>isalpha</b> and <b>toascii</b> ).
	dir.h	Contains structures, macros, and functions for working with directories and path names.
	dos.h	Defines various constants and gives declarations needed for DOS and 8086-specific calls.
ANSI C	errno.h	Defines constant mnemonics for the error codes.
	fcntl.h	Defines symbolic constants used in connection with the library routine <b>open</b> .
ANSI C	float.h	Contains parameters for floating-point routines.
	fstream.h	Declares the C++ stream classes that support file input and output.
	generic.h	Contains macros for generic class declarations.
	graphics.h	Declares prototypes for the graphics functions.
	io.h	Contains structures and declarations for low-level input/output routines.

	<code>iomanip.h</code>	Declares the C++ streams I/O manipulators and contains macros for creating parameterized manipulators.
	<code>iostream.h</code>	Declares the basic C++ (version 2.0) streams (I/O) routines.
ANSI C	<code>limits.h</code>	Contains environmental parameters, information about compile-time limitations, and ranges of integral quantities.
ANSI C	<code>locale.h</code>	Declares functions that provide country- and language-specific information.
ANSI C	<code>math.h</code>	Declares prototypes for the math functions; also defines the macro <code>HUGE_VAL</code> , and declares the exception structure used by the <code>matherr</code> routine.
	<code>mem.h</code>	Declares the memory-manipulation functions. (Many of these are also defined in <code>string.h</code> .)
	<code>process.h</code>	Contains structures and declarations for the <b>spawn...</b> and <b>exec...</b> functions.
ANSI C	<code>setjmp.h</code>	Defines a type <code>jmp_buf</code> used by the <b>longjmp</b> and <b>setjmp</b> functions and declares the routines <b>longjmp</b> and <b>setjmp</b> .
	<code>share.h</code>	Defines parameters used in functions that make use of file-sharing.
ANSI C	<code>signal.h</code>	Defines constants and declarations for use by the <b>signal</b> and <b>raise</b> functions.
ANSI C	<code>stdarg.h</code>	Defines macros used for reading the argument list in functions declared to accept a variable number of arguments (such as <b>vprintf</b> , <b>vscanf</b> , etc.).
ANSI C	<code>stddef.h</code>	Defines several common data types and macros.
ANSI C	<code>stdio.h</code>	Defines types and macros needed for the Standard I/O Package defined in Kernighan and Ritchie and extended under UNIX System V. Defines the standard I/O predefined streams <i>stdin</i> , <i>stdout</i> , <i>stderr</i> , and <i>stdin</i> , and declares stream-level I/O routines.
	<code>stdiostr.h</code>	Declares the C++ stream classes for use with <code>stdio</code> FILE structures.

ANSI C	stdlib.h	Declares several commonly used routines: conversion routines, search/sort routines, and other miscellany.
	stream.h	Declares the C++ (version 1.2) streams (I/O) routines.
ANSI C	string.h	Declares several string-manipulation and memory-manipulation routines.
	strstrea.h	Declares the C++ stream classes for use with byte arrays in memory.
	sys\stat.h	Defines symbolic constants used for opening and creating files.
	sys\timeb.h	Declares the function <b>ftime</b> and the structure <b>timeb</b> that <b>ftime</b> returns.
	sys\types.h	Declares the type <i>time_t</i> used with time functions.
ANSI C	time.h	Defines a structure filled in by the time-conversion routines <b>asctime</b> , <b>localtime</b> , and <b>gmtime</b> , and a type used by the routines <b>ctime</b> , <b>difftime</b> , <b>gmtime</b> , <b>localtime</b> , and <b>stime</b> ; also provides prototypes for these routines.
	values.h	Defines important constants, including machine dependencies; provided for UNIX System V compatibility.

## Library routines by category

---

The Turbo C++ library routines perform a variety of tasks. In this section, we list the routines, along with the include files in which they are declared, under several general categories of task performed. For complete information about any of the functions below, see the function entry in Chapter 1, "The run-time library," in the *Library Reference*.

### Classification routines

These routines classify ASCII characters as letters, control characters, punctuation, uppercase, etc.

<b>isalnum</b>	(ctype.h)	<b>isascii</b>	(ctype.h)	<b>isdigit</b>	(ctype.h)
<b>isalpha</b>	(ctype.h)	<b>iscntrl</b>	(ctype.h)	<b>isgraph</b>	(ctype.h)

<b>islower</b>	(ctype.h)	<b>ispunct</b>	(ctype.h)	<b>isupper</b>	(ctype.h)
<b>isprint</b>	(ctype.h)	<b>isspace</b>	(ctype.h)	<b>isxdigit</b>	(ctype.h)

## Conversion routines

These routines convert characters and strings from alpha to different numeric representations (floating-point, integers, longs) and vice versa, and from uppercase to lowercase and vice versa.

<b>atof</b>	(stdlib.h)	<b>itoa</b>	(stdlib.h)	<b>_tolower</b>	(ctype.h)
<b>atoi</b>	(stdlib.h)	<b>ltoa</b>	(stdlib.h)	<b>tolower</b>	(ctype.h)
<b>atol</b>	(stdlib.h)	<b>strtod</b>	(stdlib.h)	<b>_toupper</b>	(ctype.h)
<b>ecvt</b>	(stdlib.h)	<b>strtol</b>	(stdlib.h)	<b>toupper</b>	(ctype.h)
<b>fcvt</b>	(stdlib.h)	<b>strtoul</b>	(stdlib.h)	<b>ultoa</b>	(stdlib.h)
<b>gcvt</b>	(stdlib.h)	<b>toascii</b>	(ctype.h)		

## Directory control routines

These routines manipulate directories and path names.

<b>chdir</b>	(dir.h)	<b>fnsplit</b>	(dir.h)	<b>mkdir</b>	(dir.h)
<b>findfirst</b>	(dir.h)	<b>getcurdir</b>	(dir.h)	<b>mktemp</b>	(dir.h)
<b>findnext</b>	(dir.h)	<b>getcwd</b>	(dir.h)	<b>rmdir</b>	(dir.h)
<b>fnmerge</b>	(dir.h)	<b>getdisk</b>	(dir.h)	<b>searchpath</b>	(dir.h)
				<b>setdisk</b>	(dir.h)

## Diagnostic routines

These routines provide built-in troubleshooting capability.

<b>assert</b>	(assert.h)
<b>matherr</b>	(math.h)
<b>perror</b>	(errno.h)

## Graphics routines

These routines let you create onscreen graphics with text.

<b>arc</b>	(graphics.h)	<b>fillellipse</b>	(graphics.h)
<b>bar</b>	(graphics.h)	<b>fillpoly</b>	(graphics.h)
<b>bar3d</b>	(graphics.h)	<b>floodfill</b>	(graphics.h)
<b>circle</b>	(graphics.h)	<b>getarccoords</b>	(graphics.h)
<b>cleardevice</b>	(graphics.h)	<b>getaspectratio</b>	(graphics.h)
<b>clearviewport</b>	(graphics.h)	<b>getbkcolor</b>	(graphics.h)
<b>closegraph</b>	(graphics.h)	<b>getcolor</b>	(graphics.h)
<b>detectgraph</b>	(graphics.h)	<b>getdefaultpalette</b>	(graphics.h)
<b>drawpoly</b>	(graphics.h)	<b>getdrivername</b>	(graphics.h)
<b>ellipse</b>	(graphics.h)	<b>getfillpattern</b>	(graphics.h)

<b>getfillsettings</b>	(graphics.h)	<b>outtext</b>	(graphics.h)
<b>getgraphmode</b>	(graphics.h)	<b>outtextxy</b>	(graphics.h)
<b>getimage</b>	(graphics.h)	<b>pieslice</b>	(graphics.h)
<b>getlinesettings</b>	(graphics.h)	<b>putimage</b>	(graphics.h)
<b>getmaxcolor</b>	(graphics.h)	<b>putpixel</b>	(graphics.h)
<b>getmaxmode</b>	(graphics.h)	<b>rectangle</b>	(graphics.h)
<b>getmaxx</b>	(graphics.h)	<b>registerbgidriver</b>	(graphics.h)
<b>getmaxy</b>	(graphics.h)	<b>registerbgifont</b>	(graphics.h)
<b>getmodename</b>	(graphics.h)	<b>restorecrtmode</b>	(graphics.h)
<b>getmoderange</b>	(graphics.h)	<b>sector</b>	(graphics.h)
<b>getpalette</b>	(graphics.h)	<b>setactivepage</b>	(graphics.h)
<b>getpalettesize</b>	(graphics.h)	<b>setallpalette</b>	(graphics.h)
<b>getpixel</b>	(graphics.h)	<b>setaspectratio</b>	(graphics.h)
<b>gettextsettings</b>	(graphics.h)	<b>setbkcolor</b>	(graphics.h)
<b>getviewsettings</b>	(graphics.h)	<b>setcolor</b>	(graphics.h)
<b>getx</b>	(graphics.h)	<b>setcursortype</b>	(conio.h)
<b>gety</b>	(graphics.h)	<b>setfillpattern</b>	(graphics.h)
<b>graphdefaults</b>	(graphics.h)	<b>setfillstyle</b>	(graphics.h)
<b>grapherrormsg</b>	(graphics.h)	<b>setgraphbufsize</b>	(graphics.h)
<b>_graphfreemem</b>	(graphics.h)	<b>setgraphmode</b>	(graphics.h)
<b>_graphgetmem</b>	(graphics.h)	<b>setlinestyle</b>	(graphics.h)
<b>graphresult</b>	(graphics.h)	<b>setpalette</b>	(graphics.h)
<b>imagesize</b>	(graphics.h)	<b>setrgbpalette</b>	(graphics.h)
<b>initgraph</b>	(graphics.h)	<b>settextjustify</b>	(graphics.h)
<b>installuserdriver</b>	(graphics.h)	<b>settextstyle</b>	(graphics.h)
<b>installuserfont</b>	(graphics.h)	<b>setusercharsize</b>	(graphics.h)
<b>line</b>	(graphics.h)	<b>setviewport</b>	(graphics.h)
<b>linerel</b>	(graphics.h)	<b>setvisualpage</b>	(graphics.h)
<b>lineto</b>	(graphics.h)	<b>setwritemode</b>	(graphics.h)
<b>moverel</b>	(graphics.h)	<b>textheight</b>	(graphics.h)
<b>moveto</b>	(graphics.h)	<b>textwidth</b>	(graphics.h)

## Input/output routines

These routines provide stream-level and DOS-level I/O capability.

<b>access</b>	(io.h)	<b>creatnew</b>	(io.h)
<b>cgets</b>	(conio.h)	<b>creattemp</b>	(io.h)
<b>_chmod</b>	(io.h)	<b>cscanf</b>	(conio.h)
<b>chmod</b>	(io.h)	<b>dup</b>	(io.h)
<b>chsize</b>	(io.h)	<b>dup2</b>	(io.h)
<b>clearerr</b>	(stdio.h)	<b>eof</b>	(io.h)
<b>_close</b>	(io.h)	<b>fclose</b>	(stdio.h)
<b>close</b>	(io.h)	<b>fcloseall</b>	(stdio.h)
<b>cprintf</b>	(conio.h)	<b>fdopen</b>	(stdio.h)
<b>cputs</b>	(conio.h)	<b>feof</b>	(stdio.h)
<b>_creat</b>	(io.h)	<b>ferror</b>	(stdio.h)
<b>creat</b>	(io.h)	<b>fflush</b>	(stdio.h)

<b>fgetc</b>	(stdio.h)	<b>printf</b>	(stdio.h)
<b>fgetchar</b>	(stdio.h)	<b>putc</b>	(stdio.h)
<b>fgetpos</b>	(stdio.h)	<b>putch</b>	(conio.h)
<b>fgets</b>	(stdio.h)	<b>putchar</b>	(stdio.h)
<b>filelength</b>	(io.h)	<b>puts</b>	(stdio.h)
<b>fileno</b>	(stdio.h)	<b>putw</b>	(stdio.h)
<b>flushall</b>	(stdio.h)	<b>_read</b>	(io.h)
<b>fopen</b>	(stdio.h)	<b>read</b>	(io.h)
<b>fprintf</b>	(stdio.h)	<b>remove</b>	(stdio.h)
<b>fputc</b>	(stdio.h)	<b>rename</b>	(stdio.h)
<b>fputchar</b>	(stdio.h)	<b>rewind</b>	(stdio.h)
<b>fputs</b>	(stdio.h)	<b>scanf</b>	(stdio.h)
<b>fread</b>	(stdio.h)	<b>setbuf</b>	(stdio.h)
<b>freopen</b>	(stdio.h)	<b>setcursortype</b>	(conio.h)
<b>fscanf</b>	(stdio.h)	<b>setftime</b>	(io.h)
<b>fseek</b>	(stdio.h)	<b>setmode</b>	(io.h)
<b>fsetpos</b>	(stdio.h)	<b>setvbuf</b>	(stdio.h)
<b>fstat</b>	(sys\stat.h)	<b>sopen</b>	(io.h)
<b>ftell</b>	(stdio.h)	<b>sprintf</b>	(stdio.h)
<b>fwrite</b>	(stdio.h)	<b>sscanf</b>	(stdio.h)
<b>getc</b>	(stdio.h)	<b>stat</b>	(sys\stat.h)
<b>getch</b>	(conio.h)	<b>_strerror</b>	(string.h, stdio.h)
<b>getchar</b>	(stdio.h)	<b>strerror</b>	(stdio.h)
<b>getche</b>	(conio.h)	<b>tell</b>	(io.h)
<b>getftime</b>	(io.h)	<b>tmpfile</b>	(stdio.h)
<b>getpass</b>	(conio.h)	<b>tmpnam</b>	(stdio.h)
<b>gets</b>	(stdio.h)	<b>ungetc</b>	(stdio.h)
<b>getw</b>	(stdio.h)	<b>ungetch</b>	(conio.h)
<b>ioctl</b>	(io.h)	<b>unlock</b>	(io.h)
<b>isatty</b>	(io.h)	<b>vfprintf</b>	(stdio.h)
<b>kbhit</b>	(conio.h)	<b>vfscanf</b>	(stdio.h)
<b>lock</b>	(io.h)	<b>vprintf</b>	(stdio.h)
<b>lseek</b>	(io.h)	<b>vscanf</b>	(stdio.h)
<b>_open</b>	(io.h)	<b>vsprintf</b>	(stdio.h)
<b>open</b>	(io.h)	<b>vsscanf</b>	(io.h)
<b>perror</b>	(stdio.h)	<b>_write</b>	(io.h)

## Interface routines (DOS, 8086, BIOS)

These routines provide DOS, BIOS and machine-specific capabilities.

<b>absread</b>	(dos.h)	<b>bioskey</b>	(bios.h)	<b>dosexterr</b>	(dos.h)
<b>abswrite</b>	(dos.h)	<b>biosmemory</b>	(bios.h)	<b>enable</b>	(dos.h)
<b>bdos</b>	(dos.h)	<b>biosprint</b>	(bios.h)	<b>FP_OFF</b>	(dos.h)
<b>bdosptr</b>	(dos.h)	<b>biostime</b>	(bios.h)	<b>FP_SEG</b>	(dos.h)
<b>bioscom</b>	(bios.h)	<b>country</b>	(dos.h)	<b>freemem</b>	(dos.h)
<b>biosdisk</b>	(bios.h)	<b>ctrlbrk</b>	(dos.h)	<b>geninterrupt</b>	(dos.h)
<b>biosequip</b>	(bios.h)	<b>disable</b>	(dos.h)	<b>getcbrk</b>	(dos.h)

<b>getdfree</b>	(dos.h)	<b>int86</b>	(dos.h)	<b>poke</b>	(dos.h)
<b>getdta</b>	(dos.h)	<b>int86x</b>	(dos.h)	<b>pokeb</b>	(dos.h)
<b>getfat</b>	(dos.h)	<b>intdos</b>	(dos.h)	<b>randbrd</b>	(dos.h)
<b>getfatd</b>	(dos.h)	<b>intdosx</b>	(dos.h)	<b>randbwr</b>	(dos.h)
<b>getpsp</b>	(dos.h)	<b>intr</b>	(dos.h)	<b>segread</b>	(dos.h)
<b>getvect</b>	(dos.h)	<b>keep</b>	(dos.h)	<b>setcbrk</b>	(dos.h)
<b>getverify</b>	(dos.h)	<b>MK_FP</b>	(dos.h)	<b>setdta</b>	(dos.h)
<b>harderr</b>	(dos.h)	<b>outport</b>	(dos.h)	<b>setvect</b>	(dos.h)
<b>hardresume</b>	(dos.h)	<b>outportb</b>	(dos.h)	<b>setverify</b>	(dos.h)
<b>hardretn</b>	(dos.h)	<b>parsfnm</b>	(dos.h)	<b>sleep</b>	(dos.h)
<b>inport</b>	(dos.h)	<b>peek</b>	(dos.h)	<b>unlink</b>	(dos.h)
<b>inportb</b>	(dos.h)	<b>peekb</b>	(dos.h)		

## Manipulation routines

These routines handle strings and blocks of memory: copying, comparing, converting, and searching.

<b>memccpy</b>	(mem.h, string.h)	<b>stricmp</b>	(string.h)
<b>memchr</b>	(mem.h, string.h)	<b>stricmpi</b>	(string.h)
<b>memcmp</b>	(mem.h, string.h)	<b>strlen</b>	(string.h)
<b>memcpy</b>	(mem.h, string.h)	<b>strlwr</b>	(string.h)
<b>memicmp</b>	(mem.h, string.h)	<b>strncat</b>	(string.h)
<b>memmove</b>	(mem.h, string.h)	<b>strncmp</b>	(string.h)
<b>memset</b>	(mem.h, string.h)	<b>strncmpi</b>	(string.h)
<b>movedata</b>	(mem.h, string.h)	<b>strncpy</b>	(string.h)
<b>movmem</b>	(mem.h, string.h)	<b>strnicmp</b>	(string.h)
<b>setmem</b>	(mem.h)	<b>strnset</b>	(string.h)
<b>stpcpy</b>	(string.h)	<b>strpbrk</b>	(string.h)
<b>strcat</b>	(string.h)	<b>strrchr</b>	(string.h)
<b>strchr</b>	(string.h)	<b>strrev</b>	(string.h)
<b>strcmp</b>	(string.h)	<b>strset</b>	(string.h)
<b>strcoll</b>	(string.h)	<b>strspn</b>	(string.h)
<b>strcpy</b>	(string.h)	<b>strstr</b>	(string.h)
<b>strcspn</b>	(string.h)	<b>strtok</b>	(string.h)
<b>strdup</b>	(string.h)	<b>strupr</b>	(string.h)
<b>strerror</b>	(string.h)	<b>strxfrm</b>	(string.h)

## Math routines

These routines perform mathematical calculations and conversions.

<b>abs</b>	(complex.h, stdlib.h)	<b>atof</b>	(stdlib.h, math.h)
<b>acos</b>	(complex.h, math.h)	<b>atoi</b>	(stdlib.h)
<b>arg</b>	(complex.h)	<b>atol</b>	(stdlib.h)
<b>asin</b>	(complex.h, math.h)	<b>bcd</b>	(bcd.h)
<b>atan</b>	(complex.h, math.h)	<b>cabs</b>	(math.h)
<b>atan2</b>	(complex.h, math.h)	<b>ceil</b>	(math.h)

<b>_clear87</b>	(float.h)	<b>ltoa</b>	(stdlib.h)
<b>complex</b>	(complex.h)	<b>_matherr</b>	(math.h)
<b>conj</b>	(complex.h)	<b>matherr</b>	(math.h)
<b>_control87</b>	(float.h)	<b>modf</b>	(math.h)
<b>cos</b>	(complex.h, math.h)	<b>norm</b>	(complex.h)
<b>cosh</b>	(complex.h, math.h)	<b>polar</b>	(complex.h)
<b>div</b>	(math.h)	<b>poly</b>	(math.h)
<b>ecvt</b>	(stdlib.h)	<b>pow</b>	(complex.h, math.h)
<b>exp</b>	(math.h)	<b>pow10</b>	(math.h)
<b>fabs</b>	(math.h)	<b>rand</b>	(stdlib.h)
<b>fcvt</b>	(stdlib.h)	<b>random</b>	(stdlib.h)
<b>floor</b>	(math.h)	<b>randomize</b>	(stdlib.h)
<b>fmod</b>	(math.h)	<b>real</b>	(complex.h)
<b>_fpreset</b>	(float.h)	<b>_rotl</b>	(stdlib.h)
<b>frexp</b>	(math.h)	<b>_rotr</b>	(stdlib.h)
<b>gcvt</b>	(stdlib.h)	<b>sin</b>	(complex.h, math.h)
<b>hypot</b>	(math.h)	<b>sinh</b>	(complex.h, math.h)
<b>imag</b>	(complex.h)	<b>sqrt</b>	(complex.h, math.h)
<b>itoa</b>	(stdlib.h)	<b>srand</b>	(stdlib.h)
<b>labs</b>	(stdlib.h)	<b>_status87</b>	(float.h)
<b>ldexp</b>	(math.h)	<b>strtod</b>	(stdlib.h)
<b>ldiv</b>	(math)	<b>strtol</b>	(stdlib.h)
<b>log</b>	(complex.h, math.h)	<b>strtoul</b>	(stdlib.h)
<b>log10</b>	(complex.h, math.h)	<b>tan</b>	(complex.h, math.h)
<b>_lrotl</b>	(stdlib.h)	<b>tanh</b>	(complex.h, math.h)
<b>_lrotr</b>	(stdlib.h)	<b>ultoa</b>	(stdlib.h)

## Memory routines

---

These routines provide dynamic memory allocation in the small-data and large-data models.

<b>allocmem</b>	(dos.h)	<b>farrealloc</b>	(alloc.h)
<b>brk</b>	(alloc.h)	<b>free</b>	(alloc.h, stdlib.h)
<b>calloc</b>	(alloc.h)	<b>heapcheck</b>	(alloc.h)
<b>coreleft</b>	(alloc.h, stdlib.h)	<b>heapcheckfree</b>	(alloc.h)
<b>farcalloc</b>	(alloc.h)	<b>heapchecknode</b>	(alloc.h)
<b>farcoreleft</b>	(alloc.h)	<b>heapwalk</b>	(alloc.h)
<b>farfree</b>	(alloc.h)	<b>malloc</b>	(alloc.h, stdlib.h)
<b>farheapcheck</b>	(alloc.h)	<b>realloc</b>	(alloc.h, stdlib.h)
<b>farheapcheckfree</b>	(alloc.h)	<b>sbrk</b>	(alloc.h)
<b>farheapchecknode</b>	(alloc.h)	<b>setblock</b>	(dos.h)
<b>farheapfillfree</b>	(alloc.h)		
<b>farheapwalk</b>	(alloc.h)		
<b>farmalloc</b>	(alloc.h)		

## Miscellaneous routines

---

These routines provide nonlocal goto capabilities, sound effects, and locale.

<b>delay</b>	(dos.h)	<b>setjmp</b>	(setjmp.h)
<b>localeconv</b>	(locale.h)	<b>setlocale</b>	(locale.h)
<b>longjmp</b>	(setjmp.h)	<b>sound</b>	(dos.h)
<b>nosound</b>	(dos.h)		

## Process control routines

---

These routines invoke and terminate new processes from within another.

<b>abort</b>	(process.h)	<b>execvp</b>	(process.h)	<b>spawnl</b>	(process.h)
<b>execl</b>	(process.h)	<b>execvpe</b>	(process.h)	<b>spawnle</b>	(process.h)
<b>execlp</b>	(process.h)	<b>_exit</b>	(process.h)	<b>spawnlp</b>	(process.h)
<b>execlpe</b>	(process.h)	<b>exit</b>	(process.h)	<b>spawnlpe</b>	(process.h)
<b>execv</b>	(process.h)	<b>getpid</b>	(process.h)	<b>spawnv</b>	(process.h)
<b>execve</b>	(process.h)	<b>raise</b>	(signal.h)	<b>spawnve</b>	(process.h)
		<b>signal</b>	(signal.h)	<b>spawnvp</b>	(process.h)
				<b>spawnvpe</b>	(process.h)

## Standard routines

---

These are standard routines.

<b>abort</b>	(stdlib.h)	<b>exit</b>	(stdlib.h)	<b>malloc</b>	(stdlib.h)
<b>abs</b>	(stdlib.h)	<b>fcvt</b>	(stdlib.h)	<b>putenv</b>	(stdlib.h)
<b>atexit</b>	(stdlib.h)	<b>free</b>	(stdlib.h)	<b>qsort</b>	(stdlib.h)
<b>atof</b>	(stdlib.h)	<b>gcvt</b>	(stdlib.h)	<b>rand</b>	(stdlib.h)
<b>atoi</b>	(stdlib.h)	<b>getenv</b>	(stdlib.h)	<b>realloc</b>	(stdlib.h)
<b>atol</b>	(stdlib.h)	<b>itoa</b>	(stdlib.h)	<b>srand</b>	(stdlib.h)
<b>bsearch</b>	(stdlib.h)	<b>labs</b>	(stdlib.h)	<b>strtod</b>	(stdlib.h)
<b>calloc</b>	(stdlib.h)	<b>lfind</b>	(stdlib.h)	<b>strtol</b>	(stdlib.h)
<b>ecvt</b>	(stdlib.h)	<b>lsearch</b>	(stdlib.h)	<b>swab</b>	(stdlib.h)
<b>_exit</b>	(stdlib.h)	<b>ltoa</b>	(stdlib.h)	<b>system</b>	(stdlib.h)

## Text window display routines

---

These routines output text to the screen.

<b>clreol</b>	(conio.h)	<b>gotoxy</b>	(conio.h)
<b>clrscr</b>	(conio.h)	<b>highvideo</b>	(conio.h)
<b>delline</b>	(conio.h)	<b>inline</b>	(conio.h)
<b>gettext</b>	(conio.h)	<b>lowvideo</b>	(conio.h)
<b>gettextinfo</b>	(conio.h)	<b>movetext</b>	(conio.h)

<b>normvideo</b>	(conio.h)	<b>textcolor</b>	(conio.h)
<b>puttext</b>	(conio.h)	<b>textmode</b>	(conio.h)
<b>setcursortype</b>	(conio.h)	<b>wherex</b>	(conio.h)
<b>textattr</b>	(conio.h)	<b>wherey</b>	(conio.h)
<b>textbackground</b>	(conio.h)	<b>window</b>	(conio.h)

---

## Time and date routines

These are time conversion and time manipulation routines.

<b>asctime</b>	(time.h)	<b>mktime</b>	(time.h)
<b>ctime</b>	(time.h)	<b>setdate</b>	(dos.h)
<b>difftime</b>	(time.h)	<b>settime</b>	(dos.h)
<b>dostounix</b>	(dos.h)	<b>stime</b>	(time.h)
<b>ftime</b>	(sys\timeb.h)	<b>strftime</b>	(time.h)
<b>getdate</b>	(dos.h)	<b>time</b>	(time.h)
<b>gettime</b>	(dos.h)	<b>tzset</b>	(time.h)
<b>gmtime</b>	(time.h)	<b>unixtodos</b>	(dos.h)
<b>localtime</b>	(time.h)		

---

## Variable argument list routines

These routines are for use when accessing variable argument lists (such as with **vprintf**, etc).

<b>va_arg</b>	(stdarg.h)
<b>va_end</b>	(stdarg.h)
<b>va_start</b>	(stdarg.h)

## *C++ streams*

This chapter gives you a brief overview of C++ stream I/O. Stream I/O in C++ is used to convert typed objects into readable text, and vice versa. It allows you to define input/output functions which are then used automatically for corresponding user-defined types. Further examples can be found in Chapter 5, "A C++ primer," in *Getting Started*; the bibliography in that book offers some titles for more advanced study.

### New streams for old

---

Turbo C++ supports both the original C++ stream library and the new enhanced iostream library of C++ release 2.0. Having both versions will help you if you have programs written under the old conventions and need to use Turbo C++ while you make the transition to the more efficient release 2.0 iostreams. We strongly recommend that all new code should be written using the release 2.0 iostream library. While providing some material on making the transition to 2.0 streams (starting on page 184), this chapter is primarily devoted to the release 2.0 iostream classes and methods.

### Using the 2.0 streams

---

The release 2.0 iostream enhancements, while for the most part upwardly compatible with the older C++ version, offer new

opportunities through the use of multiple inheritance and other C++ release 2.0 features.

For a discussion of the differences between old streams and new iostreams, and for guidelines for converting from the old streams to the new, see “Using the older streams” and “Guidelines for upgrading to 2.0 streams” at the end of this chapter.

The C++ stream concept is aimed at solving several problems with the standard C I/O library functions such as **printf** and **scanf**. The latter, of course, are still available to C++ programmers, but the improved flexibility and elegance of C++ streams makes the `stdio.h` library functions less attractive. The classes associated with C++ streams offer you *extensible* libraries, so that you can perform type-secure formatted I/O on both predefined and user-defined data types using overloaded operators and other object-oriented techniques.

To access stream I/O, your program must include `iostream.h`. Other header files may be needed for some stream functions. For example, `strstream.h` is needed for in-memory formatting using the classes **istrstream** and **ostrstream**. The header file `strstream.h` also includes `iostream.h`. If you want `fstreams`, include `fstream.h`, which also includes `iostream.h`. Conceivably, you could include both `fstream.h` and `strstream.h`.

---

## What’s a stream?

A *stream* is an abstraction referring to any flow of data from a *source* (or *producer*) to a *sink* (or *consumer*). We also use the synonyms *extracting*, *getting*, and *fetching* when speaking of inputting characters from a source; and *inserting*, *putting*, or *storing* when speaking of outputting characters to a sink.

Despite its name, a stream class can be used to format data in situations where input and output is not involved. You will see that *in-memory* formatting is possible with arrays of characters and other structures.

---

## The iostream library

The iostream library has two parallel classes: **streambuf**, and **ios**. Both are low-level classes, each doing a different set of jobs.

**streambuf** The **streambuf** class provides general methods for buffering and handling streams when little or no formatting is required. **streambuf** is a useful base class employed by other parts of the **iostream** library, though it is also available to derive classes for your own functions and libraries. Most of **streambuf**'s member functions (methods) are inline for maximum efficiency. The classes **strstreambuf** and **filebuf** are derived from **streambuf**.

**ios** The class **ios** (and hence any of its derived classes) contains a pointer to **streambuf**.

**ios** has two derived classes: **istream** (for input) and **ostream** (for output). Another class, **iostream**, is derived from both **istream** and **ostream** by multiple inheritance:

```
class ios;  
class istream : virtual public ios;  
class ostream : virtual public ios;  
class iostream : public istream, public ostream;
```

In addition, there are three **withassign** classes derived from **istream**, **ostream**, and **iostream**:

```
class istream_withassign : public istream;  
class ostream_withassign : public ostream;  
class iostream_withassign : public iostream;
```

### The stream classes

- The class **ios** contains state variables for handling the interface with **streambuf**, and for error handling.
- The class **istream** supports both formatted and unformatted conversions of character streams fetched from **streambufs**.
- The **ostream** class supports both formatted and unformatted conversions of character streams stored into **streambufs**.
- The **iostream** class combines **istream** and **ostream** for bidirectional operations where a single stream acts as source and sink.
- The **withassign** derived classes provide four predefined “standard” streams: **cin**, **cout**, **cerr**, and **clog**, as explained in the next section. The **withassign** classes add assignment operators to their respective base classes as follows:

```
class istream_withassign : public istream {  
    istream_withassign();  
};
```

```
    istream& operator=(istream&);
    istream& operator=(streambuf*);
};
```

and similarly for **ostream\_withassign** and **istream\_withassign**.

A stream class is any class derived from **istream** or **ostream**.

---

## The four standard streams

C++ programs start with four predefined open streams, declared as objects of **withassign** classes in `iostream.h` as follows:

```
extern istream_withassign cin;
extern ostream_withassign cout;
extern ostream_withassign cerr;
extern ostream_withassign clog;
```

Their constructors are called each time `iostream.h` is included, but the actual initialization is performed just once.

The four standard streams are as follows:

<i>Corresponds to stdin.</i>	<b>cin</b>	The standard input (file descriptor 0).
<i>Corresponds to stdout.</i>	<b>cout</b>	The standard output (file descriptor 1).
<i>Corresponds to stderr.</i>	<b>cerr</b>	The standard error output (file descriptor 2). <b>cerr</b> is unit buffered, flushed after each insertion.
	<b>clog</b>	This stream is a fully buffered version of <b>cerr</b> .

As in C, you can reassign these standard names to other files or character buffers after program startup.

---

## Output

Stream output is accomplished with the *insertion* or *put to* operator, **<<**. The standard left shift operator, **<<**, is overloaded for output operations. Its left operand is an object of type class **ostream**. Its right operand is any type for which stream output has been defined (more about this later). Stream output is predefined for built-in types. The **<<** operator overloaded for type **type** is called the **type inserter**. For example,

```
cout << "Hello!\n";
```

writes the string "Hello!" to **cout** (the standard output stream, normally your screen) followed by a newline. The **<<** here is the string or **char\*** inserter.

The `<<` operator is left associative and returns a reference to the **ostream** object for which it is invoked. This allows several insertions to be cascaded as follows:

```
void function_display(int i, double d)
{
    cout << "i=" << i << ", d=" << d << "\n";
}
```

This will write something like

```
i = 8, d = 2.34
```

to your standard output.

Note that overloading does not change the normal precedence of `<<`, so you can write

```
cout << "sum = " << x+y << "\n";
```

without parentheses. However, in

```
cout << (x&y) << "\n";
```

the parentheses are needed.

**Built-in types** The inserter types directly supported are: **char (signed and unsigned), short (signed and unsigned), int (signed and unsigned), long (signed and unsigned), char\*** (treated as a string), **float, double, long double, and void\***. Integral types are converted according to the default rules for **printf** (unless you've changed these rules by setting various **ios** flags). For example, given the declarations `int i; long l;`, the two statements

```
cout << i << " " << l;
printf("%d %ld", i, l);
```

give the same result.

Similarly, floating-point types are converted according to the **printf** default rules for the **%g** conversion. So, given the declaration `double d;`, the statements

```
cout << d;
printf("%g", d);
```

produce identical results.

A pointer (**void\***) inserter is also predefined:

```
int i = 1;
cout << &i;           // display pointer in hex
```

The **char** inserter works as follows:

```
char ch = 'A';
cout << ch;      // displays A
```

The **put** and **write** functions

To output binary data or a single character, you can use the member function **put** declared in **ostream** as follows:

```
ostream& ostream::put(char ch);
```

With the declaration `int ch='x';`, the following two lines are equivalent:

```
cout.put(ch);
cout << (char)ch;
```

The **write** member functions let you output larger objects:

```
ostream& ostream::write(const signed char* ptr, int n);
ostream& ostream::write(const unsigned char* ptr, int n);
```

The **write** functions output *n* characters (including any embedded nulls) in binary format. Unlike the string inserter, **write** does not terminate when meeting a null. For example,

```
cout.write((char *)&x, sizeof(x))
```

will send the raw binary representation of *x* to the standard output.



There is a subtle difference between the formatted operator `<<` and the unformatted **put** and **write** functions. The formatted operator can cause flushing of tied streams, and can have a field width associated with it. The unformatted operators do not. So `cout << 'a'` and `cout.put('a')` can produce different results. All formatting flags apply to `<<`, but none apply to **put** or **write**.

Output formatting

Formatting for both input and output is determined by various *format state* flags enumerated in the class **ios**. The states are determined by bits in a **long int** as follows:

```
public:
enum {
    skipws      = 0x0001, // skip whitespace on input
    left        = 0x0002, // left-adjust output
    right       = 0x0004, // right-adjust output
    internal    = 0x0008, // pad after sign or base indicator
    dec         = 0x0010, // decimal conversion
    oct         = 0x0020, // octal conversion
```

```

hex           = 0x0040, // hexadecimal conversion
showbase     = 0x0080, // show base indicator on output
showpoint    = 0x0100, // show decimal point (fp output)
uppercase    = 0x0200, // uppercase hex output
showpos      = 0x0400, // show '+' with positive integers
scientific   = 0x0800, // use 1.2345E2 fp notation and E output
fixed        = 0x1000, // use 123.45 fp notation
unitbuf      = 0x2000, // flush all streams after insertion
stdio        = 0x4000, // flush stdout, stderr after insertion
);

```

These flags, of course, are inherited by the derived classes **ostream** and **istream**. In the absence of specific user action, the format flags are set to give the default formatting shown in the previous examples. Functions are available to set, test, and clear the format flags, either individually or in related groups. Some flags are automatically cleared after each output or input.

**Conversion base** By default, integers are inserted in decimal notation. This can be varied by setting the flag bits *ios::dec*, *ios::oct*, and *ios::hex* (see “Manipulators” on page 172). If all are zero (the default), insertion takes place in decimal.

**Width** The default for inserters is to output the minimum number of characters needed to represent the right-hand operand. To vary this default, you can use the convenient width functions:

```

int ios::width(int w); // set width field to w
                       // and return previous width

int ios::width();     // return current width -- no change

```

The default value for **width** is zero, which outputs without padding. A nonzero width means that inserters will output at least that many characters, padding if necessary to make up the total width needed. Note that no truncation takes place: If the width is less than the number of characters needed, it is ignored (just as if **width** were set to zero). For example,

```

int i = 123;
int old_w = cout.width(6);
cout << i;           // output bbb123 where b=blank.
                       // width is then set to 0
cout.width(old_w);   // restore previous width field

```

Notice that the width is cleared to zero after each formatted insertion, so that in

```

int i, j;
...
cout.width(4);
cout << i << " " << j;

```

the *i* would display at least four characters, but the space and the *j* would display just the minimum needed.

**Manipulators** A simpler way of changing the width state and other format variables is to use a special function-like operator called a *manipulator*. Manipulators take a stream reference as argument and return a reference to the same stream—so manipulators can be embedded in a chain of insertions (or extractions) in order to alter stream states as a side effect without actually performing any insertions (or extractions). For example,

```
cout << setw(4) << i << setw(6) << j;
```

is equivalent to the more verbose

```

cout.width(4);
cout << i;
cout.width(6);
cout << j;

```

**setw** is a *parameterized manipulator* declared in `iomanip.h`. Other parameterized manipulators, **setbase**, **setfill**, **setprecision**, **setiosflags** and **resetiosflags**, work in the same way (see Table 3.1). To make use of these, your program must include `iomanip.h`. You can write your own manipulators without parameters:

*Manipulators with a parameter are more complicated, and require `iomanip.h`.*

```

ostream& dingy( ostream& os)
{
    return os << "\a\a";
}
...
cout << i << dingy << j;

```

Table 3.1: Manipulators

Manipulator	Syntax	Action
<b>dec</b>	<i>outs</i> << <b>dec</b> <i>ins</i> >> <b>dec</b>	Set decimal conversion base format flag
<b>hex</b>	<i>outs</i> << <b>hex</b> <i>ins</i> >> <b>hex</b>	Set hexadecimal conversion base format flag
<b>oct</b>	<i>outs</i> << <b>oct</b> <i>ins</i> >> <b>oct</b>	Set octal conversion base format flag
<b>ws</b>	<i>ins</i> >> <b>ws</b>	Extract whitespace characters
<b>endl</b>	<i>outs</i> << <b>endl</b>	Insert newline and flush stream
<b>ends</b>	<i>outs</i> << <b>ends</b>	Insert terminal null in string
<b>flush</b>	<i>outs</i> << <b>flush</b>	Flush an ostream
<b>setbase(int)</b>	<i>outs</i> << <b>setbase(n)</b>	Set conversion base format to base <i>n</i> (0, 8, 10, or 16). 0 means the default: decimal on output, C rules for literal integers on input.
<b>resetiosflags(long)</b>	<i>ins</i> >> <b>resetiosflags(l)</b> <i>outs</i> << <b>resetiosflags(l)</b>	Clear the format bits in <i>ins</i> or <i>outs</i> specified by argument <i>l</i> .
<b>setiosflags(long)</b>	<i>ins</i> >> <b>setiosflags(l)</b> <i>outs</i> << <b>setiosflags(l)</b>	Set the format bits in <i>ins</i> or <i>outs</i> specified by argument <i>l</i> .
<b>setfill(int)</b>	<i>ins</i> >> <b>setfill(n)</b> <i>outs</i> << <b>setfill(n)</b>	Set the fill character to <i>n</i> .
<b>setprecision(int)</b>	<i>ins</i> >> <b>setprecision(n)</b> <i>outs</i> << <b>setprecision(n)</b>	Set the floating-point precision to <i>n</i> digits
<b>setw(int)</b>	<i>ins</i> >> <b>setw(n)</b> <i>outs</i> << <b>setw(n)</b>	Set field width to <i>n</i>

The non-parameterized manipulators **dec**, **hex**, and **oct** (declared in `ios.h`) take no arguments and simply change the conversion base (and leave it changed):

```
int i = 36;
cout << dec << i << " "
     << hex << i << " "
     << oct << i << endl;
// displays 36 24 44
```

The manipulator **endl** inserts a newline character and flushes the stream. You can also flush an **ostream** at any time with

```
ostream << flush;
```

Filling and padding    The fill character and the direction of the padding depend on the setting of the fill character and the left, right, and internal flags.

The default fill character is a space. You can vary this by using the function **fill**:

```
int i = 123;
cout.fill('*');
cout.width(6);
cout << i;           // display ***123
```

The default direction of padding gives right justification (pad on the left). You can vary these defaults (and other format flags) with the functions **setf** and **unsetf**:

```
int i = 56;
...
cout.width(6);
cout.fill('#');
cout.setf(ios::left,ios::adjustfield);
cout << i;           // display 56####
```

The second argument, *ios::adjustfield*, tells **setf** which bits to set. The first argument, *ios::left*, tells **setf** what to set those bits to. Alternatively, you can use the manipulators **setfill**, **setiosflags**, and **resetiosflags** to modify the fill character and padding mode (see Table 3.1).

User-defined inserters    You can write inserters to output your own data types by overloading the << operator. Suppose you have a type

```
struct info {
    char *name;
    double val;
    char *units;
};
```

You can overload << as follows:

```
ostream& operator << (ostream& s, info& m)
{
    s << m.name << " " << m.val << " " << m.units;
}
```

The statements

```
info x;
...
```

```
// initialize x here
...
cout << x;
```

would produce output like “capacity 1.25 liters”.

## Input

---

Stream input is similar to output but uses the overloaded right shift operator, `>>`, known as the *extraction* (get from) operator, or *extractor*. The `>>` operator provides a more compact and readable alternative to the `scanf` family of functions in `stdio` (it’s also less error-prone). The left operand of `>>` is an object of type class **istream**. As with output, the right operand can be of any type for which stream input has been defined.

All the built-in types listed earlier for output also have predefined extraction operators. You are also free to overload `>>` for stream input to your own data types. The `>>` operator overloaded for type **type** is called the **type extractor**. For example,

```
cin >> x;
```

inputs a value from **cin** (the standard input stream, usually your keyboard) to *x*. The conversion and formatting functions will depend on the type of *x*, how its extractor is defined, and on the settings of the format state flags.

By default, `>>` skips whitespace (as defined by the **isspace** function in `ctype.h`), then reads in characters appropriate to the type of the input object. Whitespace skipping is controlled by the `ios::skipws` flag in the format state’s enumeration (see “Output formatting” on page 170). The `skipws` flag is normally set to give whitespace skipping. Clearing this flag (with **setf**, for example) turns off whitespace skipping. Note also the special “sink” manipulator, **ws**, that lets you discard whitespace (see Table 3.1).

### Chaining extractors

As with `<<`, the `>>` operator is left associative and returns its left operand. The left operand is a reference to the **istream** object for which it is invoked. This allows several input operations to be combined in one statement. Consider the following example:

```
int i;
double d;
cin >> i >> d;
```

The last line causes whitespace to be skipped; digits read from the standard input (by default, your keyboard) are then converted to internal binary form and saved in variable *i*; more whitespace is skipped, and finally a floating-point number is read, converted, and saved in variable *d*.

## Extractors for built-in types

Extractors for the built-in types fall into three categories: integral, floating point, and strings. Each is described in the following sections. For all numeric types, if the first non-whitespace character is not a digit or a sign (or a decimal point for floating-point conversions), the stream enters the fail state (as described on page 177) and no further input will be done until the error condition is cleared.

### Integral extractors

For types **short**, **int**, and **long** (**signed** and **unsigned**), the default action of **>>** is to skip whitespace and convert an integral value, reading input characters until one is found which cannot be part of the legal representation of the type. The format of integral values recognized is the same as that of integer constants in C++, excluding integer suffixes. (See page 11.)

**Warning** If you specify *hex*, *dec*, or *oct* conversion, that is what you'll get. `0x10` becomes 0 in decimal or octal; `010` becomes 10 in decimal, 16 in hexadecimal.

### Floating-point extractors

For types **float** and **double**, the effect of the **>>** operator is to skip whitespace and convert a floating-point value, reading input characters until one is found that cannot be part of a floating-point representation. The format of floating-point values recognized is the same as that of floating-point constants in C++, excluding floating suffixes. (See page 16.)

### Character extractors

For type **char** (signed or unsigned), the effect of the **>>** operator is to skip whitespace and store the next (non-whitespace) character. If you need to read the next character, whether it is whitespace or not, you can use one of the **get** member functions:

```
char ch;
```

```
cin.get(ch);           // ch is set to next char in istream
                      // even if it's a whitespace character
```

The **get** functions for input correspond to the **put** functions for output. The following **get** variant offers control over the number of characters extracted, where they are placed, and the terminating character:

```
istream& istream::get(char *buf, int max, int term='\n');
```

This function reads characters from the input stream into the character array *buf* until it has read *max* – 1 characters, or until it encounters the terminating character given by *term*, whichever comes first. A final null is appended automatically. The default terminator (which need not be specified) is the newline character ('\n'). The terminator itself is not read into the *buf* array, nor is it removed from the **istream**. The *buf* array must be at least *max* chars.

Corresponding to the ostream **write** member function (see page 170), you can read “raw” binary data as follows:

```
cin.read ( (char*)&x, sizeof(x) );
```

For type **char\*** (treated as a string), the effect of the **>>** operator is to skip whitespace and store the next (non-whitespace) characters until another whitespace character is found. A final null (0) character is then appended. Care is needed to avoid “overflowing” a string. The default width of zero (meaning no limit) can be altered using **setw** as follows:

```
char array[SIZE];
...
// initialize array
...
cin.width(sizeof(array));
cin >> array;           // avoids overflow
```

For all input with built-in types, if the end of input occurs before any non-whitespace character is encountered, nothing is stored in the target *buf*, and the *istream* state is set to “fail.” So, if the target was uninitialized, it will still be uninitialized.

Putback function    The member function

```
istream& istream::putback(char c);
```

pushes back just the character *c* into the **istream**; if the character can't be put back, the state of the stream is set to “fail.” The fol-

lowing simple routine reads a C++ identifier from the standard input:

```
void getident (char *s /* where to put ident */)
{
    char c = 0;           // guard against EOF
    cin >> c;            // skip whitespace
    if (isalpha(c) || c == '_')
        do {
            *s++ = c;
            c = 0;       // guard against EOF
            cin.get(c);
        } while (isalnum(c) || c == '_');
    *s = 0;             // terminate the string
    if (c)
        cin.putback(c); // we always get one too many
}
```

User-defined types for input

You can create extractors for your own defined types in the same way as for inserters. Taking the structure information defined on page 174, the operator `>>` can be overloaded as follows:

```
istream& operator >> (istream& s, info& m);
{
    s >> m.name >> m.val >> m.units;
    return s;
}
```

(In a real application, of course, you would add code to check for input errors.) To read an input line such as “capacity 1.25 liters” you would use a line such as

```
cin >> m;
```

---

## Initializing streams

The streams **cin**, **cout**, **cerr**, and **clog** are initialized and opened at program start, and then connected to their standard files. Initializing (constructing) a stream means associating it with a stream buffer. The **ostream** class has the constructor

```
ostream::ostream(streambuf*);
```

which initializes the **ios** state variables and associates a stream buffer with an **ostream** object. The **istream** constructor works in the same way. In most cases, you need not be concerned with the mechanics of buffering.

The `iostream` library offers a variety of classes derived from **`streambuf`**, **`ostream`**, and **`istream`**, allowing a wide choice of methods for creating streams with different sources and sinks and different buffering strategies.

Classes derived from **`streambuf`** are as follows:

- `filebuf`**      **`filebuf`** supports I/O through file descriptors. Member functions support the functions of opening, closing, and seeking.
- `stdiobuf`**     **`stdiobuf`** supports I/O via `stdio` FILE structures, and is provided solely to allow compatibility when mixing C++ code with existing C programs.
- `strstreambuf`** **`strstreambuf`** lets you input and output characters from byte arrays in memory. Two additional classes, **`istrstream`** and **`ostrstream`**, provide formatted in-memory I/O.

Specialized classes for file I/O are derived as follows:

- `ifstream`** is derived from **`istream`**
- `ofstream`** is derived from **`ostream`**
- `fstream`** is derived from **`iostream`**

These three classes support formatted file I/O using **`filebufs`**.

---

## Simple file I/O

The class **`ofstream`** inherits the insertion operations from **`ostream`**, while **`ifstream`** inherits the extraction operations from **`istream`**. They also provide constructors and member functions for creating files and handling file I/O. You must include `fstream.h` in all programs using these classes. Consider the following example that copies the file `FILE_FROM` to the file `FILE_TO`:

```
#include fstream.h
...
char ch;
ifstream f1("file_from");
if (!f1) errmsg("Cannot open 'file_from' for input");
ofstream f2("file_to");
if (!f2) errmsg("Cannot open 'file_to' for output");
while ( f2 && f1.get(ch) ) f2.put(ch);
```

*The not operator (!) is overloaded; see page 183.*

*Stream errors are discussed in detail on page 181.*

Note that if the **`ifstream`** or **`ofstream`** constructors are unable to open the specified files, the appropriate stream error state is set.

The constructors allow you to declare a file stream without specifying a named file. Later, you can associate the file stream with a particular file:

```
ofstream ofile;           // creates output file stream
...
ofile.open("payroll");   // ofile stream associates with
                        // file "payroll"
// do some payrolling
ofile.close();           // "payroll" closes
ofile.open("employee"); // ofile can be reused
```

By default, files are opened in text mode. This means that on input, carriage return/linefeed sequences are converted to the '\n' character. On output, the '\n' character is converted to a carriage-return/linefeed sequence. These translations are not done in binary mode.

The member function **ofstream::open** is declared as follows:

```
void open(char * name, int=ios::out, int prot=filebuf::openprot);
```

Similarly, **ifstream::open** is declared thus:

```
void open(char * name, int=ios::in, int prot=filebuf::openprot);
```

The second argument, known as the *open mode*, defaults as shown. The open mode argument (possibly OR'd with several mode bits) can be given explicitly as follows:

Mode bit	Action
<i>ios::app</i>	Append data—always write at end of file.
<i>ios::ate</i>	Seek to end of file upon original open.
<i>ios::in</i>	Open for input (implied for <b>ifstream</b> s).
<i>ios::out</i>	Open for output (implied for <b>ofstream</b> s).
<i>ios::trunc</i>	Discard contents if file exists (implied if <i>ios::out</i> is specified and neither <i>ios::ate</i> nor <i>ios::app</i> is specified).
<i>ios::nocreate</i>	If file does not exist, <b>open</b> fails.
<i>ios::noreplace</i>	If file exists, <b>open</b> for output fails unless <i>ate</i> or <i>app</i> is set.

The mode mnemonics come from the enumeration **open\_mode** in **ios**:

```
class ios {
public:
    enum open_mode { in, out, app, ate, nocreate, noreplace };
};
```

The statement

```
ofstream ofile("data", ios::app|ios::nocreate);
```

will try to open the file DATA for appended output; it will fail if the file does not exist. Failure will be signaled in the error state of **ofile**. If successful, the stream **ofile** will be attached to the file DATA. The class **fstream** (derived from both **ifstream** and **ofstream**) can be used to create files for simultaneous input and output:

```
fstream inout("data",ios::in|ios::out);
inout << i;
...
inout >> j;
```

You can use the functions **tellg** and **tellp** to determine the current “get” position and the current “put” position of the file; that is, the places in the stream where the next output or input operation will operate:

```
streampos cgp = inout.tellg(); // cgp is current "get" position
```

where **streampos** is **typedef**'d in **fstream.h**. The member functions **seekg** and **seekp** can reset the current get and put positions:

```
inout.seekp(cp); // set current "put" position to cp
```

Variants of **seekp** and **seekg** let you set the current position via relative offsets:

```
inout.seekg(5,ios::beg); // move cp 5 bytes from beginning
inout.seekg(5,ios::cur); // move cp 5 bytes forward
inout.seekp(-5,ios::end); // move cp 5 bytes before end
```

You might want to print out and study the commented header files to see how the various stream classes are related, and how their member functions are declared.

---

## I/O stream error states

*Note that **goodbit** is not a real bit, but a zero value indicating that no error bits have been set.*

Each stream has an associated *error state*, a set of error bits declared as an **io\_state** enumeration in class **ios**:

```
class ios {
public:
    ...
    // stream status bits
    enum io_state {
        goodbit   = 0x00,
        eofbit    = 0x01,
        failbit   = 0x02,
        badbit    = 0x04,
```

```

        hardfail = 0x80
    };
    ...
};

```

Errors occurring during stream I/O set the appropriate bit(s) as indicated in Table 3.2.

Table 3.2  
ios error bits

Status bit	Meaning
<i>goodbit</i>	No bit set, so all is well.
<i>eofbit</i>	“End of file”: set if <b>istream</b> has no more characters available for extraction. Subsequent extraction attempts are ignored.
<i>failbit</i>	Set if last I/O operation (extraction or conversion) has failed. Stream is still usable once error bit cleared.
<i>badbit</i>	Set if last attempted I/O operation was invalid. Stream <i>may</i> be usable after clearing error condition.
<i>hardfail</i>	Set if stream is in an irrecoverable error state.

Once a stream is placed in an error state, all attempts to insert into or extract from that stream will be ignored until the error condition is corrected and the error bit(s) cleared (using, for example, the member function **ios::clear(i)**). The **ios::clear(i)** member function actually sets the error bits according to the integer argument *i*, so that **ios::clear(0)** clears all error bits, except *hardfail*, which cannot be cleared.

Note that inserters and extractors cannot change the state of a stream once an error has occurred. It is therefore sound practice to test for stream errors at appropriate points in your program. Table 3.3 lists the member functions available for testing and setting the error bits.

Table 3.3  
Current stream state  
member functions

Member function	Action
<code>int rdstate();</code>	Returns current error state.
<code>void clear(int i = 0);</code>	Sets error bits to <i>i</i> . For example, this code: <pre>str.clear(ios::failbit str.rdstate());</pre> sets <i>failbit</i> of stream <b>str</b> without disturbing the other bits.
<code>int good();</code>	Returns nonzero if no error bits set; otherwise, returns zero.
<code>int eof();</code>	Returns nonzero if <b>istream</b> <i>eofbit</i> is set; otherwise, returns zero.

Table 3.3: Current stream state member functions (continued)

<code>int fail();</code>	Returns nonzero if <i>failbit</i> , <i>badbit</i> , or <i>hardfail</i> is set; otherwise, returns zero.
<code>int bad();</code>	Returns nonzero if <i>badbit</i> or <i>hardfail</i> is set; otherwise, returns zero.

You can also check for errors by testing a stream as though it were a Boolean expression:

```
if (cin >> x) return;    // input ok
...                    // error recovery here
if (!cout) errmsg("Output Error!");
```

These examples reveal the elegance of C++. The class **ios** has the following operator function declarations:

```
int operator! ();
operator void* ();
```

The **void\*()** operator is defined to “convert” a stream to a pointer which will be 0 (false) if *failbit*, *badbit*, or *hardfail* are set, but non-null (true) otherwise. (Note: The returned pointer is to be used only for Boolean testing; it has no other practical significance.) The overloaded not operator (!) is defined to return nonzero (true) if the stream’s *failbit*, *badbit*, or *hardfail* are set; otherwise, it returns zero (false).

## Using the older streams

---

Although the C++ release 1.x stream and release 2.0 iostream libraries share many class and function names and offer many similar facilities, their structures differ in some crucial areas. Turbo C++ therefore implements the two streams with separate libraries and header files. To work entirely with old streams code, you must include `stream.h`, avoid including `iostream.h`, and link with the old stream library. Consult the file `OLDSTR.DOC` for more information on the release 1.x streams. We also encourage you to study the declarations and comments in `stream.h`.

Depending on which classes and features your old streams programs use, it is possible that they might compile and run satisfactorily with the new iostream library.

## Guidelines for upgrading to 2.0 streams

---

A key difference between the old and new streams classes is that most of the old **streambuf** class public members are now declared as protected in the new **streambuf** class. If your old stream code makes direct reference to such members, or if you have derived classes from **streambuf** that rely on such members, you will need to revamp your programs before they can run with the **iostream** library. Another difference that may affect upward compatibility is that the old **streambuf** directly supported the use of character arrays for in-memory formatting. Under **iostreams**, this support is assumed by the derived class **strstreambuf** declared in **strstream.h**.

Old stream constructors invoking **filebufs** such as

```
istream instream(file_descriptor);
```

must be replaced by

```
ifstream instream(file_descriptor);
```

in **iostreams** programs.

The old and new stream classes interact differently with **stdio**. For example, **stream.h** includes **stdio.h** and the old **istream** and **ostream** support pointers to the **stdio** **FILE** structure. With **istream**, **stdio** is supported via the specialized **stdiostream** class declared in **stdiostream.h**.

In the old stream library, the predefined streams **cin**, **cout**, and **cerr** are connected directly to **stdio**'s **FILEs** **stdin**, **stdout**, and **stderr**. With **iostream**, they are connected to file descriptors and use different buffering strategies. To avoid buffering problems when mixing **stdout** and **cout** code, you can use

```
ios::sync_with_stdio();
```

which connects the predefined streams with the **stdio** files in unbuffered mode. Note, though, that this slows **cin**, **cout**, and **cerr** considerably.

The old stream library allowed a stream to be directly assigned to another stream; for example,

```
ostream outs; outs = cout;    // old streams only
```

With **istream**, this is only possible if the left-hand stream is assignable; in other words, of type **istream\_withassign** or

**ostream\_withassign.** If your program contains such assignments, you can either rewrite them using pointers or references, or you can change the declarations:

```
ostream_withassign outs = cout;    // new iostreams only
outs << i;
```



# Memory models, floating point, and overlays

This chapter covers three major topics:

- **Memory models**, from tiny to huge. We tell you what they are, how to choose one, and why you would (or would not) want to use a particular memory model.
- **Floating-point options**. How and when to use them.
- **Overlays**. How they work, how to use them.

## Memory models

---

*See page 194 for a summary of each memory model.*

Turbo C++ gives you six memory models, each suited for different program and code sizes. Each memory model uses memory differently. What do you need to know to use memory models? To answer that question, we have to take a look at the computer system you're working on. Its central processing unit (CPU) is a microprocessor belonging to the Intel iAPx86 family; probably an 8088 or 80286, though possibly an 8086, 80186, 80386, or an 80486. For now, we'll just refer to it as an 8086.

## The 8086 registers

---

These are the registers found in the 8086 processor. There is one more register—IP (instruction pointer)—but Turbo C++ can't access it directly, so it isn't shown here.

Figure 4.1  
8086 registers

General-purpose registers			
AX	AH	AL	accumulator (math operations)
BX	BH	BL	base (indexing)
CX	CH	CL	count (indexing)
DX	DH	DL	data (holding data)

Segment address registers			
CS			code segment pointer
DS			data segment pointer
SS			stack segment pointer
ES			extra segment pointer

Special-purpose registers			
SP			stack pointer
BP			base pointer
SI			source index
DI			destination index

### General-purpose registers

The general-purpose registers are the ones used most often to hold and manipulate data. Each has some special functions that only it can do. For example,

- Some math operations can only be done using AX.
- BX can be used as an index register.
- CX is used by LOOP and some string instructions.
- DX is implicitly used for some math operations.

But there are many operations that all these registers can do; in many cases, you can freely exchange one for another.

Segment registers The segment registers hold the starting address of each of the four segments. As described in the next section, the 16-bit value in a segment register is shifted left 4 bits (multiplied by 16) to get the true 20-bit address of that segment.

Special-purpose registers The 8086 also has some special-purpose registers:

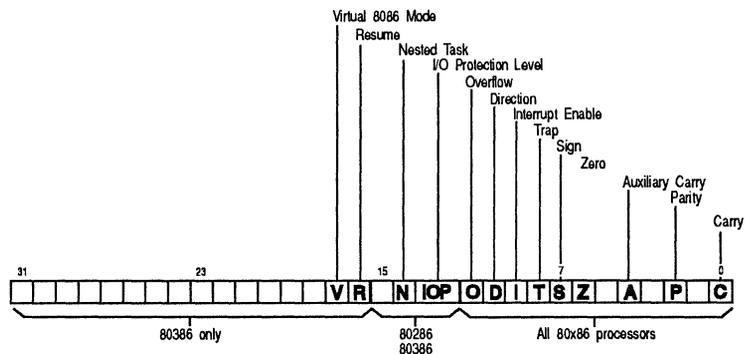
- The SI and DI registers can do many of the things the general-purpose registers can, plus they are used as index registers. They're also used by Turbo C++ for register variables.
- The SP register points to the current top-of-stack and is an offset into the stack segment.
- The BP register is a secondary stack pointer, usually used to index into the stack in order to retrieve arguments or automatic variables.

C functions use the base pointer (BP) register as a base address for arguments and automatic variables. Parameters have positive offsets from BP, which vary depending on the memory model. BP always points to the saved previous BP value. Functions that have no arguments will not use or save BP if the Standard Stack Frame option is *Off*.

Automatic variables are given negative offsets from BP. The offsets depend on how much space has already been assigned to variables.

The flags register The 16-bit flags register contains all pertinent information about the state of the 8086 and the results of recent instructions.

Figure 4.2  
Flags register of the 8086



For example, if you wanted to know whether a subtraction produced a zero result, you would check the *zero flag* (the Z bit in the flags register) immediately after the instruction; if it were set, you would know the result was zero. Other flags, such as the *carry* and *overflow flags*, similarly report the results of arithmetic and logical operations.

Other flags control modes of operation of the 8086. The *direction flag* controls the direction in which the string instructions move, and the *interrupt flag* controls whether external hardware, such as a keyboard or modem, is allowed to halt the current code temporarily so that urgent needs can be serviced. The *trap flag* is used only by software that debugs other software.

The flags register isn't usually modified or read directly. Instead, the flags register is generally controlled through special assembler instructions (such as **CLD**, **STI**, and **CMC**) and through arithmetic and logical instructions that modify certain flags. Likewise, the contents of certain bits of the flags register affect the operation of instructions such as **JZ**, **RCR**, and **MOVSB**. The flags register is not really used as a storage location, but rather holds the status and control data for the 8086.

---

## Memory segmentation

The Intel 8086 microprocessor has a *segmented memory architecture*. It has a total address space of 1 Mb, but it is designed to directly address only 64K of memory at a time. A 64K chunk of memory is known as a segment; hence the phrase, "segmented memory architecture."

- The 8086 keeps track of four different segments: *code*, *data*, *stack*, and *extra*. The code segment is where the machine instructions are; the data segment, where information is; the stack is, of course, the stack; and the extra segment is also used for extra data.
- The 8086 has four 16-bit segment registers (one for each segment) named CS, DS, SS, and ES; these point to the code, data, stack, and extra segments, respectively.
- A segment can be located anywhere in memory—at least, almost anywhere. For reasons that will become clear as you read on, a segment must start on an address that's evenly divisible by 16 (in base 10).

Address calculation    A complete address on the 8086 is composed of two 16-bit values: the segment address and the offset. Suppose the data segment address—the value in the DS register—is 2F84 (base 16), and you want to calculate the actual address of some data that has an offset of 0532 (base 16) from the start of the data segment; how is that done?

Address calculation is done as follows: Shift the value of the segment register 4 bits to the left (equivalent to one hex digit), then add in the offset.

The resulting 20-bit value is the actual address of the data, as illustrated here:

DS register (shifted):	0010 1111 1000 0100 0000	=	2F840
Offset:	0000 0101 0011 0010	=	00532
<hr/>			
Address:	0010 1111 1101 0111 0010	=	2FD72

*A chunk of 16 bytes is known as a paragraph, so you could say that a segment always starts on a paragraph boundary.*

The starting address of a segment is always a 20-bit number, but a segment register only holds 16 bits—so the bottom 4 bits are always assumed to be all zeros. This means—as we said—that segments can only start every 16 bytes through memory, at an address where the last 4 bits (or last hex digit) are zero. So, if the DS register is holding a value of 2F84, then the data segment actually starts at address 2F840.

The standard notation for an address takes the form *segment:offset*; for example, the previous address would be written as 2F84:0532. Note that since offsets can overlap, a given segment:offset pair is not unique; the following addresses all refer to the same memory location:

0000:0123  
0002:0103  
0008:00A3  
0010:0023  
0012:0003

Segments can (but do not have to) overlap. For example, all four segments could start at the same address, which means that your entire program would take up no more than 64K—but that's all the space you would have for your code, your data, and your stack.

## Pointers

---

What do pointers have to do with memory models and Turbo C++? A lot. The type of memory model you choose will determine the default type of pointers used for code and data (though you can explicitly declare a pointer or a function to be of a specific type regardless of the model being used). Pointers come in four flavors: *near* (16 bits), *far* (32 bits), *huge* (also 32 bits), and *segment* (16 bits).

**Near pointers** A 16-bit (near) pointer relies on one of the segment registers to finish calculating its address; for example, a pointer to a function would add its 16-bit value to the left-shifted contents of the code segment (CS) register. In a similar fashion, a near data pointer contains an offset to the data segment (DS) register. Near pointers are easy to manipulate, since any arithmetic (such as addition) can be done without worrying about the segment.

**Far pointers** A far (32-bit) pointer contains not only the offset within the segment, but also (as another 16-bit value) the segment address, which is then left-shifted and added to the offset. By using far pointers, you can have multiple code segments; that, in turn, allows you to have programs larger than 64K. You can also address more than 64K of data.

When you use far pointers for data, you need to be aware of some potential problems in pointer manipulation. As explained in the section on address calculation, you can have many different segment:offset pairs refer to the same address. For example, the far pointers 0000:0120, 0010:0020, and 0012:0000 all resolve to the same 20-bit address. However, if you had three different far pointer variables—*a*, *b*, and *c*—containing those three values respectively, then all the following expressions would be *false*:

```
if (a == b) . . .
if (b == c) . . .
if (a == c) . . .
```

A related problem occurs when you want to compare far pointers using the `>`, `>=`, `<`, and `<=` operators. In those cases, only the offset (as an **unsigned**) is used for comparison purposes; given that *a*, *b*, and *c* still have the values previously listed, the following expressions would all be *true*:

```
if (a > b) . . .
if (b > c) . . .
if (a > c) . . .
```

The equals (==) and not-equal (!=) operators use the 32-bit value as an **unsigned long** (not as the full memory address). The comparison operators (<=, >=, <, and >) use just the offset.

The == and != operators need all 32 bits, so the computer can compare to the NULL pointer (0000:0000). If you used only the offset value for equality checking, any pointer with 0000 offset would be equal to the NULL pointer, which is not what you want.

**Important!**

If you add values to a far pointer, only the offset is changed. If you add enough to cause the offset to exceed FFFF (its maximum possible value), the pointer just wraps around back to the beginning of the segment. For example, if you add 1 to 5031:FFFF, the result would be 5031:0000 (not 6031:0000). Likewise, if you subtract 1 from 5031:0000, you would get 5031:FFFF (not 5030:000F).

If you want to do pointer comparisons, it's safest to use either near pointers—which all use the same segment address—or huge pointers, described next.

**Huge pointers**

Huge pointers are also 32 bits long. Like far pointers, they contain both a segment address and an offset. Unlike far pointers, however, they are *normalized* to avoid the problems associated with far pointers.

What is a normalized pointer? It is a 32-bit pointer which has as much of its value in the segment address as possible. Since a segment can start every 16 bytes (10 in base 16), this means that the offset will only have a value from 0 to 15 (0 to F in base 16).

To normalize a pointer, convert it to its 20-bit address, then use the right 4 bits for your offset and the left 16 bits for your segment address. For example, given the pointer 2F84:0532, you would convert that to the absolute address 2FD72, which you would then normalize to 2FD7:0002. Here are a few more pointers with their normalized equivalents:

```
0000:0123    0012:0003
0040:0056    0045:0006
500D:9407    594D:0007
7418:D03F    811B:000F
```

There are three reasons why it is important to always keep huge pointers normalized.

1. Because, doing it that way, for any given memory address there is only one possible huge address—segment:offset pair—for it. That means that the `==` and `!=` operators return correct answers for any huge pointers.
2. In addition, the `>`, `>=`, `<`, and `<=` operators are all used on the full 32-bit value for huge pointers. Normalization guarantees that the results there will be correct also.
3. Finally, because of normalization, the offset in a huge pointer automatically wraps around every 16 values, but—unlike far pointers—the segment is adjusted as well. For example, if you were to increment `811B:000F`, the result would be `811C:0000`; likewise, if you decrement `811C:0000`, you get `811B:000F`. It is this aspect of huge pointers that allows you to manipulate data structures greater than 64K in size. This ensures that, for example, if you have a huge array of **structs** that's larger than 64K, indexing into the array and selecting a **struct** field will always work with structs of any size.

There is a price for using huge pointers: additional overhead. Huge pointer arithmetic is done with calls to special subroutines. Because of this, huge pointer arithmetic is significantly slower than that of far or near pointers.

## The six memory models

---

Turbo C++ gives you six memory models: tiny, small, medium, compact, large, and huge. Your program requirements determine which one you pick. Here's a brief summary of each:

*Use this model when memory is at an absolute premium.*

### **Tiny**

As you might guess, this is the smallest of the memory models. All four segment registers (CS, DS, SS, ES) are set to the same address, so you have a total of 64K for all of your code, data, and stack. Near pointers are always used. Tiny model programs can be converted to .COM format by linking with the `/t` option.

*This is a good size for average applications.*

### **Small**

The code and data segments are different and don't overlap, so you have 64K of code and 64K of data and stack. Near pointers are always used.

*Best for large programs that don't keep much data in memory.*

### **Medium**

Far pointers are used for code, but not for data. As a result, data plus stack are limited to 64K, but code can occupy up to 1 Mb.

Best if your code is small but you need to address a lot of data.

Large and huge are needed only for very large applications.

- Compact** The inverse of medium: Far pointers are used for data, but not for code. Code is then limited to 64K, while data has a 1-Mb range.
- Large** Far pointers are used for both code and data, giving both a 1-Mb range.
- Huge** Far pointers are used for both code and data. Turbo C++ normally limits the size of all static data to 64K; the huge memory model sets aside that limit, allowing data to occupy more than 64K.

In order to select these memory models, you can either use menu selections from the integrated environment, or you can type options invoking the command-line compiler version of Turbo C++.

The following illustrations (Figures 4.3 through 4.8) show how memory in the 8086 is apportioned for the six Turbo C++ memory models.

Figure 4.3  
Tiny model memory segmentation

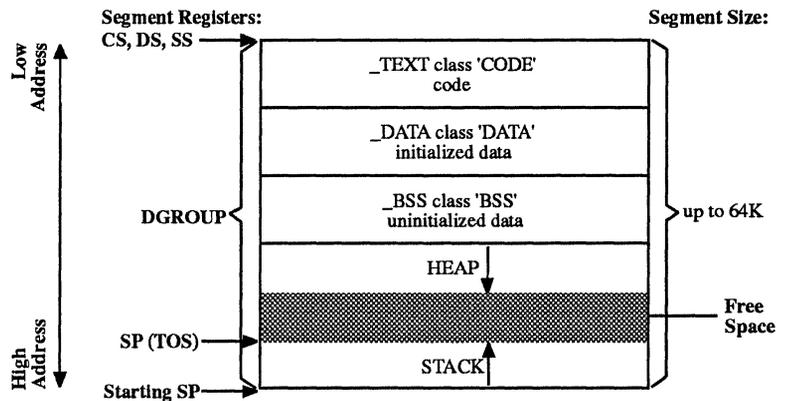


Figure 4.4  
Small model memory segmentation

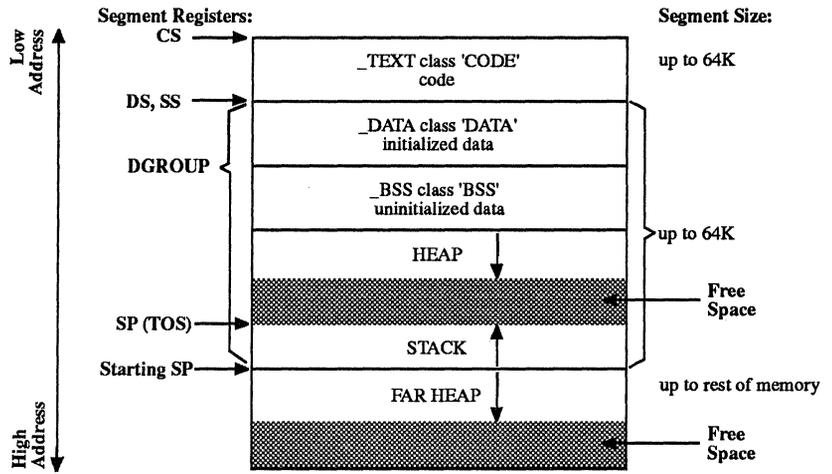


Figure 4.5  
Medium model memory segmentation  
*CS points to only one sfile at a time*

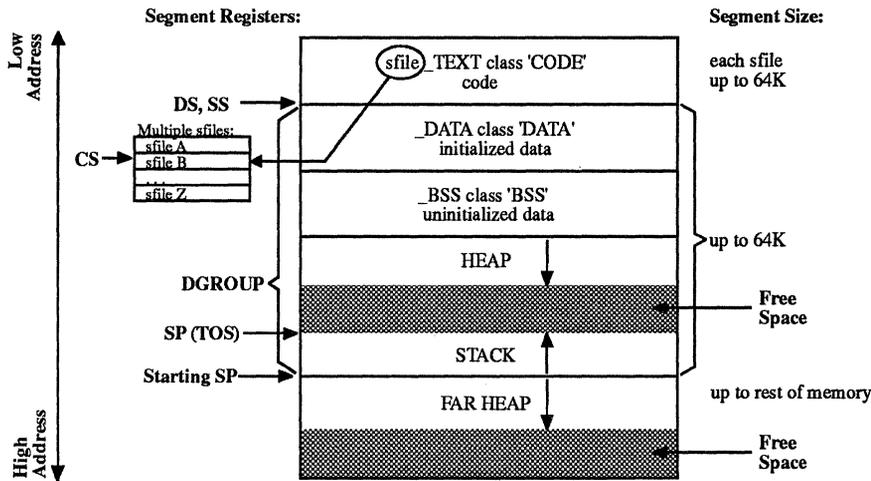


Figure 4.6  
Compact model memory  
segmentation

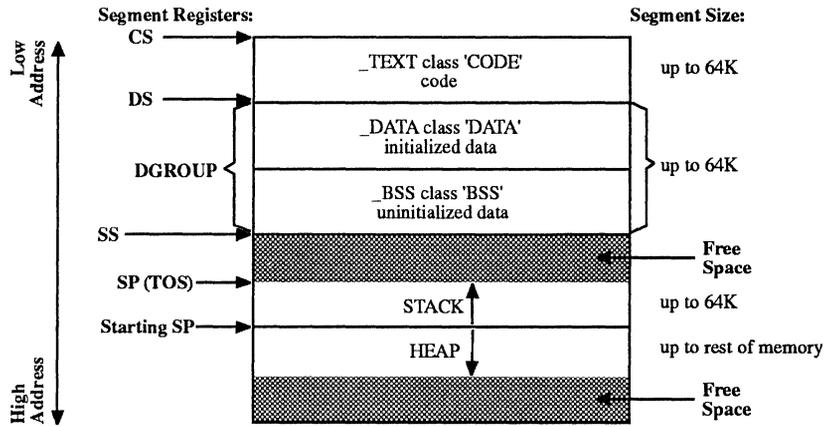


Figure 4.7  
Large model memory  
segmentation

*CS points to only one sfile at a time*

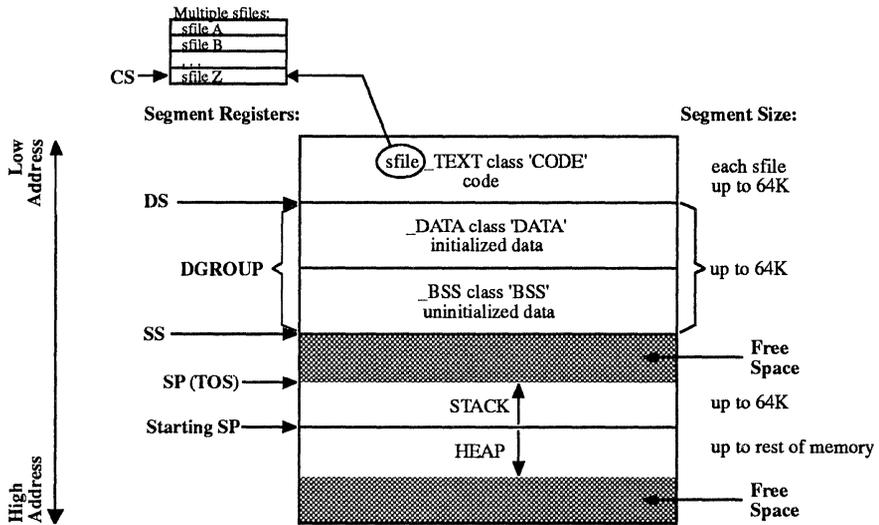


Figure 4.8  
Huge model memory segmentation

CS and DS point to only one sfile at a time

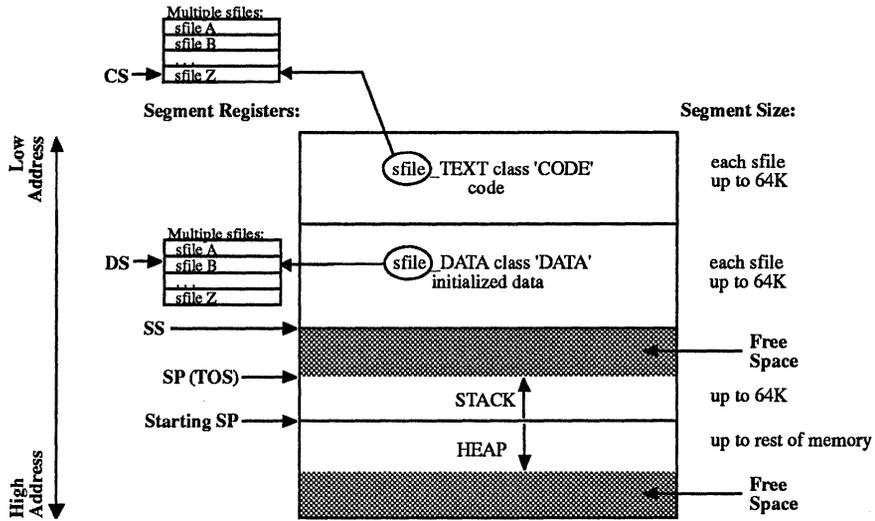


Table 4.1 summarizes the different models and how they compare to one another. The models are often grouped according to whether their code or data models are *small* (64K) or *large* (1 Mb); these groups correspond to the rows and columns in Table 4.1.

Table 4.1  
Memory models

Data size	Code size	
	64K	1 Mb
64K	Tiny (data, code overlap; total size = 64K)	Medium (small data, large code)
1 Mb	Compact (large data, small code)	Large (large data, code)  Huge (same as large but static data > 64K)

The models tiny, small, and compact are small code models because, by default, code pointers are near; likewise, compact, large, and huge are large data models because, by default, data pointers are far.

**Important!**

When you compile a module (a given source file with some number of routines in it), the resulting code for that module cannot be greater than 64K, since it must all fit inside of one code segment. This is true even if you're using one of the larger code models (medium, large, or huge). If your module is too big to fit

into one (64K) code segment, you must break it up into different source code files, compile each file separately, then link them together. Similarly, even though the huge model permits static data to total more than 64K, it still must be less than 64K in *each* module.

## Mixed-model programming: Addressing modifiers

---

Turbo C++ introduces eight new keywords not found in standard ANSI C (**near**, **far**, **huge**, **\_cs**, **\_ds**, **\_es**, **\_ss**, and **\_seg**) that can be used as modifiers to pointers (and in some cases, to functions), with certain limitations and warnings.

In Turbo C++, you can modify the declarations of functions and pointers with the keywords **near**, **far**, or **huge**. We explained **near**, **far**, and **huge** data pointers earlier in this chapter. **near** functions are invoked with near calls and exit with near returns. Similarly, **far** functions are called far and do far returns. **huge** functions are like **far** functions, except that **huge** functions set DS to a new value, while **far** functions do not.

There are also four special **near** data pointers: **\_cs**, **\_ds**, **\_es**, and **\_ss**. These are 16-bit pointers that are specifically associated with the corresponding segment register. For example, if you were to declare a pointer to be

```
char _ss *p;
```

then *p* would contain a 16-bit offset into the stack segment.

Segment pointers are restricted in terms of what can be done with them.

- You can't increment or decrement segment pointers. When you add or subtract an integer to a segment pointer, it is implicitly converted to a far pointer, and the arithmetic is performed as if the integer were added to or subtracted from the far pointer.
- When a segment pointer is used in an indirection expression, it is also implicitly converted to a far pointer.
- As an extension to the binary **+** operator, if a segment pointer is added to a near pointer, the result is a far pointer that is formed by using the segment from the segment pointer and the offset from the near pointer. This operation is only allowed if the two pointers point to the same type, or else if one of the pointers points to a **void** type.

- Segment pointers can be compared. They are compared as if their values were **unsigned** integers.

Functions and pointers within a given program default to near or far, depending on the memory model you select. If the function or pointer is near, it is automatically associated with either the CS or DS register.

The next table shows just how this works. Note that the size of the pointer corresponds to whether it is working within a 64K memory limit (near, within a segment) or inside the general 1 Mb memory space (far, has its own segment address).

Table 4.2  
Pointer results

Memory model	Function pointers	Data pointers
Tiny	near, _cs	near, _ds
Small	near, _cs	near, _ds
Medium	far	near, _ds
Compact	near, _cs	far
Large	far	far
Huge	far	far

Pointers to data can also be declared using the **\_seg** modifier. These are 16-bit segment pointers.

## Declaring functions to be near or far

On occasion, you'll want (or need) to override the default function type of your memory model shown in Table 4.1 (page 198).

For example, suppose you're using the large memory model, but you have a recursive (self-calling) function in your program, like this:

```
double power(double x,int exp)
{
    if (exp <= 0)
        return(1);
    else
        return(x * power(x, exp-1));
}
```

Every time **power** calls itself, it has to do a far call, which uses more stack space and clock cycles. By declaring **power** as **near**, you eliminate some of the overhead by forcing all calls to that function to be near:

```
double near power(double x,int exp)
```

This guarantees that **power** is callable only within the code segment in which it was compiled, and that all calls to it are near calls.

This means that if you are using a large code model (medium, large, or huge), you can only call **power** from within the module where it is defined. Other modules have their own code segment and thus cannot call **near** functions in different modules. Furthermore, a near function must be either defined or declared before the first time it is used, or the compiler won't know it needs to generate a near call.

Conversely, declaring a function to be far means that a far return is generated. In the small code models, the far function must be declared or defined before its first use to ensure it is invoked with a far call.

Look back at the **power** example. It is wise to also declare **power** as static, since it should only be called from within the current module. That way, being a static, its name will not be available to any functions outside the module.

---

## Declaring pointers to be near, far, or huge

You've seen why you might want to declare functions to be of a different model than the rest of the program. Why might you want to do the same thing for pointers? For the same reasons given in the preceding section: either to avoid unnecessary overhead (declaring **near** when the default would be **far**) or to reference something outside of the default segment (declaring **far** or **huge** when the default would be **near**).

There are, of course, potential pitfalls in declaring functions and pointers to be of nondefault types. For example, say you have the following small model program:

```
void myputs(s)
char *s;
{
    int i;
    for (i = 0; s[i] != 0; i++) putc(s[i]);
}

main()
{
    char near *mystr;
    mystr = "Hello, world\n";
```

```

    myputs (mystr);
}

```

This program works fine, and, in fact, the **near** declaration on *mystr* is redundant, since all pointers, both code and data, will be near.

But what if you recompile this program using the compact (or large or huge) memory model? The pointer *mystr* in **main** is still near (it's still a 16-bit pointer). However, the pointer *s* in **myputs** is now far, since that's the default. This means that **myputs** will pull two words out of the stack in an effort to create a far pointer, and the address it ends up with will certainly not be that of *mystr*.

How do you avoid this problem? The solution is to define **myputs** in modern C style, like this:

```

void myputs(char *s)
{
    /* body of myputs */
}

```

*If you're going to explicitly declare pointers to be of type **far** or **near**, be sure to use function prototypes for any functions that might use them.*

Now when Turbo C++ compiles your program, it knows that **myputs** expects a pointer to **char**; and since you're compiling under the large model, it knows that the pointer must be **far**. Because of that, Turbo C++ will push the data segment (DS) register onto the stack along with the 16-bit value of *mystr*, forming a far pointer.

How about the reverse case: Arguments to **myputs** declared as **far** and compiling with a small data model? Again, without the function prototype, you will have problems, since **main** will push both the offset and the segment address onto the stack, but **myputs** will only expect the offset. With the prototype-style function definitions, though, **main** will only push the offset onto the stack.

Pointing to a given segment:offset address

How do you make a far pointer point to a given memory location (a specific segment:offset address)? You can use the built-in library routine **MK\_FP**, which takes a segment and an offset and returns a far pointer. For example,

```

MK_FP(segment_value, offset_value)

```

Given a **far** pointer, *fp*, you can get the segment component with **FP\_SEG(fp)** and the offset component with **FP\_OFF(fp)**. For more information about these three Turbo C++ library routines, refer to the *Library Reference*.

## Using library files

---

Turbo C++ offers a version of the standard library routines for each of the six memory models. Turbo C++ is smart enough to link in the appropriate libraries in the proper order, depending on which model you've selected. However, if you're using the Turbo C++ linker, TLINK, directly (as a standalone linker), you need to specify which libraries to use. Read the section on TLINK in Chapter 5, "Utilities", in the *User's Guide* for details on how to do so.

## Linking mixed modules

---

What if you compiled one module using the small memory model, and another module using the large model, then wanted to link them together? What would happen?

The files would link together fine, but the problems you would encounter would be similar to those described in the earlier section, "Declaring functions to be near or far." If a function in the small module called a function in the large module, it would do so with a near call, which would probably be disastrous. Furthermore, you could face the same problems with pointers as described in the earlier section, "Declaring pointers to be near, far, or huge," since a function in the small module would expect to pass and receive **near** pointers, while a function in the large module would expect **far** pointers.

The solution, again, is to use function prototypes. Suppose that you put **myputs** into its own module and compile it with the large memory model. Then create a header file called `myputs.h` (or some other name with a `.h` extension), which would have the following function prototype in it:

```
void far myputs(char far *s);
```

Now, if you put **main** into its own module (called `MYMAIN.C`), set things up like this:

```
#include <stdio.h>
#include "myputs.h"

main()
{
    char near *mystr;
    mystr = "Hello, world\n";
```

```
    myputs(mystr);  
}
```

When you compile this program, Turbo C++ reads in the function prototype from MYPUTS.H and sees that it is a **far** function that expects a **far** pointer. Because of that, it will generate the proper calling code, even if it's compiled using the small memory model.

What if, on top of all this, you need to link in library routines? Your best bet is to use one of the large model libraries and declare everything to be **far**. To do this, make a copy of each header file you would normally include (such as `stdio.h`), and rename the copy to something appropriate (such as `fstdio.h`).

Then edit each function prototype in the copy so that it is explicitly **far**, like this:

```
int far cdecl printf(char far * format, ...);
```

That way, not only will **far** calls be made to the routines, but the pointers passed will be **far** pointers as well. Modify your program so that it includes the new header file:

```
#include <fstdio.h>  
  
main()  
{  
    char near *mystr;  
    mystr = "Hello, world\n";  
    printf(mystr);  
}
```

Compile your program with TCC, then link it with TLINK, specifying a large model library, such as CL.LIB. Mixing models is tricky, but it can be done; just be prepared for some difficult bugs if you do things wrong.

## Floating-point options

---

There are two types of numbers you work with in C: integer (**int**, **short**, **long**, and so on) and floating point (**float**, **double**, and **long double**). Your computer's processor is set up to easily handle integer values, but it takes more time and effort to handle floating-point values.

However, the iAPx86 family of processors has a corresponding family of math coprocessors, the 8087, the 80287, and the 80387.

We refer to this entire family of math coprocessors as the 80x87, or “the coprocessor.”

*If you have an 80486 processor, the numeric coprocessor is already built in.*

The 80x87 is a special hardware numeric processor that can be installed in your PC. It executes floating-point instructions very quickly. If you use floating point a lot, you’ll probably want a coprocessor. The CPU in your computer interfaces to the 80x87 via special hardware lines.

---

## Emulating the 80x87 chip

The default Turbo C++ code generation option is *emulation* (the `-f` command-line compiler option). This option is for programs that may or may not have floating point, and for machines that may or may not have an 80x87 math coprocessor.

With the emulation option, the compiler will generate code as if the 80x87 were present, but will also link in the emulation library (EMU.LIB). When the program runs, it will use the 80x87 if it is present; if no coprocessor is present at run time, the program uses special software that *emulates* the 80x87.

---

## Using 80x87 code

If your program is only going to run on machines with an 80x87 math coprocessor, you can save about 10K bytes in your .EXE file by omitting the 80x87 autodetection and emulation logic. Simply choose the 80x87 floating-point code generation option (the `-f87` command-line compiler option). Turbo C++ will then link your programs with FP87.LIB instead of EMU.LIB.

---

## No floating-point code

If there is no floating-point code in your program, you can save a small amount of link time by choosing **None** for the floating-point code generation option (the `-f-` command-line compiler option). Then Turbo C++ will not link with EMU.LIB, FP87.LIB, or MATHx.LIB.

---

## Fast floating-point option

Turbo C++ has a fast floating-point option (the `-ff` command-line compiler option). It can be turned off with `-ff-` on the command line. Its purpose is to allow certain optimizations that are technically contrary to correct C semantics. For example,

```
double x;  
x = (float) (3.5*x);
```

To execute this correctly,  $x$  is multiplied by 3.5 to give a **double** that is truncated to **float** precision, then stored as a **double** in  $x$ . Under the fast floating-point option, the **long double** product is converted directly to a **double**. Since very few programs depend on the loss of precision in passing to a narrower floating-point type, fast floating point is the default.

---

## The 87 environment variable

If you build your program with 80x87 emulation, which is the default, your program will automatically check to see if an 80x87 is available, and will use it if it is.

There are some situations in which you might want to override this default autodetection behavior. For example, your own runtime system might have an 80x87, but you need to verify that your program will work as intended on systems without a coprocessor. Or your program may need to run on a PC-compatible system, but that particular system returns incorrect information to the autodetection logic (saying that a nonexistent 80x87 is available, or vice versa).

Turbo C++ provides an option for overriding the start-up code's default autodetection logic; this option is the 87 environment variable.

You set the 87 environment variable at the DOS prompt with the SET command, like this:

```
C> SET 87=N
```

or like this:

```
C> SET 87=Y
```

Don't include spaces to either side of the =. Setting the 87 environment variable to N (for No) tells the start-up code that you do not want to use the 80x87, even though it might be present in the system.

Setting the 87 environment variable to Y (for Yes) means that the coprocessor is there, and you want the program to use it. Let the programmer beware!! If you set 87 = Y when, in fact, there is no 80x87 available on that system, your system will hang.

If the 87 environment variable has been defined (to any value) but you want to undefine it, enter the following at the DOS prompt:

```
C> SET 87=
```

Press *Enter* immediately after typing the equal sign.

---

## Registers and the 80x87

There are a couple of points concerning registers that you should be aware of when using floating point.

1. In 80x87 emulation mode, register wraparound and certain other 80x87 peculiarities are not supported.
  2. If you are mixing floating point with inline assembly, you may need to take special care when using registers. This is because the 80x87 register set is emptied before Turbo C++ calls a function. You might need to pop and save the 80x87 registers before calling functions that use the coprocessor, unless you are sure that enough free registers exist.
- 

## Disabling floating-point exceptions

By default, Turbo C++ programs abort if a floating-point overflow or divide by zero error occurs. You can mask these floating-point exceptions by a call to **\_control87** in **main**, before any floating-point operations are performed. For example,

```
#include <float.h>
main() {
    _control87(MCW_EM,MCW_EM);
    ...
}
```

You can determine whether a floating-point exception occurred after the fact by calling **\_status87** or **\_clear87**. See the entries for these functions in Chapter 1 of the *Library Reference* for details.

Certain math errors can also occur in library functions; for instance, if you try to take the square root of a negative number. The default behavior is to print an error message to the screen, and to return a NAN (an IEEE not-a-number). Use of the NAN will likely cause a floating-point exception later, which will abort the program if unmasked. If you don't want the message to be printed, insert the following version of **matherr** into your program.

```

#include <math.h>
int cdecl matherr(struct exception *e)
{
    return 1;          /* error has been handled */
}

```

Any other use of **matherr** to intercept math errors is not encouraged, as it is considered obsolete and may not be supported in future versions of Turbo C++.

## Using complex math

---

Complex numbers are numbers of the form  $x + yi$ , where  $x$  and  $y$  are real numbers, and  $i$  is the square root of  $-1$ . Turbo C++ has always had a type

```

struct complex
{
    double x, y;
};

```

defined in `math.h`. This type is convenient for holding complex numbers, as they can be considered a pair of real numbers. However, the limitations of C make arithmetic with complex numbers rather cumbersome. With the addition of C++, complex math is much simpler.

*See the description of class **complex** in the Library Reference for more information.*

To use complex numbers in C++, all you have to do is to include `complex.h`. In `complex.h`, all the following have been overloaded to handle complex numbers:

- all of the usual arithmetic operators
- the stream operators, `>>` and `<<`
- the usual math functions, such as **sqrt** and **log**

The complex library is invoked only if the argument is of type **complex**. Thus, to get the complex square root of  $-1$ , use

```
sqrt(complex(-1))
```

and not

```
sqrt(-1)
```

As an example of the use of complex numbers, the following function computes a complex Fourier transform.

```
#include <complex.h>
```

```

// calculate the discrete Fourier transform of a[0], ..., a[n-1].
void Fourier(int n, complex a[], complex b[])
{
    int j, k;
    complex i(0,1);          // square root of -1
    for (j = 0; j < n; ++j)
    {
        b[j] = 0;
        for (k = 0; k < n; ++k)
            b[j] += a[k] * exp(2*M_PI*j*k*i/n);
        b[j] /= sqrt(n);
    }
}

```

---

## Using BCD math

Turbo C++, along with almost every other computer and compiler, does arithmetic on binary numbers (that is, base 2). This is sometimes confusing to people who are used to decimal (base 10) representations. Many numbers that are exactly representable in base 10, such as 0.01, can only be approximated in base 2.

Binary numbers are preferable for most applications, but in some situations the roundoff error involved in converting between base 2 and 10 is undesirable. The most common case is a financial or accounting application, where the pennies are supposed to add up. Consider the following program to add up 100 pennies and subtract a dollar:

```

#include <stdio.h>
int i;
float x = 0.0;
for (i = 0; i < 100; ++i)
    x += 0.01;
x -= 1.0;
printf("100*.01 - 1 = %g\n", x);

```

The correct answer is 0.0, but the computed answer is a small number close to 0.0. The computation magnifies the tiny roundoff error that occurs when converting 0.01 to base 2. Changing the type of *x* to **double** or **long double** reduces the error, but does not eliminate it.

To solve this problem, Turbo C++ offers the C++ type **bcd**, which is declared in `bcd.h`. With **bcd**, the number 0.01 is represented exactly, and the **bcd** variable *x* will give an exact penny count.

```

#include <bcd.h>

```

```

int i;
bcd x = 0.0;
for (i = 0; i < 100; ++i)
    x += 0.01;
x -= 1.0;
cout << "100*.01 - 1 = " << x << "\n";

```

Here are some facts to keep in mind about **bcd**.

- **bcd** does not eliminate all roundoff error: A computation like 1.0/3.0 will still have roundoff error.
- The usual math functions, such as **sqrt** and **log**, have been overloaded for **bcd** arguments.
- BCD numbers have about 17 decimal digits precision, and a range of about  $1 \times 10^{-125}$  to  $1 \times 10^{125}$ .

### Converting BCD numbers

**bcd** is a defined type distinct from **float**, **double**, or **long double**; decimal arithmetic is only performed when at least one operand is of the type **bcd**.

#### *Important!*

The **bcd** member function **real** is available for converting a **bcd** number back to one the usual base 2 formats (**float**, **double**, or **long double**), though the conversion is not done automatically. **real** does the necessary conversion to **long double**, which can then be converted to other types using the usual C conversions. For example,

```
bcd a = 12.1;
```

can be printed using any of the following four lines of code:

```

double x = a; printf("a = %g", x);
printf("a = %Lg", real(a));
printf("a = %g", (double)real(a));
cout << "a = " << a;

```

Note that since **printf** does not do argument checking, the format specifier must have the *L* if the **long double** value **real(a)** is passed.

### Number of decimal digits

You can specify how many decimal digits after the decimal point are to be carried in a conversion from a binary type to a **bcd**. The number of places is an optional second argument to the constructor **bcd**. For example, to convert  $\$1000.00/7$  to a **bcd** variable rounded to the nearest penny, use

```
bcd a = bcd(1000.00/7, 2)
```

where 2 indicates two digits following the decimal point. Thus,

```
1000.00/7           = 142.85714...
bcd(1000.00/7, 2)   = 142.860
bcd(1000.00/7, 1)   = 142.900
bcd(1000.00/7, 0)   = 143.000
bcd(1000.00/7, -1)  = 140.000
bcd(1000.00/7, -2)  = 100.000
```

*This method of rounding is specified by IEEE.*

The number is rounded using banker's rounding, which means round to the nearest whole number, with ties being rounded to an even digit. For example,

```
bcd(12.335, 2)      = 12.34
bcd(12.345, 2)      = 12.34
bcd(12.355, 2)      = 12.36
```

## Turbo C++'s use of RAM

---

Turbo C++ does not generate any intermediate data structures to disk when it is compiling (Turbo C++ writes only .OBJ files to disk); instead it uses RAM for intermediate data structures between passes. Because of this, you might encounter the message "Out of memory..." if there is not enough memory available for the compiler.

The solution to this problem is to make your functions smaller, or to split up the file that has large functions. You might also delete any RAM-resident programs you have installed to free up more memory for Turbo C++ to use.

## Overlays (VROOMM)

---

*Overlays* are parts of a program's code that share a common memory area. Only the parts of the program that are required for a given function reside in memory at the same time.

Overlays can significantly reduce a program's total run-time memory requirements. With overlays, you can execute programs that are much larger than the total available memory, since only parts of the program reside in memory at any given time.

## How overlays work

---

Turbo C++'s overlay manager (called VROOMM for Virtual Runtime Object-Oriented Memory Manager) is highly sophisticated; it does much of the work for you. In a conventional overlay system, modules are grouped together into a base and a set of overlay units. Routines in a given overlay unit may call other routines in the same unit and routines in the base, but not routines in other units. The overlay units are overlaid against each other; that is, only one overlay unit may be in memory at a time, and they each occupy the same physical memory. The total amount of memory needed to run the program is the size of the base plus the size of the largest overlay.

This conventional scheme is quite inflexible. It requires complete understanding of the possible calling dependencies in the program, and requires you to have the overlays grouped accordingly. It may be impossible to break your program into overlays if you can't split it into separable calling dependencies.

VROOMM's scheme is quite different. It provides *dynamic segment swapping*. The basic swapping unit is the segment. A segment can be one or more modules. More importantly, any segment can call *any other* segment.

Memory is divided into an area for the base plus a swap area. Whenever a function is called in a segment that is neither in the base nor in the swap area, the segment containing the called function is brought into the swap area, possibly displacing other segments. This is a powerful approach—it is like software virtual memory. You no longer have to break your code into static, distinct, overlay units. You just let it run!

What happens when a segment needs to be brought into the swap area? If there is room for the segment, execution just continues. If there is not, then one or more segments in the swap area must be thrown out to make room. How to decide which segment to throw out? The actual algorithm is quite sophisticated. A simplified version: If there is an inactive segment, choose it for removal. Inactive segments are those without executing functions. Otherwise, pick an active segment and toss it out. Keep tossing out segments until there is enough room available. This technique is called *dynamic swapping*.

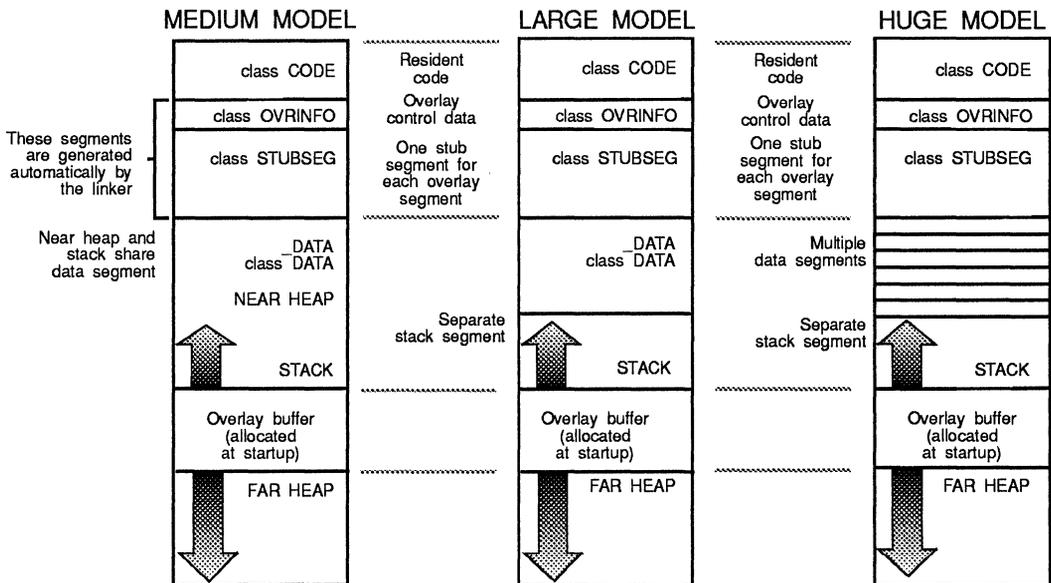
The more memory you provide for the swap area, the better the program performs. The swap area acts like a cache; the bigger the cache, the faster the program runs. The best setting for the size of the swap area is the size of the program's *working set*.

Once an overlay is loaded into memory, it is placed in the overlay buffer, which resides in memory between the stack segment and the far heap. By default, the size of the overlay buffer is estimated and set at startup, but you can change it using the global variable `_ovrbuffer`. If enough memory isn't available, an error message will be displayed by DOS ("Program too big to fit in memory") or by the C startup code ("Not enough memory to run program").

One very important option of the overlay manager is the ability to swap the modules to expanded or extended memory when they are discarded from the overlay buffer. Next time the module is needed, the overlay manager can copy it from where the module was swapped to instead of reading from the file. This makes it much faster.

When using overlays, memory is used as shown in the next figure.

Figure 4.9: Memory maps for overlays



## Getting the best out of Turbo C++ overlays

To get the best out of Turbo C++ overlays,

- Minimize resident code (resident run-time library, interrupt handlers, and device drivers is a good starting point).
- Set overlay pool size to be a comfortable working set (start with 128K and adjust up and down to see the speed/size tradeoff).
- Think versatility and variety: Take advantage of the overlay system to provide support for special cases, interactive help, and other end-user benefits you could not consider before.

---

## Requirements

In order to create overlays, you'll need to remember a few simple rules,

- The smallest part of a program that can be made into an overlay is a segment.
- Overlaid applications must use the medium, large, or huge programming models; the tiny, small, and compact models are not supported.
- Normal segment merging rules govern overlaid segments. That is, several .OBJ modules can contribute to the same overlaid segment.

The link-time generation of overlays is completely separated from the run-time overlay management; the linker does *not* automatically include code to manage the overlays. In fact, from the linker's point of view, the overlay manager is just another piece of code that gets linked in. The only assumption the linker makes is that the overlay manager takes over an interrupt vector (typically INT 3FH) through which all dynamic loading is controlled. This level of transparency makes it very easy to implement custom-built overlay managers that suit the particular needs of each application.

---

## Using overlays

To overlay a program, all of its modules must be compiled with the `-Y` compiler option enabled. To make a particular module into an overlay, it needs to be compiled with the `-Yo` option. (`-Yo` automatically enables `-Y`.)

The **-Yo** option applies to all modules and libraries that follow it on the TCC command line; you can disable it with **-Yo-**. These are the only command line options that are allowed to follow file names. For example, to overlay the module OVL.C but not the library GRAPHICS.LIB, either of the following command lines could be used:

```
TCC -ml -Yo ovl.c -Yo- graphics.lib
```

or

```
TCC -ml graphics.lib -Yo ovl.c
```

If TLINK is invoked explicitly to link the .EXE file, the **b** linker option must be specified on the linker command line or response file. Read the section on TLINK in Chapter 5, “Utilities” (in the *User’s Guide*) for how to use the **b** option.

### Overlay example

Suppose that you want to overlay a program consisting of three modules: MAIN.C, O1.C, and O2.C. Only the modules O1.C and O2.C should be made into overlays. (MAIN.C contains time-critical routines and interrupt handlers, so it should stay resident.) Let’s assume that the program uses the large memory model.

The following command accomplishes the task:

```
TCC -ml -Y main.c -Yo o1.c o2.c
```

The result will be an executable file MAIN.EXE, containing two overlays.

### Overlaying in the IDE

In order to overlay modules in the IDE, the following steps must be taken:

1. Go into **Options | Compiler | Code generation** dialog and set checkbox **Overlays On**.
2. Go into **Options | Linker** and set **Overlay** checkbox *On*.
3. In the project manager, use project item **Options** to specify each module that needs to go into an overlay.

The first step is the integrated environment’s equivalent of the command-line compiler’s **-Y** option. If this check box is not on, the two other options cannot be used. The second step controls whether the overlay information is being used in the IDE link process. By turning this option off, you can globally turn overlays off without needing to recompile or change any individual module settings in the project manager. The third checkbox controls which

modules go into overlays and which remain fixed. This checkbox closely resembles the command-line compiler's `-Yo` switch.

➡ If you are building an .EXE file containing overlays, compile all modules with the **Code Generation | Overlays** switch *On* (be sure you've selected **Options | Full Menus** *On* first).

➡ No module going into an overlay should ever change the default Code Class name. The IDE lets you change the set of modules residing in overlays without having to worry about recompiling. This can only be accomplished (with current .OBJ information) if overlays keep default code class names.

---

## Designing overlaid programs

This section provides some important information on designing programs with overlays. Look it over carefully, since a number of the issues discussed are vital to well-behaved overlaid applications.

### The far call requirement

Use a large code model (medium, large, or huge) when you want to compile an overlay module. At any call to an overlaid function in another module, you *must* guarantee that all currently active functions are far.

You *must* compile all overlaid modules with the `-Y` option, which causes the compiler to ensure that the generated code can be overlaid.

**Important!** Failing to observe the far call requirement in an overlaid program will cause unpredictable and possibly catastrophic results when the program is executed.

### Buffer size

The default overlay buffer size is twice the size of the largest overlay. This is adequate for some applications. But imagine a situation where a particular function of a program is implemented through many modules, each of which is overlaid. If the total size of those modules is larger than the overlay buffer, a substantial amount of swapping will occur if the modules make frequent calls to each other.

Obviously, the solution is to increase the size of the overlay buffer so that enough memory is available at any given time to contain all overlays that make frequent calls to each other. You can do this by setting the `_ovrbuffer` global variable to the required size in

paragraphs. For example, to set the overlay buffer to 128K, include the following statement in your code:

```
unsigned _ovrbuffer = 0x2000;
```

There is no general formula for determining the ideal overlay buffer size. Borland's Turbo Profiler can help provide a suitable value. If you don't have Turbo Profiler, a good knowledge of the application and a bit of experimenting will provide a suitable value.

**What not to overlay** Don't overlay modules that contain interrupt handlers, or small and time-critical routines. Due to the non-reentrant nature of the DOS operating system, modules that may be called by interrupt functions should not be overlaid.

Turbo C++'s overlay manager fully supports passing overlaid functions as arguments, assigning and initializing function pointer variables with addresses of overlaid functions, and calling overlaid routines via function pointers.

**Debugging overlays** Most debuggers have very limited overlay debugging capabilities, if any at all. Not so with Turbo C++'s integrated debugger and Turbo Debugger, the standalone debugger. Both debuggers fully support single-stepping and breakpoints in overlays in a manner completely transparent to you. By using overlays, you can easily engineer and debug huge applications—all from inside the IDE or by using Turbo Debugger.

**External routines in overlays** Like normal C functions, **external** assembly language routines must observe certain programming rules to work correctly with the overlay manager.

If an assembly language routine makes calls to *any* overlaid functions, the assembly language routine *must* be declared FAR, and it *must* set up a stack frame using the BP register. For example, assuming that *OtherFunc* is an overlaid function in another module, and that the assembly language routine *ExternFunc* calls it, then *ExternFunc* must be FAR and set up a stack frame, as the following demonstrates:

```
ExternFunc      PROC      FAR
                push      bp                ;Save BP
                mov       bp,sp            ;Set up stack frame
                sub       sp,LocalSize     ;Allocate local variables
```

```

    . . .
    call OtherFunc      ;Call another overlaid module
    . . .
    mov  sp, bp        ;Dispose local variables
    pop  bp            ;Restore BP
    RET                ;Return
ExternFunc            ENDP

```

where *LocalSize* is the size of the local variables. If *LocalSize* is zero, you can omit the two lines to allocate and dispose local variables, but you must not omit setting up the BP stack frame even if you have no arguments or variables on the stack.

These requirements are the same if *ExternFunc* makes *indirect* references to overlaid functions. For example, if *OtherFunc* makes calls to overlaid functions, but is not itself overlaid, *ExternFunc* must be FAR and still has to set up a stack frame.

In the case where an assembly language routine doesn't make any direct or indirect references to overlaid functions, there are no special requirements; the assembly language routine can be declared NEAR. It does not have to set up a stack frame.

Overlaid assembly language routines should *not* create variables in the code segment, since any modifications made to an overlaid code segment are lost when the overlay is disposed. Likewise, pointers to objects based in an overlaid code segment cannot be expected to remain valid across calls to other overlays, since the overlay manager freely moves around and disposes overlaid code segments.

---

## Swapping

If you have expanded or extended memory available, you can tell the overlay manager to use it for swapping. If you do so, when the overlay manager has to discard a module from the overlay buffer (because it should load a new module and the buffer is full), it can store the discarded module in this memory. Any later loading of this module is reduced to in-memory transfer, which is significantly faster than reading from a disk file.

In both cases there are two possibilities: The overlay manager can either detect the presence of expanded or extended memory and can take it over by itself, or it can use an already detected and allocated portion of the expanded/extended memory. In the case of extended memory, the detection of the memory usage is not

always successful because of the multitude of different cache and RAM disk programs that can take over extended memory without any mark. To avoid this problem, you can tell the overlay manager the starting address of the extended memory and how much of it is safe to use.

Expanded memory    The **\_OvrInitEms** function initializes expanded memory swapping. Here's its prototype:

**\_OvrInitEms** and **\_OvrInitExt**  
are defined in *dos.h*.

```
extern int far _OvrInitEms
(
    unsigned emsHandle,
    unsigned emsFirst,
    unsigned emsPages
);
```

If the *emsHandle* parameter is zero, the overlay manager checks for the presence of expanded memory and allocates the amount (if it can) that can contain all of the overlays minus the size of the overlay buffer. Otherwise, *emsHandle* should be a legal EMS handle, *emsFirst* is the first usable EMS page, and *emsPages* is the number of pages usable by the overlay manager. This function returns 0 if expanded memory is available.

Extended memory    The **\_OvrInitExt** function initializes extended memory swapping. Here's its prototype:

```
extern int far _OvrInitExt
(
    unsigned long extStart,
    unsigned long extLength
);
```

If the *extStart* parameter is zero, the overlay manager checks for extended memory. If it can, the overlay manager uses the amount of free memory that can contain all of the overlays minus the size of the overlay buffer. Otherwise, *extStart* is the start of the usable extended memory, with *extLength* bytes usable by the overlay manager. If *extLength* is zero, the overlay manager will use all available extended memory above *extStart*. This function returns 0 if extended memory is available. **\_OvrInitExt** is defined in *dos.h*.

**Important!** The use of extended memory is not standardized. Though the overlay manager tries every known method to find out the amount of extended memory which is already used, use this function carefully. For example, if you have a 2 Mb hard disk cache

program installed (that uses extended memory), you could use the following call to let the overlay manager use the remaining extended memory:

```
if (_OvrInitExt (1024L * (2048 + 1024), 0L))
    puts ("No extended memory available for overlay swapping");
```

## *Video functions*

Turbo C++ comes with a complete library of graphics functions, so you can produce onscreen charts and diagrams. This chapter first briefly discusses video modes and windows. Then it explains how to program in text mode and in graphics mode.

Turbo C++'s video functions are similar to corresponding routines in Turbo Pascal. If you are not already familiar with controlling your PC's screen modes or creating and managing windows and viewports, take a few minutes to read the following words on those topics.

### Some words about video modes

Your PC has some kind of video adapter. This can be a Monochrome Display Adapter (MDA) for your basic text-only display, or it can be capable of displaying graphics, such as a Color/Graphics Adapter (CGA), an Enhanced Graphics Adapter (EGA), or a Hercules Monochrome Graphics Adapter. Each adapter can operate in a variety of modes; the mode specifies whether the screen displays 80 or 40 columns (text mode only), the display resolution (graphics mode only), and the display type (color or black and white).

The screen's operating mode is defined when your program calls one of the mode-defining functions (**textmode**, **initgraph**, or **setgraphmode**).

- In *text mode*, your PC's screen is divided into cells (80- or 40-columns wide by 25, 43, or 50 lines high). Each cell consists of an attribute and a character. The character is the displayed ASCII character, while the attribute specifies *how* the character is displayed (its color, intensity, and so on). Turbo C++ provides a full range of routines for manipulating the text screen, for writing text directly to the screen, and for controlling the cell attributes.
- In *graphics mode*, your PC's screen is divided into pixels; each pixel displays a single dot onscreen. The number of pixels (the resolution) depends on the type of video adapter connected to your system and the mode that adapter is in. You can use functions from Turbo C++'s graphics library to create graphic displays onscreen: You can draw lines and shapes, fill enclosed areas with patterns, and control the color of each pixel.

In text modes, the upper left corner of the screen is position (1,1), with x-coordinates increasing from left to right, and y-coordinates increasing from screen-top to screen-bottom. In graphics modes, the upper left corner is position (0,0), with the x- and y-coordinate values increasing in the same manner.

## Some words about windows and viewports

---

Turbo C++ provides functions for creating and managing windows on your screen in text mode (and viewports in graphics mode). If you are not familiar with windows and viewports, you should read this brief overview. Turbo C++'s window- and viewport-management functions are explained in "Programming in text mode" and "Programming in graphics mode" later in this chapter.

### What is a window?

---

A window is a rectangular area defined on your PC's video screen when it's in a text mode. When your program writes to the screen, its output is restricted to the active window. The rest of the screen (outside the window) remains untouched.

The default window is a full-screen text window. Your program can change this default full-screen text window to a text window smaller than the full screen (with a call to the **window** function).

This function specifies the window's position in terms of screen coordinates.

---

## What is a viewport?

In graphics mode, you can also define a rectangular area on your PC's video screen; this is a viewport. When your graphics program outputs drawings and so on, the viewport acts as the virtual screen. The rest of the screen (outside the viewport) remains untouched. You define a viewport in terms of screen coordinates with a call to the **setviewport** function.

---

## Coordinates

Except for these window- and viewport-defining functions, all coordinates for text-mode and graphics-mode functions are given in window- or viewport-relative terms, not in absolute screen coordinates. The upper left corner of the text-mode window is the coordinate origin, referred to as (1,1); in graphics modes, the viewport coordinate origin is position (0,0).

## Programming in text mode

---

In this section, we give a brief summary of the functions you use in text mode. For more detailed information about these functions, refer to Chapter 1, "The run-time library," of the *Library Reference*.

In Turbo C++, the direct console I/O package (**cprintf**, **cputs**, and so on) provides high-performance text output, window management, cursor positioning, and attribute control functions. These functions are all part of the standard Turbo C++ libraries; they are prototyped in the header file `conio.h`.

## The console I/O functions

Turbo C++'s text-mode functions work in any of the six possible video text modes. The modes available on your system depend on the type of video adapter and monitor you have. You specify the current text mode with a call to **textmode**. We explain how to use this function later in this chapter and under the **textmode** entry in Chapter 1 in the *Library Reference*.

These text mode functions are divided into four separate groups:

- text output and manipulation
- window and mode control
- attribute control
- state query

We cover these four text mode function groups in the following sections.

## Text output and manipulation

Here's a quick summary of the text output and manipulation functions:

Writing and reading text:

<b>cprintf</b>	Sends formatted output to the screen.
<b>cputs</b>	Sends a string to the screen.
<b>getche</b>	Reads a character and echoes it to the screen.
<b>putch</b>	Sends a single character to the screen.

Manipulating text (and the cursor) onscreen:

<b>clr_EOL</b>	Clears from the cursor to the end of the line.
<b>clrscr</b>	Clears the text window.
<b>delline</b>	Deletes the line where the cursor rests.
<b>gotoxy</b>	Positions the cursor.
<b>insline</b>	Inserts a blank line below the line where the cursor rests.
<b>movetext</b>	Copies text from one area onscreen to another.

Moving blocks of text into and out of memory:

<b>gettext</b>	Copies text from an area onscreen to memory.
<b>puttext</b>	Copies text from memory to an area onscreen.

Your screen-output programs will come up in a full-screen text window by default, so you can immediately write, read, and manipulate text without any preliminary mode-setting. You write text to the screen with the direct console output functions **cprintf**, **cputs**, and **putch**, and echo input with the function **getche**. Text wrapping is controlled by the global variable `_wscroll`. If `_wscroll` is 1, text wraps onto the next line, scrolling as necessary. If `_wscroll` is 0, text wraps onto the same line, and there is no scrolling. `_wscroll` is 1 by default.

Once your text is on the screen, you can erase the active window with **clrscr**, erase part of a line with **clr\_EOL**, delete a whole line with **delline**, and insert a blank line with **insline**. The latter three

functions operate relative to the cursor position; you move the cursor to a specified location with **gotoxy**. You can also copy a whole block of text from one rectangular location in the window to another with **movetext**.

You can capture a rectangle of onscreen text to memory with **gettext**, and put that text back on the screen (anywhere you want) with **puttext**.

#### Window and mode control

There are two window- and mode-control functions:

<b>textmode</b>	Sets the screen to a text mode.
<b>window</b>	Defines a text-mode window.

You can set your screen to any of several video text modes with **textmode** (limited only by your system's type of monitor and adapter). This initializes the screen as a full-screen text window, in the particular mode specified, and clears any residual images or text.

When your screen is in a text mode, you can output to the full screen, or you can set aside a *portion* of the screen—a window—to which your program's output is confined. To create a text window, you call **window**, specifying what area on the screen it will occupy.

#### Attribute control

Here's a quick summary of the text-mode attribute control functions:

Setting foreground and background:

<b>textattr</b>	Sets the foreground and background colors (attributes) at the same time.
<b>textbackground</b>	Sets the background color (attribute).
<b>textcolor</b>	Sets the foreground color (attribute).

Modifying intensity:

<b>highvideo</b>	Sets text to high intensity.
<b>lowvideo</b>	Sets text to low intensity.
<b>normvideo</b>	Sets text to original intensity.

The attribute control functions set the current attribute, which is represented by an 8-bit value: The four lowest bits represent the foreground color, the next three bits give the background color, and the high bit is the "blink enable" bit.

Subsequent text is displayed in the current attribute. With the attribute control functions, you can set the background and foreground (character) colors separately (with **textbackground** and **textcolor**) or combine the color specifications in a single call to **textattr**. You can also specify that the character (the foreground) will blink. Most color monitors in color modes will display the true colors. Non-color monitors may convert some or all of the attributes to various monochromatic shades or other visual effects, such as bold, underscore, reverse video, and so on.

You can direct your system to map the high-intensity foreground colors to low-intensity colors with **lowvideo** (which turns off the high-intensity bit for the characters). Or you can map the low-intensity colors to high intensity with **highvideo** (which turns on the character high-intensity bit). When you're through playing around with the character intensities, you can restore the settings to their original values with **normvideo**.

State query Here's a quick summary of the state-query functions:

- |                    |   |
|--------------------|---|
| <b>gettextinfo</b> | Fills in a <b>text_info</b> structure with information about the current text window. |
| <b>wherex</b>      | Gives the x-coordinate of the cell containing the cursor.                             |
| <b>wherey</b>      | Gives the y-coordinate of the cell containing the cursor.                             |

Turbo C++'s console I/O functions include some designed for *state queries*. With these functions, you can retrieve information about your text-mode window and the current cursor position within the window.

The **gettextinfo** function fills a **text\_info** structure (defined in `conio.h`) with several details about the text window, including:

- the current video mode
- the window's position in absolute screen coordinates
- the window's dimensions
- the current foreground and background colors
- the cursor's current position

Sometimes you might need only a few of these details. Rather than retrieving all the text window information, you can find out just the cursor's (window-relative) position with **wherex** and **wherey**.

Cursor shape You can use the new function `_setcursortype` to change the appearance of your cursor. The values are `_NOCURSOR`, which turns off the cursor; `_SOLIDCURSOR`, which gives you a solid block (large) cursor; and `_NORMALCURSOR`, which gives you the normal underscore cursor.

---

## Text windows

The default text window is full screen; you can change this to a less-than-full-screen text window with a call to the `window` function. Text windows can contain up to 50 lines and up to 40 or 80 columns.

The coordinate origin (point where the numbers start) of a Turbo C++ text window is the upper left corner of the window. The coordinates of the window's upper left corner are (1,1); the coordinates of the bottom right corner of a full-screen 80-column, 25-line text window are (80,25).

An example Suppose your 100% PC-compatible system is in 80-column text mode, and you want to create a window. The upper left corner of the window will be at screen coordinates (10, 8), and the lower right corner of the window will be at screen coordinates (50, 21). To do this, you call the `window` function, like this:

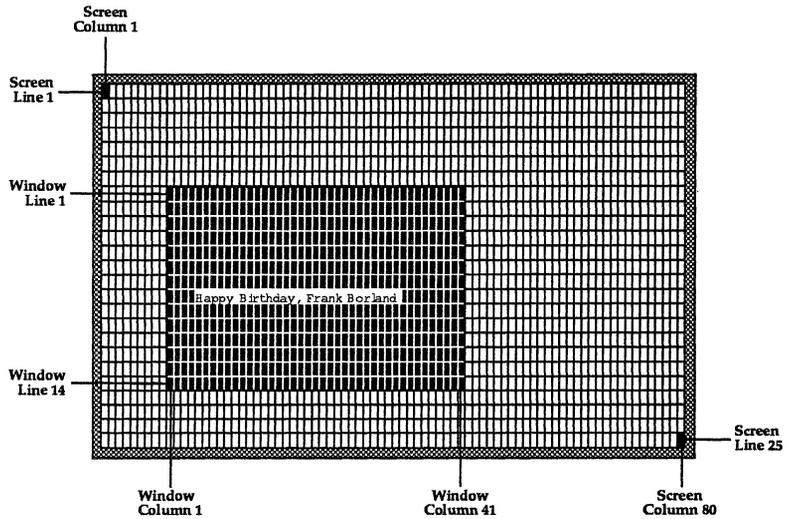
```
window(10, 8, 50, 21);
```

Now that you've created the text-mode window, you want to move the cursor to the `window` position (5, 8) and write some text in it, so you decide to use `gotoxy` and `cputs`.

```
gotoxy(5, 8);  
cputs("Happy Birthday, Frank Borland");
```

The next figure illustrates these ideas.

Figure 5.1  
A window in 80x25 text mode



## The `text_modes` type

You can put your monitor into one of seven PC text modes with a call to the `textmode` function. The enumeration type `text_modes`, defined in `conio.h`, enables you to use symbolic names for the `mode` argument to the `textmode` function, instead of “raw” mode numbers. However, if you use the symbolic constants, you must put

```
#include <conio.h>
```

in your source code.

The numeric and symbolic values defined by `text_modes` are as follows:

Symbolic constant	Numeric value	Video text mode
LASTMODE	-1	Previous text mode enabled
BW40	0	Black and white, 40 columns
C40	1	16-color, 40 columns
BW80	2	Black and white, 80 columns
C80	3	16-color, 80 columns
MONO	7	Monochrome, 80 columns
C4350	64	EGA, 80x43; VGA, 80x50 lines

For example, the following calls to `textmode` put your color monitor in the indicated operating mode:

<code>textmode(0)</code>	Black and white, 40 column
<code>textmode(BW80)</code>	Black and white, 80 column
<code>textmode(C40)</code>	16-color, 40 column
<code>textmode(3)</code>	16-color, 80 column
<code>textmode(7)</code>	Monochrome, 80 columns
<code>textmode(C4350)</code>	EGA, 80x43; VGA, 80x50 lines

Use **settextinfo** to determine the number of rows in the screen after calling **textmode** in the mode C4350.

## Text colors

---

For a detailed description of how cell attributes are laid out, refer to the **textattr** entry in Chapter 1 of the *Library Reference*.

When a character occupies a cell, the color of the character is the *foreground*; the color of the cell's remaining area is the *background*. Color monitors with color video adapters can display up to 16 different colors; monochrome monitors substitute different visual attributes (highlighted, underscored, reverse video, and so on) for the colors.

The include file `conio.h` defines symbolic names for the different colors. If you use the symbolic constants, you must include `conio.h` in your source code.

The following table lists these symbolic constants and their corresponding numeric values. Note that only the first eight colors are available for the foreground and background; the last eight (colors 8 through 15) are available for the foreground (the characters themselves) only.

Symbolic constant	Numeric value	Foreground or background?
BLACK	0	Both
BLUE	1	Both
GREEN	2	Both
CYAN	3	Both
RED	4	Both
MAGENTA	5	Both
BROWN	6	Both
LIGHTGRAY	7	Both
DARKGRAY	8	Foreground only
LIGHTBLUE	9	Foreground only
LIGHTGREEN	10	Foreground only
LIGHTCYAN	11	Foreground only
LIGHTRED	12	Foreground only
LIGHTMAGENTA	13	Foreground only
YELLOW	14	Foreground only
WHITE	15	Foreground only
BLINK	128	Foreground only

You can add the symbolic constant BLINK (numeric value 128) to a foreground argument if you want the character to blink.

### High-performance output: The *directvideo* variable

Turbo C++'s console I/O package includes a variable called *directvideo*. This variable controls whether your program's console output goes directly to the video RAM (*directvideo* = 1) or goes via BIOS calls (*directvideo* = 0).

The default value is *directvideo* = 1 (console output goes directly to the video RAM). In general, going directly to video RAM gives very high performance (spelled f-a-s-t-e-r o-u-t-p-u-t), but doing so requires your computer to be 100% IBM PC-compatible: Your video hardware must be identical to IBM display adapters. Setting *directvideo* = 0 will work on any machine that is IBM BIOS-compatible, but the console output will be slower.

## Programming in graphics mode

In this section, we give a brief summary of the functions you use in graphics mode. For more detailed information about these functions, refer to Chapter 1 of the *Library Reference*.

Turbo C++ provides a separate library of over 70 graphics functions, ranging from high-level calls (like **setviewport**, **bar3d**, and **drawpoly**) to bit-oriented functions (like **getimage** and **putimage**). The graphics library supports numerous fill and line styles, and provides several text fonts that you can size, justify, and orient horizontally or vertically.

These functions are in the library file GRAPHICS.LIB, and they are prototyped in the header file graphics.h. In addition to these two files, the graphics package includes graphics device drivers (\*.BGI files) and stroked character fonts (\*.CHR files); we discuss these additional files in following sections.

In order to use the graphics functions:

- If you're using the integrated environment, toggle Full Menus to *On*, then check **Options | Linker | Graphics Library**. When you make your program, the linker automatically links in the Turbo C++ graphics library.
- If you're using TCC.EXE, you have to list GRAPHICS.LIB on the command line. For example, if your program, MYPROG.C, uses graphics, the TCC command line would be

```
tcc myprog graphics.lib
```

**Important!** Because graphics functions use **far** pointers, graphics are not supported in the tiny memory model.

There is only one graphics library, not separate versions for each memory model (in contrast to the standard libraries CS.LIB, CC.LIB, CM.LIB, etc., which are memory-model specific). Each function in GRAPHICS.LIB is a **far** function, and those graphics functions that take pointers take **far** pointers. For these functions to work correctly, it is important that you `#include graphics.h` in every module that uses graphics.

## The graphics library functions

---

Turbo C++'s graphics functions fall into seven categories:

- graphics system control
- drawing and filling
- manipulating screens and viewports
- text output
- color control
- error handling
- state query

Graphics system control

Here's a quick summary of the graphics system control:

<b>closegraph</b>	Shuts down the graphics system.
<b>detectgraph</b>	Checks the hardware and determines which graphics driver to use; recommends a mode.
<b>graphdefaults</b>	Resets all graphics system variables to their default settings.
<b>_graphfreemem</b>	Deallocates graphics memory; hook for defining your own routine.
<b>_graphgetmem</b>	Allocates graphics memory; hook for defining your own routine.
<b>getgraphmode</b>	Returns the current graphics mode.
<b>getmoderange</b>	Returns lowest and highest valid modes for specified driver.
<b>initgraph</b>	Initializes the graphics system and puts the hardware into graphics mode.
<b>installuserdriver</b>	Installs a vendor-added device driver to the BGI device driver table.
<b>installuserfont</b>	Loads a vendor-added stroked font file to the BGI character file table.
<b>registerbgidriver</b>	Registers a linked-in or user-loaded driver file for inclusion at link time.
<b>restorecrtmode</b>	Restores the original (pre- <b>initgraph</b> ) screen mode.
<b>setgraphbufsize</b>	Specifies size of the internal graphics buffer.
<b>setgraphmode</b>	Selects the specified graphics mode, clears the screen, and restores all defaults.

Turbo C++'s graphics package provides graphics drivers for the following graphics adapters (and true compatibles):

- Color/Graphics Adapter (CGA)
- Multi-Color Graphics Array (MCGA)
- Enhanced Graphics Adapter (EGA)
- Video Graphics Array (VGA)
- Hercules Graphics Adapter
- AT&T 400-line Graphics Adapter
- 3270 PC Graphics Adapter
- IBM 8514 Graphics Adapter

To start the graphics system, you first call the **initgraph** function. **initgraph** loads the graphics driver and puts the system into graphics mode.

You can tell **initgraph** to use a particular graphics driver and mode, or to autodetect the attached video adapter at run time and pick the corresponding driver. If you tell **initgraph** to autodetect, it calls **detectgraph** to select a graphics driver and mode. If you tell **initgraph** to use a particular graphics driver and mode, you must be sure that the hardware is present. If you force **initgraph** to use hardware that is not present, the results will be unpredictable.

Once a graphics driver has been loaded, you can find out the name of the driver by using the **getdrivername** function and how many modes a driver supports with **getmaxmode**. **getgraphmode** will tell you which graphics mode you are currently in. Once you have a mode number, you can find out the name of the mode with **getmodename**. You can change graphics modes with **setgraphmode** and return the video mode to its original state (before graphics was initialized) with **restorecrtmode**. **restorecrtmode** returns the screen to text mode, but it does not close the graphics system (the fonts and drivers are still in memory).

**graphdefaults** resets the graphics state's settings (viewport size, draw color, fill color and pattern, etc.) to their default values.

**installuserdriver** and **installuserfont** let you add new device drivers and fonts to your BGI.

Finally, when you're through using graphics, call **closegraph** to shut down the graphics system. **closegraph** unloads the driver from memory and restores the original video mode (via **restorecrtmode**).

A more detailed discussion

The previous discussion provided an overview of how **initgraph** operates. In the following paragraphs, we describe the behavior of **initgraph**, **\_graphgetmem**, and **\_graphfreemem** in some detail.

Normally, the **initgraph** routine loads a graphics driver by allocating memory for the driver, then loading the appropriate .BGI file from disk. As an alternative to this dynamic loading scheme, you can link a graphics driver file (or several of them) directly into your executable program file. You do this by first converting the .BGI file to an .OBJ file (using the BGI OBJ utility—see UTIL.DOC, included with your distribution disks), then placing calls to **registerbgidriver** in your source code (before the call to **initgraph**)

to *register* the graphics driver(s). When you build your program, you need to link the .OBJ files for the registered drivers.

After determining which graphics driver to use (*via* **detectgraph**), **initgraph** checks to see if the desired driver has been registered. If so, **initgraph** uses the registered driver directly from memory. Otherwise, **initgraph** allocates memory for the driver and loads the .BGI file from disk.

**Note** Using **registerbgidriver** is an advanced programming technique, not recommended for novice programmers. This function is described in more detail in Chapter 1 in the *Library Reference*.

During run time, the graphics system might need to allocate memory for drivers, fonts, and internal buffers. If this is necessary, it calls **\_graphgetmem** to allocate memory, and calls **\_graphfreemem** to free it. By default, these routines simply call **malloc** and **free**, respectively.

*If you provide your own **\_graphgetmem** or **\_graphfreemem**, you may get a "duplicate symbols" warning message. Just ignore the warning.*

You can override this default behavior by defining your own **\_graphgetmem** and **\_graphfreemem** functions. By doing this, you can control graphics memory allocation yourself. You must, however, use the same names for your own versions of these memory-allocation routines: They will override the default functions with the same names that are in the standard C libraries.

Drawing and filling Here's a quick summary of the drawing and filling functions:

Drawing:

<b>arc</b>	Draws a circular arc.
<b>circle</b>	Draws a circle.
<b>drawpoly</b>	Draws the outline of a polygon.
<b>ellipse</b>	Draws an elliptical arc.
<b>getarccoords</b>	Returns the coordinates of the last call to <b>arc</b> or <b>ellipse</b> .
<b>getaspectratio</b>	Returns the aspect ratio of the current graphics mode.
<b>getlinesettings</b>	Returns the current line style, line pattern, and line thickness.
<b>line</b>	Draws a line from $(x_0,y_0)$ to $(x_1,y_1)$ .
<b>linere1</b>	Draws a line to a point some relative distance from the current position (CP).
<b>lineto</b>	Draws a line from the current position (CP) to $(x,y)$ .
<b>moveto</b>	Moves the current position (CP) to $(x,y)$ .

<b>moverel</b>	Moves the current position (CP) a relative distance.
<b>rectangle</b>	Draws a rectangle.
<b>setaspectratio</b>	Changes the default aspect ratio-correction factor.
<b>setlinestyle</b>	Sets the current line width and style.

Filling:

<b>bar</b>	Draws and fills a bar.
<b>bar3d</b>	Draws and fills a 3-D bar.
<b>fillellipse</b>	Draws and fills an ellipse.
<b>fillpoly</b>	Draws and fills a polygon.
<b>floodfill</b>	Flood-fills a bounded region.
<b>getfillpattern</b>	Returns the user-defined fill pattern.
<b>getfillsettings</b>	Returns information about the current fill pattern and color.
<b>pieslice</b>	Draws and fills a pie slice.
<b>sector</b>	Draws and fills an elliptical pie slice.
<b>setfillpattern</b>	Selects a user-defined fill pattern.
<b>setfillstyle</b>	Sets the fill pattern and fill color.

With Turbo C++'s drawing and painting functions, you can draw colored lines, arcs, circles, ellipses, rectangles, pie slices, two- and three-dimensional bars, polygons, and regular or irregular shapes based on combinations of these. You can fill any bounded shape (or any region surrounding such a shape) with one of 11 predefined patterns, or your own user-defined pattern. You can also control the thickness and style of the drawing line, and the location of the current position (CP).

You draw lines and unfilled shapes with the functions **arc**, **circle**, **drawpoly**, **ellipse**, **line**, **linerel**, **lineto**, and **rectangle**. You can fill these shapes with **floodfill**, or combine drawing/filling into one step with **bar**, **bar3d**, **fillellipse**, **fillpoly**, **pieslice**, and **sector**. You use **setlinestyle** to specify whether the drawing line (and border line for filled shapes) is thick or thin, and whether its style is solid, dotted, and so forth, or some other line pattern you've defined. You can select a predefined fill pattern with **setfillstyle**, and define your own fill pattern with **setfillpattern**. You move the CP to a specified location with **moveto**, and move it a specified displacement with **moverel**.

To find out the current line style and thickness, you call **getline-settings**. For information about the current fill pattern and fill

color, you call **getfillsettings**; you can get the user-defined fill pattern with **getfillpattern**.

You can get the aspect ratio (the scaling factor used by the graphics system to make sure circles come out round) with **getaspectratio**, and get coordinates of the last drawn arc or ellipse by calling **getarcoords**. If your circles are not perfectly round, use **setaspectratio** to correct them.

## Manipulating the screen and viewport

Here's a quick summary of the screen-, viewport-, image-, and pixel-manipulation functions:

Screen manipulation:

<b>cleardevice</b>	Clears the screen (active page).
<b>setactivepage</b>	Sets the active page for graphics output.
<b>setvisualpage</b>	Sets the visual graphics page number.

Viewport manipulation:

<b>clearviewport</b>	Clears the current viewport.
<b>getviewsettings</b>	Returns information about the current viewport.
<b>setviewport</b>	Sets the current output viewport for graphics output.

Image manipulation:

<b>getimage</b>	Saves a bit image of the specified region to memory.
<b>imagesize</b>	Returns the number of bytes required to store a rectangular region of the screen.
<b>putimage</b>	Puts a previously saved bit image onto the screen.

Pixel manipulation:

<b>getpixel</b>	Gets the pixel color at $(x,y)$ .
<b>putpixel</b>	Plots a pixel at $(x,y)$ .

Besides drawing and painting, the graphics library offers several functions for manipulating the screen, viewports, images, and pixels. You can clear the whole screen in one fell swoop with a call to **cleardevice**; this routine erases the entire screen and homes the CP in the viewport, but leaves all other graphics system settings intact (the line, fill, and text styles; the palette; the viewport settings; and so on).

Depending on your graphics adapter, your system has between one and four screen-page buffers, which are areas in memory where individual whole-screen images are stored dot-by-dot. You can specify which screen page is the active one (where graphics functions place their output) and which is the visual page (the one displayed onscreen) with **setactivepage** and **setvisualpage**, respectively.

Once your screen's in a graphics mode, you can define a viewport (a rectangular "virtual screen") on your screen with a call to **setviewport**. You define the viewport's position in terms of absolute screen coordinates and specify whether clipping is on (active) or off. You clear the viewport with **clearviewport**. To find out the current viewport's absolute screen coordinates and clipping status, call **getviewsettings**.

You can capture a portion of the onscreen image with **getimage**, call **imagesize** to calculate the number of bytes required to store that captured image in memory, then put the stored image back on the screen (anywhere you want) with **putimage**.

The coordinates for all output functions (drawing, filling, text, and so on) are viewport-relative.

You can also manipulate the color of individual pixels with the functions **getpixel** (which returns the color of a given pixel) and **putpixel** (which plots a specified pixel in a given color).

Text output in graphics mode

Here's a quick summary of the graphics-mode text output functions:

<b>gettextsettings</b>	Returns the current text font, direction, size, and justification.
<b>outtext</b>	Sends a string to the screen at the current position (CP).
<b>outtextxy</b>	Sends a string to the screen at the specified position.
<b>registerbgifont</b>	Registers a linked-in or user-loaded font.
<b>settextjustify</b>	Sets text justification values used by <b>outtext</b> and <b>outtextxy</b> .
<b>settextstyle</b>	Sets the current text font, style, and character magnification factor.
<b>setusercharsize</b>	Sets width and height ratios for stroked fonts.
<b>textheight</b>	Returns the height of a string in pixels.
<b>textwidth</b>	Returns the width of a string in pixels.

The graphics library includes an 8×8 bit-mapped font and several stroked fonts for text output while in graphics mode.

- In a *bit-mapped* font, each character is defined by a matrix of pixels.
- In a *stroked* font, each character is defined by a series of vectors that tell the graphics system how to draw that character.

The advantage of using a stroked font is apparent when you start to draw large characters. Since a stroked font is defined by vectors, it will still retain good resolution and quality when the font is enlarged. On the other hand, when you enlarge a bit-mapped font, the matrix is multiplied by a scaling factor; as the scaling factor becomes larger, the characters' resolution becomes coarser. For small characters, the bit-mapped font should be sufficient, but for larger text you should select a stroked font.

You output graphics text by calling either **outtext** or **outtextxy**, and control the justification of the output text (with respect to the CP) with **settextjustify**. You choose the character font, direction (horizontal or vertical), and size (scale) with **settextstyle**. You can find out the current text settings by calling **gettextsettings**, which returns the current text font, justification, magnification, and direction in a **textsettings** structure. **setusercharsize** allows you to modify the character width and height of stroked fonts.

If clipping is *on*, all text strings output by **outtext** and **outtextxy** will be clipped at the viewport borders. If clipping is *off*, these functions will throw away bit-mapped font output if any part of the text string would go off the screen edge; stroked font output is truncated at the screen edges.

To determine the onscreen size of a given text string, call **text-height** (which measures the string's height in pixels) and **textwidth** (which measures its width in pixels).

The default 8×8 bit-mapped font is built into the graphics package, so it is always available at run time. The stroked fonts are each kept in a separate .CHR file; they can be loaded at run time or converted to .OBJ files (with the BGIOBJ utility) and linked into your .EXE file.

Normally, the **settextstyle** routine loads a font file by allocating memory for the font, then loading the appropriate .CHR file from disk. As an alternative to this dynamic loading scheme, you can link a character font file (or several of them) directly into your executable program file. You do this by first converting the .CHR

file to an .OBJ file (using the BGIOBJ utility—read about it in UTIL.DOC, the online documentation included with your distribution disks), then placing calls to **registerbgifont** in your source code (before the call to **settextstyle**) to *register* the character font(s). When you build your program, you need to link in the .OBJ files for the stroked fonts you register.

*Note* Using **registerbgifont** is an advanced programming technique, not recommended for novice programmers. This function is described in more detail in UTIL.DOC, included with your distribution disks.

Color control Here's a quick summary of the color control functions:

Get color information:

<b>getbkcolor</b>	Returns the current background color.
<b>getcolor</b>	Returns the current drawing color.
<b>getdefaultpalette</b>	Returns the palette definition structure.
<b>getmaxcolor</b>	Returns the maximum color value available in the current graphics mode.
<b>getpalette</b>	Returns the current palette and its size.
<b>getpalettesize</b>	Returns the size of the palette look-up table.

Set one or more colors:

<b>setallpalette</b>	Changes all palette colors as specified.
<b>setbkcolor</b>	Sets the current background color.
<b>setcolor</b>	Sets the current drawing color.
<b>setpalette</b>	Changes one palette color as specified by its arguments.

Before summarizing how these color control functions work, we first present a basic description of how colors are actually produced on your graphics screen.

Pixels and palettes The graphics screen consists of an array of pixels; each pixel produces a single (colored) dot onscreen. The pixel's value does not specify the precise color directly; it is an index into a color table called a *palette*. The palette entry corresponding to a given pixel value contains the exact color information for that pixel.

This indirection scheme has a number of implications. Though the hardware might be capable of displaying many colors, only a subset of those colors can be displayed at any given time. The num-

ber of colors that can be displayed at any one time is equal to the number of entries in the palette (the palette's *size*). For example, on an EGA, the hardware can display 64 different colors, but only 16 of them at a time; the EGA palette's *size* is 16.

The *size* of the palette determines the range of values a pixel can assume, from 0 to (*size* - 1). The **getmaxcolor** function returns the highest valid pixel value (*size* - 1) for the current graphics driver and mode.

When we discuss the Turbo C++ graphics functions, we often use the term *color*, such as the current drawing color, fill color and pixel color. In fact, this color is a pixel's value: it's an index into the palette. Only the palette determines the true color on the screen. By manipulating the palette, you can change the actual color displayed on the screen even though the pixel values (drawing color, fill color, and so on) have not changed.

Background and drawing color

The *background color* always corresponds to pixel value 0. When an area is cleared to the background color, that area's pixels are simply set to 0.

The *drawing color* is the value to which pixels are set when lines are drawn. You choose a drawing color with `setcolor(n)`, where *n* is a valid pixel value for the current palette.

Color control on a CGA

Due to graphics hardware differences, how you actually control color differs quite a bit between the CGA and the EGA, so we'll present them separately. Color control on the AT&T driver, and the lower resolutions of the MCGA driver is similar to CGA color control.

On the CGA, you can choose to display your graphics in low resolution (320×200), which allows you to use four colors, or high resolution (640×200), in which you can use two colors.

#### **CGA low resolution**

In the low resolution modes, you can choose from four predefined four-color palettes. In any of these palettes, you can only set the first palette entry; entries 1, 2, and 3 are fixed. The first palette entry (color 0) is the background color. This background color can be any one of the 16 available colors (see table of CGA background colors below).

You choose which palette you want by the mode you select (CGAC0, CGAC1, CGAC2, CGAC3); these modes use color palette 0 through color palette 3, as detailed in the following table. The CGA drawing colors and the equivalent constants are defined in graphics.h.

Palette number	Constant assigned to color number (pixel value)		
	1	2	3
0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WHITE
2	CGA_GREEN	CGA_RED	CGA_BROWN
3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGRAY

To assign one of these colors as the CGA drawing color, call **setcolor** with either the color number or the corresponding constant name as an argument; for example, if you are using palette 3 and you want to use cyan as the drawing color:

```
setcolor(1);
```

or

```
setcolor(CGA_CYAN);
```

The available CGA background colors, defined in graphics.h, are listed in the following table:

Numeric value	Symbolic name	Numeric value	Symbolic name
0	BLACK	8	DARKGRAY
1	BLUE	9	LIGHTBLUE
2	GREEN	10	LIGHTGREEN
3	CYAN	11	LIGHTCYAN
4	RED	12	LIGHTRED
5	MAGENTA	13	LIGHTMAGENTA
6	BROWN	14	YELLOW
7	LIGHTGRAY	15	WHITE

*The CGA's foreground colors are the same as those listed in this table.*

To assign one of these colors to the CGA background color, use **setbkcolor**(*color*), where *color* is one of the entries in the preceding table. Note that for CGA, this color is not a pixel value (palette index); it directly specifies the *actual* color to be put in the first palette entry.

### CGA high resolution

In high resolution mode (640×200), the CGA displays two colors: a black background and a colored foreground. Pixels can take on values of either 0 or 1. Because of a quirk in the CGA itself, the foreground color is actually what the hardware thinks of as its background color; you set it with the **setbkcolor** routine. (Strange but true.)

The colors available for the colored foreground are those listed in the preceding table. The CGA uses this color to display all pixels whose value equals 1.

The modes that behave in this way are CGAHI, MCGAMED, MCGAHI, ATT400MED, and ATT400HI.

### CGA palette routines

Because the CGA palette is predetermined, you should not use the **setallpalette** routine on a CGA. Also, you should not use **setpalette**(*index*, *actual\_color*), except for *index* = 0. (This is an alternate way to set the CGA background color to *actual\_color*.)

### Color control on the EGA and VGA

On the EGA, the palette contains 16 entries from a total of 64 possible colors, and each entry is user-settable. You can retrieve the current palette with **getpalette**, which fills in a structure with the palette's size (16) and an array of the actual palette entries (the "hardware color numbers" stored in the palette). You can change the palette entries individually with **setpalette**, or all at once with **setallpalette**.

The default EGA palette corresponds to the 16 CGA colors, as given in the previous color table: black is in entry 0, blue in entry 1, ..., white in entry 15. There are constants defined in `graphics.h` that contain the corresponding hardware color values: these are `EGA_BLACK`, `EGA_WHITE`, and so on. You can also get these values with **getpalette**.

The **setbkcolor**(*color*) routine behaves differently on an EGA than on a CGA. On an EGA, **setbkcolor** copies the actual color value that's stored in entry *#color* into entry #0.

As far as colors are concerned, the VGA driver behaves like the EGA driver; it just has higher resolution (and smaller pixels).

Error handling in graphics mode

Here's a quick summary of the graphics-mode error-handling functions:

- grapherrormsg** Returns an error message string for the specified error code.
- graphresult** Returns an error code for the last graphics operation that encountered a problem.

If an error occurs when a graphics library function is called (such as a font requested with **setttextstyle** not being found), an internal error code is set. You retrieve the error code for the last graphics operation that reported an error by calling **graphresult**. The following error return codes are defined:

Error code	<i>graphics_errors</i> constant	Corresponding error message string
0	grOk	No error
-1	grNoInitGraph	(BGI) graphics not installed (use <b>initgraph</b> )
-2	grNotDetected	Graphics hardware not detected
-3	grFileNotFound	Device driver file not found
-4	grInvalidDriver	Invalid device driver file
-5	grNoLoadMem	Not enough memory to load driver
-6	grNoScanMem	Out of memory in scan fill
-7	grNoFloodMem	Out of memory in flood fill
-8	grFontNotFound	Font file not found
-9	grNoFontMem	Not enough memory to load font
-10	grInvalidMode	Invalid graphics mode for selected driver
-11	grError	Graphics error
-12	grIOerror	Graphics I/O error
-13	grInvalidFont	Invalid font file
-14	grInvalidFontNum	Invalid font number
-15	grInvalidDeviceNum	Invalid device number
-18	grInvalidVersion	Invalid version of file

A call to **grapherrormsg(graphresult())** returns the error strings listed in the previous table.

The error return code accumulates, changing only when a graphics function reports an error. The error return code is reset to 0 only when **initgraph** executes successfully, or when you call **graphresult**. Therefore, if you want to know which graphics function returned which error, you should store the value of **graphresult** into a temporary variable and then test it.

State query Here's a quick summary of the graphics mode state query functions:

<b>getarccoords</b>	Returns information about the coordinates of the last call to <b>arc</b> or <b>ellipse</b> .
<b>getaspectratio</b>	Returns the aspect ratio of the graphics screen.
<b>getbkcolor</b>	Returns the current background color.
<b>getcolor</b>	Returns the current drawing color.
<b>getdrivername</b>	Returns name of current graphics driver.
<b>getfillpattern</b>	Returns the user-defined fill pattern.
<b>getfillsettings</b>	Returns information about the current fill pattern and color.
<b>getgraphmode</b>	Returns the current graphics mode.
<b>getlinesettings</b>	Returns the current line style, line pattern, and line thickness.
<b>getmaxcolor</b>	Returns the current highest valid pixel value.
<b>getmaxmode</b>	Returns maximum mode number for current driver.
<b>getmaxx</b>	Returns the current <i>x</i> resolution.
<b>getmaxy</b>	Returns the current <i>y</i> resolution.
<b>getmodename</b>	Returns name of a given driver mode.
<b>getmoderange</b>	Returns the mode range for a given driver.
<b>getpalette</b>	Returns the current palette and its size.
<b>getpixel</b>	Returns the color of the pixel at <i>x,y</i> .
<b>gettextsettings</b>	Returns the current text font, direction, size, and justification.
<b>getviewsettings</b>	Returns information about the current viewport.
<b>getx</b>	Returns the <i>x</i> coordinate of the current position (CP).
<b>gety</b>	Returns the <i>y</i> coordinate of the current position (CP).

In each of Turbo C++'s graphics functions categories there is at least one state query function. These functions are mentioned under their respective categories and also covered here. Each of the Turbo C++ graphics state query functions is named **getsomething** (except in the error-handling category). Some of them take no argument and return a single value representing the requested information; others take a pointer to a structure defined

in `graphics.h`, fill that structure with the appropriate information, and return no value.

The state query functions for the graphics system control category are **getgraphmode**, **getmaxmode**, and **getmoderange**: The first returns an integer representing the current graphics driver and mode, the second returns the maximum mode number for a given driver, and the third returns the range of modes supported by a given graphics driver. **getmaxx** and **getmaxy** return the maximum *x* and *y* screen coordinates for the current graphics mode.

The drawing and filling state query functions are **getarcoords**, **getaspectratio**, **getfillpattern**, **getfillsettings**, and **getlinesettings**. **getarcoords** fills a structure with coordinates from the last call to **arc** or **ellipse**; **getaspectratio** tells the current mode's aspect ratio, which the graphics system uses to make circles come out round. **getfillpattern** returns the current user-defined fill pattern. **getfillsettings** fills a structure with the current fill pattern and fill color. **getlinesettings** fills a structure with the current line style (solid, dashed, and so on), line width (normal or thick), and line pattern.

In the screen- and viewport-manipulation category, the state query functions are **getviewsettings**, **getx**, **gety**, and **getpixel**. When you have defined a viewport, you can find out its absolute screen coordinates and whether clipping is active by calling **getviewsettings**, which fills a structure with the information. **getx** and **gety** return the (viewport-relative) *x*- and *y*-coordinates of the CP. **getpixel** returns the color of a specified pixel.

The graphics mode text-output function category contains one all-inclusive state query function: **gettextsettings**. This function fills a structure with information about the current character font, the direction in which text will be displayed (horizontal or bottom-to-top vertical), the character magnification factor, and the text-string justification (both horizontal and vertical).

Turbo C++'s color-control function category includes three state query functions. **getbkcolor** returns the current background color, and **getcolor** returns the current drawing color. **getpalette** fills a structure with the size of the current drawing palette and the palette's contents. **getmaxcolor** returns the highest valid pixel value for the current graphics driver and mode (*palette size* - 1).

Finally, **getmodename** and **getdrivername** return the name of a given driver mode and the name of the current graphics driver, respectively.



## *Interfacing with assembly language*

This chapter tells you how to write your assembler code so that it works well with Turbo C++. We assume that you know how to write 8086 assembly language routines and how to define segments, data constants, and so on. If you are unfamiliar with these concepts, read the Turbo Assembler manuals for more information, especially "Interfacing Turbo Assembler with Turbo C" in the *Turbo Assembler User's Guide*. Turbo Assembler 2.0 includes several features which make interfacing with Turbo C++ easy and transparent.

### Mixed-language programming

---

Turbo C++ eases the way for your C programs to call routines written in assembler and, in return, for programs written in assembler to call your C routines. In this section, we make it clear how easy interfacing Turbo C++ to assembly language can be; we also provide support information for such interfacing.

#### Parameter- passing sequences

---

Turbo C++ supports two methods of passing parameters to a function. One is the standard C method, which we will explain first; the other is the Pascal method.

## C parameter-passing sequence

Suppose you have declared the following function prototype:

```
void funca(int p1, int p2, long p3);
```

By default, Turbo C++ uses the C parameter-passing sequence, also called the C calling convention. When this function (**funca**) is called, the parameters are pushed on the stack in right-to-left order (*p3, p2, p1*), following which the return address is pushed on the stack. So, if you make the call

```
main()
{
    int i, j;
    long k;
    . . .
    i = 5; j = 7; k = 0x1407AA;
    funca(i, j, k);
    . . .
}
```

the stack will look like this (just before the return address is pushed):

*On the 8086, the stack grows from high memory to low memory, so i is currently at the top of the stack.*

```
sp + 06: 0014
sp + 04: 07AA k = p3
sp + 02: 0007 j = p2
sp:      0005 i = p1
```

The routine being called doesn't need to know exactly how many parameters have been pushed onto the stack. All it assumes is that the parameters it expects are there.

Also—and this is very important—the routine being called should not pop parameters off the stack. Why? Because the calling routine will. For example, the assembly language that the compiler produces from the C source code for this main function looks something like this:

```
mov WORD PTR [bp-8], 5           ;Set i = 5
mov WORD PTR [bp-6], 7           ;Set j = 7
mov WORD PTR [bp-2], 0014h       ;Set k = 0x1407AA
mov WORD PTR [bp-4], 07AAh
push WORD PTR [bp-2]             ;Push high word of k
push WORD PTR [bp-4]             ;Push low word of k
push WORD PTR [bp-6]             ;Push j
push WORD PTR [bp-8]             ;Push i
call NEAR PTR funca              ;Call funca (push addr)
add sp, 8                         ;Adjust stack
```

Note the last instruction carefully: `add sp, 8`. The compiler knows at that point exactly how many parameters have been pushed onto the stack; it also knows that the return address was pushed by the call to **funca** and was already popped off by the **ret** instruction at the end of **funca**.

### Pascal parameter-passing sequence

The other approach is the standard Pascal method for passing parameters (also known as the Pascal calling convention). This does *not* mean you can call Turbo Pascal functions from Turbo C++; You can't. If **funca** is declared as

```
void pascal funca(int p1, int p2, long p3);
```

then, when this function is called, the parameters are pushed on the stack in left-to-right order (*p1, p2, p3*), following which the return address is pushed on the stack. So, if you make the call

```
main()
{
    int i, j;
    long k;
    . . .
    i = 5; j = 7; k = 0x1407AA;
    funca(i, j, k);
    . . .
}
```

the stack will look like this (just before the return address is pushed):

```
sp + 06: 0005 i = p1
sp + 04: 0007 j = p2
sp + 02: 0014
sp:      07AA k = p3
```

So, what's the big difference? Well, besides switching the order in which the parameters are pushed, the Pascal parameter-passing sequence assumes that the routine being called (**funca**) knows how many parameters are being passed to it and adjusts the stack accordingly. In other words, the assembly language for the call to **funca** now looks like this:

```
push WORD PTR [bp-8]    ;Push i
push WORD PTR [bp-6]    ;Push j
push WORD PTR [bp-2]    ;Push high word of k
push WORD PTR [bp-4]    ;Push low word of k
call NEAR PTR funca     ;Call funca (push addr)
```

Note that there is no `add sp, 8` instruction after the call. Instead, **funca** uses the instruction `ret 8` at termination to clean up the stack before returning to **main**.

By default, all functions you write in Turbo C++ use the C method of parameter passing. The only exception is when you use the `-p` compiler option (Pascal button on the Code Generation dialog); in which case, all functions use the Pascal method. In that situation, you can force a given function to use the C method of parameter passing by using the modifier **cdecl**, as in

```
void cdecl funca(int p1, int p2, int p3);
```

That overrides the `-p` compiler directive.

Now, why would you want to use the Pascal calling convention at all? There are two major reasons.

- You may be calling existing assembly language routines that use that calling convention.
- The calling code produced is slightly smaller, since it doesn't have to clean up the stack afterwards.

*Some problems might arise from using the Pascal calling convention.*

First, it's not as robust as the C calling convention. You cannot pass a variable number of parameters (as you can with the C convention), since the routine being called has to know how many parameters are being passed and clean up the stack accordingly. Passing either too few or too many parameters will almost certainly lead to serious problems, whereas doing so to a C-convention routine usually has no ill effects (beyond, possibly, wrong answers).

Second, if you use the `-p` compiler option, then you must be sure to include any header files for standard C functions that you call. Why? Because if you don't, Turbo C++ will use the Pascal calling (and naming) convention for each of those functions—and your program will not link.

The header files declare each of those functions as **cdecl**, so if you include them, the compiler will see that and use the C calling convention instead.

The upshot is this: If you're going to use the Pascal calling convention in a Turbo C++ program, be sure to use function prototypes as much as possible, with each function explicitly declared as **cdecl** or **pascal**. It's useful in this case to enable the "Function call with no prototype" warning to ensure that every function called has a prototype.

# Setting up to call .ASM from Turbo C++

---

When writing your assembly language routines, there are certain conventions that you must follow to (1) ensure that the linker can get the necessary information, and (2) ensure that the file format jibes with the memory model used for your C program.

## Simplified segment directives

---

Ordinarily, your assembly language modules will consist of three sections: code, initialized data, and uninitialized data. Each type of information is organized into its own segment using certain names which are dependent on the memory model used for your C program.

Turbo Assembler (TASM) provides three simplified segment directives (.CODE, .DATA, and .DATA?) for you to use when defining these segments. They tell the assembler to use the default segment names for the memory model which you specify using the .MODEL directive. For example, if your C program uses the small memory model, you could organize each assembly module with simplified segment directives as shown in the following table.

---

.MODEL SMALL
.CODE
...code segment...
.DATA
...initialized data segment...
.DATA?
...uninitialized data segment...

---

## Standard segment directives

---

In some cases, you may want to use segment names other than the defaults for your memory model. To do so, you must use the standard segment directives as illustrated in Table 6.1.

Table 6.1  
Assembly language file  
format

<i>code</i>	SEGMENT ASSUME .....code segment.....	BYTE PUBLIC 'CODE' CS: code, DS: dseg
<i>code</i>	ENDS	
<i>dseg</i> <i>data</i>	GROUP SEGMENT .....initialized data segment.....	_DATA, _BSS WORD PUBLIC 'DATA'
<i>data</i>	ENDS	
_BSS	SEGMENT .....uninitialized data segment.....	WORD PUBLIC 'BSS'
_BSS	ENDS	
	END	

The identifiers *code*, *data*, and *dseg* in this layout have specific replacements, depending on the memory model being used; Table 6.2 shows what you should use for each model. *filename* in Table 6.2 is the name of the module; use it consistently in the NAME directive and in the identifier replacements.

Note that with the huge memory model, there is no \_BSS segment, and the GROUP definition is dropped completely. In general, \_BSS is optional; you only define it if you will be using it.

The best way to create an assembly language template is to compile an empty program to .ASM (using the TCC option **-S**) and look at the generated assembly code.

Table 6.2  
Identifier replacements and  
memory models

Model	Identifier replacements	Code and data pointers
Tiny, Small	<i>code</i> = _TEXT <i>data</i> = _DATA <i>dseg</i> = DGROUP	Code: DW _TEXT:xxx Data: DW DGROUP:xxx
Compact	<i>code</i> = _TEXT <i>data</i> = _DATA <i>dseg</i> = DGROUP	Code: DW _TEXT:xxx Data: DD DGROUP:xxx
Medium	<i>code</i> = <i>filename</i> _TEXT <i>data</i> = _DATA <i>dseg</i> = DGROUP	Code: DDxxx Data: DW DGROUP:xxx
Large	<i>code</i> = <i>filename</i> _TEXT <i>data</i> = _DATA <i>dseg</i> = DGROUP	Code: DDxxx Data: DD DGROUP:xxx
Huge	<i>code</i> = <i>filename</i> _TEXT <i>data</i> = <i>filename</i> _DATA	Code: DDxxx Data: DDxxx

## Defining data constants and variables

---

Memory models also affect how you define any data constants that are pointers to code, data, or both. Table 6.2 shows what those pointers should look like, where *xxx* is the address being pointed to.

Some definitions use DW (Define Word), while others use DD (Define Doubleword), indicating the size of the resulting pointer. Numeric and text constants are defined normally.

Variables are, of course, defined just the same as constants. If you want variables that are not initialized to specific values, you can declare them in the `_BSS` segment, entering a question mark (?) where you would normally put a value.

## Defining global and external identifiers

---

Once you have created a module, your Turbo C++ program needs to know which functions it can call and which variables it can reference. Likewise, you may want to be able to call your Turbo C++ functions from within your assembly language routines, or you may want to be able to reference variables declared within your Turbo C++ program.

When making these calls, you need to understand something about the Turbo C++ compiler and linker. When you declare an external identifier, the compiler automatically adds an underscore (`_`) to the front before saving that identifier in the object module. This means that you should put an underscore on the front of any identifiers in your assembly language module that you want to reference from your C program. Pascal identifiers are treated differently than C identifiers—they are uppercased and are *not* prefixed with an underscore character.

Underscores for C identifiers are optional, but on by default. They can be turned off with the `-u-` command-line option. However, if you are using the standard Turbo C++ libraries, you will encounter problems unless you rebuild the libraries. (To do this, you will need another Turbo C++ product—the source code to the run-time libraries; contact Borland for more information.)

If any **asm** code in your source file references any C identifiers (data or functions), those identifiers must begin with underscore

characters (unless you are using one of the language specifiers described above).

Turbo Assembler (TASM) is not case-sensitive; in other words, when you assemble a program, all identifiers are saved as uppercase only. TASM's `/mx` option makes it case sensitive for public and externals. The Turbo C++ linker also saves **extern** identifiers as uppercase, so things should match up fine. In our examples, we put keywords and directives in uppercase, and all other identifiers and opcodes in lowercase; this matches the style found in the TASM reference manual. You are free to use all uppercase (or all lowercase), or any mixture thereof, as you please.

To make identifiers visible outside of your assembly language module, you need to declare them as being PUBLIC.

So, for example, if you were to write a module that had the integer functions *max* and *min*, and the integer variables MAXINT, *lastmax* and *lastmin*, you would put the statement

```
PUBLIC _max, _min
```

in your code segment, and the statements

```
PUBLIC _MAXINT, _lastmax, _lastmin
_MAXINT DW 32767
_lastmin DW 0
_lastmax DW 0
```

in your data segment.

**TASM 2.0** Turbo Assembler 2.0 extends the syntax of many directives to allow an optional language specifier. For instance, if you specify `C` in your module's `.MODEL` directive, all identifier names will be saved in the object module with leading underscores. This feature can also be specified on a directive-by-directive basis. Using Turbo Assembler 2.0's `C` language specifier, the above declarations could also be written as

```
PUBLIC C max, min
PUBLIC C MAXINT, lastmax, lastmin
MAXINT DW 32767
lastmin DW 0
lastmax DW 0
```

# Setting up to call Turbo C++ from .ASM

---

Use the `EXTRN` statement to let your assembly language module reference functions and variables that are declared in your Turbo C++ program.

## Referencing functions

---

To be able to call a C function from an assembly language routine, you must declare it in your assembly language module with the statement

```
EXTRN fname : fdist
```

where *fname* is the name of the function, and *fdist* is either `near` or `far`, depending on whether the C function is near or far. So you could have the following in your code segment:

```
EXTRN _myCfunc1:near, _myCfunc2:far
```

allowing you to call `myCfunc1` and `myCfunc2` from within your assembly language routines.

**TASM 2.0** Using Turbo Assembler 2.0's C language specifier, the statement above could also be written as

```
EXTRN C myCfunc1:near, myCfunc2:far
```

## Referencing data

---

To reference variables, place the appropriate `EXTRN` statement(s) inside of your data segment, using the format

```
EXTRN vname : size
```

where *vname* is the name of the variable, and *size* indicates the size of the variable.

The possible values for *size* are as follows:

BYTE (1 byte)	QWORD (8 bytes)
WORD (2 bytes)	TBYTE (10 bytes)
DWORD (4 bytes)	

So, if your C program had the following global variables:

```
int i,jarray[10];
char ch;
long result;
```

you could make them visible within your module with the following statement:

```
EXTRN _i:WORD, _jarray:WORD, _ch:BYTE, _result:DWORD
```

**TASM 2.0** or using Turbo Assembler 2.0's C language specifier:

```
EXTRN C i:WORD, jarray:WORD, ch:BYTE, result:DWORD
```

**Important!** If you're using the huge memory model, the EXTRN statements must appear outside of any segments. This applies to both functions and variables.

## Defining assembly language routines

---

Now that you know how to set everything up, take a look at how to actually write a function in assembly language. There are some important things to consider: passing parameters, returning values, and using the correct register conventions.

Suppose you want to write the function **min**, which you can assume has the following function prototype in C:

```
extern int min(int v1, int v2);
```

You want **min** to return the minimum of the two values passed to it. The overall format of **min** is going to be

```
        PUBLIC  _min
_min PROC  NEAR
        ...
_min ENDP
```

This assumes, of course, that **min** is going to be a near function; if it were a far function, you would substitute FAR for NEAR. Note that we've added the underscore to the start of **min**, so that the Turbo C++ linker can correctly resolve the references. If we had used Turbo Assembler 2.0's C language specifier in the PUBLIC statement, the assembler would have taken care of this task for us.

---

### Passing parameters

Your first decision is which parameter-passing convention to use; barring an adequate reason to use it, avoid the Pascal convention; use the C method instead. This means that when **min** gets called, the stack is going to look like this:

```
sp + 04: v2
```

```

    sp + 02: v1
    sp:      return addr

```

You want to get to the parameters without popping anything off the stack, so you'll save the base pointer (BP), move the stack pointer (SP) into the base pointer, then use that to index directly into the stack to get your values. Note that when you push BP onto the stack, the relative offsets of the parameters will increase by two, since there will now be two more bytes on the stack.

**TASM 2.0** Turbo Assembler 2.0 provides an easy way to reference your function's parameters and deal with the stack. Keep reading though; it's important to understand how stack addressing works.

## Handling return values

Your function returns an integer value; where do you put that? For 16-bit (2-byte) values (**char**, **short**, **int**, **enum**, and **near** pointers), you use the AX register; for 32-bit (4-byte) values (including **far** and **huge** pointers), you use the DX register as well, with the high-order word (segment address for pointers) in DX and the low-order word in AX.

**float**, **double**, and **long double** values are returned in the 80x87 top-of-stack (TOS) register, ST(0); if the 80x87 emulator is being used, then the value is returned in the emulator TOS register. The calling function must then copy that value to wherever it's needed.

Structures that are 1 byte long are returned in AL. Structures that are 2 bytes long are returned in AX. Structures 4 bytes long are returned in AX:DX. Structure values of size 3 or greater than 5 bytes are returned by placing the value in a static data location, then returning a pointer to that location (AX in the small data models, DX:AX in the large data models). The *called* routine has to copy the return value to the location this pointer points to.

For the **min** example, all you're dealing with is a 16-bit value, so you can just place the answer in AX.

Here's what your code looks like now:

```

        PUBLIC  _min
_min PROC    NEAR
        push   bp                ;Save bp on stack
        mov   bp,sp             ;Copy sp into bp
        mov   ax,[bp+4]         ;Move v1 into ax
        cmp   ax,[bp+6]         ;Compare with v2

```

```

        jle     exit                ;If v1 > v2
        mov     ax,[bp+6]          ;Then load ax with v2
exit:   pop     bp                ;Restore bp
        ret                                     ;And return to C
_min   ENDP

```

What if you declare **min** as a **far** function—how will that change things? The major difference is that the stack on entry will now look like this:

```

sp + 06: v2
sp + 04: v1
sp + 02: return segment
sp:      return offset

```

This means that the offsets into the stack have increased by two, since 2 extra bytes (for the return segment) had to be pushed onto the stack. Your far version of **min** would look like this:

```

        PUBLIC _min
_min   PROC FAR
        push   bp                ;Save bp on stack
        mov    bp,sp            ;Copy sp into bp
        mov    ax,[bp+6]        ;Move v1 into ax
        cmp    ax,[bp+8]        ;Compare with v2
        jle    exit            ;If v1 > v2
        mov    ax,[bp+6]        ;Then load ax with v2
exit:   pop    bp                ;Restore bp
        ret                                     ;And return to C
_min   ENDP

```

Note that all the offsets for *v1* and *v2* increased by two, to reflect the additional bytes on the stack.

Now, what if you decide to use the Pascal parameter-passing sequence?

Your stack on entry will now look like this (assuming **min** is back to being a NEAR function):

```

SP + 04: v1
SP + 02: v2
SP:      return addr

```

In addition, you will need to follow Pascal conventions for the identifier **min**: uppercase and no underscore.

Besides having swapped the locations of *v1* and *v2*, this convention also requires **min** to clean up the stack when it leaves, by specifying in the **RET** instruction how many bytes to pop off the

stack. In this case, you have to pop off four additional bytes for *v1* and *v2* (the return address is popped off automatically by **RET**).

Here's what the modified routine looks like:

```

                PUBLIC  MIN
MIN  PROC      NEAR                ;Pascal version
        push   bp                ;Save bp on stack
        mov    bp,sp             ;Copy sp into bp
        mov    ax,[bp+6]         ;Move v1 into ax
        cmp    ax,[bp+4]         ;Compare with v2
        jle    exit              ;If v1 > v2
        mov    ax,[bp+4]         ;Then load ax with v2
exit:  pop     bp                ;Restore bp
        ret    4                 ;Clear stack and return
MIN  ENDP

```

Here's one last example to show you why you might want to use the C parameter-passing sequence. Suppose you redefined **min** as follows:

```
int min(int count,...);
```

**min** can now accept any number of integers and will return the minimum value of them all. However, since **min** has no way of automatically knowing how many values are being passed, make the first parameter a *count value*, indicating how many values follow it.

For example, you might use it as follows:

```
i = min(5, j, limit, indx, lcount, 0);
```

assuming *i*, *j*, *limit*, *indx*, and *lcount* are all of type **int** (or a compatible type). The stack upon entry will look like this:

```

sp + 08:  (etc.)
sp + 06:  v2
sp + 04:  v1
sp + 02:  count
sp:      return addr

```

The modified version of **min** now looks like this:

```

                PUBLIC  _min
_min  PROC      NEAR
        push   bp                ;Save bp on stack
        mov    bp,sp             ;Copy sp into bp
        mov    cx,[bp+4]         ;Move count into cx
        cmp    cx,0              ;Compare with 0
        jle    exit              ;If <= 0, then exit

```

```

        lea    bx,[bp+6]           ;Make bx point to first value
        mov    ax,[bx]           ;Move first value into ax
        jmp    ltest             ;And test loop
compare: cmp    ax,[bx]         ;Compare with next value
        jle    ltest             ;If next value is lower
        mov    ax,[bx]         ;Then load ax with next value
ltest:   add    bx,2             ;Move to new value
        loop   compare          ;Then loop back
exit:    pop    bp              ;Restore bp
        ret     ;And return to C
_min    ENDP

```

This version correctly handles all possible values of *count*.

- If *count*  $\leq$  0, **min** returns 0.
- If *count* = 1, **min** returns the first value in the list.
- If *count*  $\geq$  2, **min** makes successive comparisons to find the lowest value in the parameter list.

Now that you understand how to manipulate the stack when writing your own functions, you might appreciate some of the new extensions to Turbo Assembler version 2.0. Some of these extensions let you automatically create variable names, set up and clean up the stack inside your PROC, and access parameters easily, all using the conventions of the language that you specify.

Our first version of **min** (page 257) can be written using the new extensions as

```

        PUBLIC C MIN
min     PROC   C NEAR v1: WORD, v2: WORD
        mov    ax,v1
        cmp    ax,v2
        jle    exit
        mov    ax,v2
exit:   ret
min     ENDP

```

The Pascal-style version (page 259) can be written as

```

        PUBLIC PASCAL MIN
min     PROC   PASCAL NEAR v1: WORD, v2: WORD
        mov    ax,v1
        cmp    ax,v2
        jle    exit
        mov    ax,v2
exit:   ret
min     ENDP

```

Notice that the code you write is identical, except for the substitution of the keyword **PASCAL** for C. The code that is actually generated by the assembler, however, corresponds to our original examples. See the Turbo Assembler manuals for a complete description of these new language-independent features.

Like normal C procedures and functions, **external** assembly language routines must observe certain programming rules to work correctly with the overlay manager.

If an assembly language routine makes calls to *any* overlaid procedures or functions, the assembly language routine must be far, and it must set up a stack frame using the BP register. See page 217 for more details.

## Register conventions

---

You used several registers (BP, SP, AX, BX, CX) in **min**; were you able to do so safely? What about any registers that your Turbo C++ program might be using?

As it turns out, you wrote this function correctly. Of those you used, the only register that you had to worry about was BP, and you saved that on the stack on entry, then restored it from the stack on exit.

The other two registers that you have to worry about are SI and DI; these are the registers Turbo C++ uses for any register variables. If you use them at all within an assembly language routine, then you should save them (probably on the stack) on entering the routine, and restore them on leaving. However, if you compile your Turbo C++ program with the **-r-** option (Register Variables should be unchecked on the Code Generation dialog), then you don't have to worry about saving SI and DI.

**Note** You must use caution if you use the **-r-** option. Refer to Chapter 4, "The command-line compiler," in the *User's Guide* for details about this register variables option.

The registers CS, DS, SS, and ES have specific values, depending on the memory model being used. Here are the relationships:

Tiny	CS = DS = SS ES = scratch
Small, Medium	CS != DS, DS = SS ES = scratch

Compact, Large	CS != DS != SS ES = scratch (one CS per module)
Huge	CS != DS != SS ES = scratch (one CS and one DS per module)

You can set DS to be not equal to SS for the tiny, small, and medium models by setting these command-line compiler options: **-mtl**, **-ms!**, and **-mm!**. See Chapter 4, "The command-line compiler," in the *User's Guide* for more information on these options.

**TASM 2.0** Turbo Assembler 2.0 also lets you specify this (DS!=SS) when you are using the simplified segment directives through the use of the model modifier in the `.MODEL` directive.

## Calling C functions from .ASM routines

---

Yes, you can go the other way: You can call your C routines from within your assembly language modules. First, though, you have to make the C function visible to your assembly language module. We've already discussed briefly how to do this: Declare it as `EXTRN`, with either a **near** or a **far** modifier. For example, say you've written the following C function:

```
long docalc(int *fact1, int fact2, int fact3);
```

For simplicity, assume **docalc** is a C function (as opposed to Pascal). Assuming you're using the tiny, small, or compact memory model, you'd declare it as this in your assembly module:

```
EXTRN _docalc:near
```

Likewise, if you were using the medium, large, or huge memory models, you'd declare it as `_docalc:far`.

**TASM 2.0** Using Turbo Assembler 2.0's C language specifier, these declarations could be written as

```
EXTRN C docalc:near
```

and

```
EXTRN C docalc:far
```

**docalc** is to be called with three parameters:

- the address of a location named *xval*
- the value stored in a location named *imax*
- a third constant value of 421 (base 10)

You should also assume that you want to save the result in a 32-bit location named *ans*. The equivalent call in C would then be

```
ans = docalc(&xval,imax,421);
```

You'll need to push 421 on the stack first, then *imax*, then the address of *xval*, and then call **docalc**. When it returns, you'll need to clean up the stack, which will have six extra bytes on it, and then move the answer into *ans* and *ans+2*.

Here's what your code will look like:

```

mov   ax,421                ;Get 421, push onto stack
push  ax
push  imax                  ;Get imax, push onto stack
lea   ax,xval               ;Get &xval, push onto stack
push  ax
call  _docalc               ;Call docalc
add   sp,6                  ;Clean up stack
mov   ans,ax                ;Move 32-bit result into ans
mov   ans+2,dx              ;Including high-order word

```

**TASM 2.0** Turbo Assembler version 2.0 includes several extensions to make the C-assembly interface process even easier. Some of these extensions let you automatically create C-style variable names, push parameters on the stack in the correct order for C, and clean up the stack after calling a C function. For example, the **docalc** routine can be written as

```

EXTRN  C docalc:near
mov    bx,421
lea    ax,xval
call   docalc C ax,imax,bx
mov    ans,ax
mov    ans+2,dx

```

See the Turbo Assembler manuals for a complete description of these new features.

What if **docalc** used the Pascal parameter-passing sequence instead? Then you would have to reverse the order of the parameters, and you wouldn't have to worry about cleaning up the stack upon return, since the routine would have done that for

you. Also, you would need to spell **docalc** in the assembly source using Pascal conventions (uppercase and no underscore).

The EXTRN statement is then

```
EXTRN DOCALC:near
```

and the code to call **docalc** is

```
lea    ax,xval                ;Get &xval, push onto stack
push  ax
push  imax                    ;Get imax, push onto stack
mov   ax,421                  ;Get 421, push onto stack
push  ax
call  DOCALC                  ;Call docalc
mov   ans,ax                  ;Move 32-bit result into ans
mov   ans+2,dx                ;Including high-order word
```

Turbo Assembler version 2.0 also includes extensions to simplify the Pascal-style assembly interface, letting you automatically create Pascal-style variable names and push parameters on the stack in the correct order for Pascal. For example, the Pascal-style **docalc** routine can be written as

```
EXTRN PASCAL docalc:near

lea    ax,xval
mov    bx,421
call  docalc PASCAL ax,imax,bx
mov    ans,ax
mov    ans+2,dx
```

That's all you need to know to get started interfacing assembly language with Turbo C++.

## Pseudovariables, inline assembly, and interrupt functions

---

What if you want to do some low-level work, but don't want to go to all the trouble of setting up a separate assembly language module? Turbo C++ still has the answer for you—three answers, in fact: pseudovariables, inline assembly, and interrupt functions. The rest of this chapter shows how each of these can help you get the job done.

## Pseudovariables

---

The CPU in your system (the 8088 or 80x86 processor) has a number of registers, or special storage areas, that it uses to manipulate values. Each register is 16 bits (2 bytes) long; most of them have some special purpose, though several can be used for general purposes as well. See “Memory models” on page 187 of Chapter 4 for specific details on these CPU registers.

Sometimes in low-level programming, you might want to directly access these registers from your C program.

- You might want to load values into them before calling a system routine.
- You might want to see what values they currently hold.

For example, you can call certain routines in your computer’s ROM by executing the **INT** (interrupt) instruction, but first you have to put the necessary information into certain registers, like this:

```
void readchar(unsigned char page, unsigned char *ch,
             unsigned char *attr);

{
    _AH = 8;           /* Service code: read char, attribute */
    _BH = page;       /* Specify which display page */
    geninterrupt(0x10) /* Call INT 10h services */
    *ch = _AL;        /* Get ASCII code of character read */
    *attr = _AH;      /* Get attribute of character read */
}
```

As you can see, the service code and the display page number are both being passed to the **INT 10h** routine; the values returned are copied over into *ch* and *attr*.

Turbo C++ makes it very easy for you to access these registers through *pseudovariables*. A pseudovvariable is simply an identifier that corresponds to a given register: You can use it as if it were a variable of type **unsigned int** or **unsigned char**.

Here are some guidelines for using pseudovariables safely:

- Assigning simple variables to pseudovariables and vice versa doesn’t trash any other registers if no type conversion is involved.

- Assigning constants to pseudovariables will also not destroy the data in other registers, except for assignments to segment registers (`_CS`, `_DS`, `_SS`, `_ES`), which use the AX register.
- Simple indirections via a pointer variable will generally destroy the data in one of these: `_BX`, `_SI` or `_DI`, and possibly `_ES`.
- If you have to set up a number of registers (for example, for doing a ROM call), it is safest to set up `_AX` last, because it is most likely to be inadvertently modified by other statements.

The next table shows a complete list of the pseudovariables you can use, their types, the registers they correspond to, and what those registers are usually used for.

Table 6.3  
Pseudovariables

Pseudo-variable	Type	Register	Purpose
<code>_AX</code>	unsigned int	AX	General/accumulator
<code>_AL</code>	unsigned char	AL	Lower byte of AX
<code>_AH</code>	unsigned char	AH	Upper byte of AX
<code>_BX</code>	unsigned int	BX	General/indexing
<code>_BL</code>	unsigned char	BL	Lower byte of BX
<code>_BH</code>	unsigned char	BH	Upper byte of BX
<code>_CX</code>	unsigned int	CX	General/counting and loops
<code>_CL</code>	unsigned char	CL	Lower byte of CX
<code>_CH</code>	unsigned char	CH	Upper byte of CX
<code>_DX</code>	unsigned int	DX	General/holding data
<code>_DL</code>	unsigned char	DL	Lower byte of DX
<code>_DH</code>	unsigned char	DH	Upper byte of DX
<code>_CS</code>	unsigned int	CS	Code segment address
<code>_DS</code>	unsigned int	DS	Data segment address
<code>_SS</code>	unsigned int	SS	Stack segment address
<code>_ES</code>	unsigned int	ES	Extra segment address
<code>_SP</code>	unsigned int	SP	Stack pointer (offset to SS)
<code>_BP</code>	unsigned int	BP	Base pointer (offset to SS)
<code>_DI</code>	unsigned int	DI	Used for register variables
<code>_SI</code>	unsigned int	SI	Used for register variables
<code>_FLAGS</code>	unsigned int	flag	Processor status

The pseudovariables can be treated just as if they were regular global variables of the appropriate type (**unsigned int**, **unsigned char**). However, since they refer to the CPU's registers, rather than some arbitrary location in memory, there are some restrictions and concerns you must be aware of.

- You cannot use the address-of operator (&) with a pseudovari-  
able, since a pseudovari-able has no address.
- Since the compiler is constantly generating code that uses the  
registers (after all, that's what most of the 8086's instructions  
do), you have absolutely no guarantee that values you place in  
pseudovari-ables will be preserved for any length of time.  
This means you must assign values right before using them and  
read values right after obtaining them, as in **readchar** (previous  
example). This is especially true of the general-purpose regis-  
ters (AX, AH, AL, and so on), since the compiler freely uses  
these for temporary storage. On top of that, the CPU changes  
them in ways you might not expect; for example, using CX  
when it sets up a loop or does a shift operation, or using DX to  
hold the upper word of a 16-bit multiply.
- You can't rely on values of pseudovari-ables remaining the same  
across a function call. As an example of this, take the following  
code fragment:

```

_CX = 18;
myFunc();
i = _CX;

```

Not all registers are saved during a function call, so you have  
no guarantee that *i* will get assigned a value of 18. The only  
registers that you can count on having the same values before  
and after a function call are `_DS`, `_BP`, `_SI`, and `_DI`.

- You need to be very careful modifying certain registers, since  
this could have unexpected and untoward effects. For example,  
directly storing values to `_CS`, `_DS`, `_SS`, `_SP`, or `_BP` can (and  
almost certainly will) cause your program to behave erratically,  
since the machine code produced by the Turbo C++ compiler  
uses those registers in various ways.

---

## Inline assembly language

You've seen how to write separate assembly language routines  
and link them into your Turbo C++ program. Turbo C++ also lets  
you write assembly language code right inside your C program.  
This is known as *inline assembly*.

By default, **-B** invokes TASM. You can override it with **-Exxx**, where *xxx* is another assembler. See Chapter 4, "The command-line compiler," in the User's Guide for details.

To use inline assembly in your C program, you can use the **-B** compiler option. If you don't, and the compiler encounters inline assembly, it issues a warning and restarts itself with the **-B** option. You can avoid this with the `#pragma inline` statement in your source, which in effect enables the **-B** option for you when the compiler encounters it.

You must have a copy of Turbo Assembler (TASM). The compiler first generates an assembly file, then invokes TASM on that file to produce the .OBJ file.

Of course, you also need to be familiar with the 8086 instruction set and architecture. While you're not writing complete assembly language routines, you still need to know how the instructions you're using work, how to use them, and how not to use them.

Having done all that, you need only use the keyword **asm** to introduce an inline assembly language instruction. The format is

```
asm opcode operands ; or newline
```

where

- *opcode* is a valid 8086 instruction (Table 6.4 lists all allowable *opcodes*).
- *operands* contains the operand(s) acceptable to the *opcode*, and can reference C constants, variables, and labels.
- ; or *newline* is a semicolon or a new line, either of which signals the end of the **asm** statement.

A new **asm** statement can be placed on the same line, following a semicolon, but no **asm** statement can continue to the next line.

If you want to include a number of **asm** statements, surround them with braces:

```
asm {  
    pop ax; pop ds  
    iret  
}
```

Semicolons are not used to start comments (as they are in TASM). When commenting **asm** statements, use C-style comments, like this:

```
asm mov ax,ds; /* This comment is OK */  
asm {pop ax; pop ds; iret;} /* This is legal too */  
asm push ds ;THIS COMMENT IS INVALID!!
```

The assembly language portion of the statement is copied straight to the output, embedded in the assembly language that Turbo C++ is generating from your C instructions. Any C symbols are replaced with appropriate assembly language equivalents.

The inline assembly facility is not a complete assembler, so many errors will not be immediately detected. TASM will catch whatever errors there might be. However, TASM might not identify the location of errors, particularly since the original C source line number is lost.

Each **asm** statement counts as a C statement. For example,

```
myfunc()
{
    int i;
    int x;

    if (i > 0)
        asm mov x,4
    else
        i = 7;
}
```

This construct is a valid C **if** statement. Note that no semicolon was needed after the `mov x,4` instruction. **asm** statements are the only statements in C that depend on the occurrence of a new line. This is not in keeping with the rest of the C language, but this is the convention adopted by several UNIX-based compilers.

An assembly statement can be used as an executable statement inside a function, or as an external declaration outside of a function. Assembly statements located outside any function are placed in the DATA segment, and assembly statements located inside functions are placed in the CODE segment.

Here is an inline assembly version of the function **min** (introduced in "Handling return values" on page 257).

```

int min (int V1, int V2)
{
    asm {
        mov ax,V1
        cmp ax,V2
        jle minexit
        mov ax,V2
    }
    minexit:
    return (_AX);
}

```

Notice how similar this code is to the code on page 260, which takes advantage of Turbo Assembler 2.0's language-specific extensions.

You can include any of the 8086 instruction opcodes as inline assembly statements. There are four classes of instructions allowed by the Turbo C++ compiler:

- normal instructions—the regular 8086 opcode set
- string instructions—special string-handling codes
- jump instructions—various jump opcodes
- assembly directives—data allocation and definition

Note that all operands are allowed by the compiler, even if they are erroneous or disallowed by the assembler. The exact format of the operands is not enforced by the compiler.

**Opcodes** The following is a summary list of the opcode mnemonics that can be used in inline assembler:

Table 6.4  
Opcode mnemonics

*If you are using inline assembly in routines that use floating-point emulation (the TCC option -f), the opcodes marked with \*\* are not supported.*

aaa	fdivrp	fpatan	lsl
aad	feni	fprem	mov
aam	ffree**	fptan	mul
aas	fiadd	frndint	neg
adc	ficom	frstor	nop
add	ficomp	fsave	not
and	fidiv	fscale	or
bound	fidivr	fsqrt	out
call	file	fst	pop
cbw	fimul	fstcw	popa
clc	fincstp**	fstenv	popf
cld	finit	fstp	push
cli	fist	fstsw	pusha
cmc	fistp	fsub	pushf
cmp	fisub	fsubp	rcl
cwd	fisubr	fsubr	rcr
daa	fld	fsubrp	ret
das	fld1	ftst	rol
dec	fldcw	fwait	ror
div	fldenv	fxam	sahf
enter	fldl2e	fxch	sal
f2xm1	fldl2t	fxtract	sar
fabs	fldlg2	fyl2x	sbb
fadd	fldln2	fyl2xp1	shl
faddp	fldpi	hlt	shr
fbld	fldz	idiv	smsw
fbstp	fmul	imul	stc
fchs	fmulp	in	std
fclex	fclex	inc	sti
fcom	fndisi	int	sub
fcomp	fneni	into	test
fcompp	fninit	iret	verr
fdecstp**	fnop	lahf	verw
fdisi	fnsave	lds	wait
fdiv	fnstcw	lea	xchg
fdivp	fnstenv	leave	xlat
fdivr	fnstsw	les	xor

When using 80186 instruction mnemonics in your inline assembly statements, you must include the `-1` command-line option. This forces appropriate statements into the assembly language compiler output so that Turbo Assembler will expect the mnemonics. If you are using an older assembler, these mnemonics may not be supported.

### String instructions

In addition to the listed opcodes, string instructions given in the following table may be used alone or with repeat prefixes.

Table 6.5  
String instructions

cmps	insw	movsb	outsw	stos
cmpsb	lods	movsw	scas	stosb
cmpsw	lodsb	outs	scasb	stosw
ins	lodsw	outsb	scasw	
insb	movs			

### Prefixes

The following prefixes may be used:

lock rep repe repne repnz repz

### Jump instructions

Jump instructions are treated specially. Since a label cannot be included on the instruction itself, jumps must go to C labels (discussed in "Using jump instructions and labels" on page 274). The allowed jump instructions are given in the next table.

Table 6.6  
Jump instructions

ja	jge	jnc	jnp	js
jae	jle	jne	jns	jz
jb	jle	jng	jnz	loop
jbe	jmp	jnge	jo	loope
jc	jna	jnl	jp	loopne
jcxz	jnae	jnl	jpe	loopnz
je	jnb	jno	jpo	loopz
jg	jnbe			

### Assembly directives

The following assembly directives are allowed in Turbo C++ inline assembly statements:

db dd dw extrn

Inline assembly references to data and functions

You can use C symbols in your **asm** statements; Turbo C++ automatically converts them to appropriate assembly language operands and tacks underscores onto identifier names. You can use any symbol, including automatic (local) variables, register variables, and function parameters.

In general, you can use a C symbol in any position where an address operand would be legal. Of course, you can use a register variable wherever a register would be a legal operand.

If the assembler encounters an identifier while parsing the operands of an inline assembly instruction, it searches for the identifier in the C symbol table. The names of the 8086 registers are excluded from this search. Either uppercase or lowercase forms of the register names may be used.

### Inline assembly and register variables

Inline assembly code can freely use SI or DI as scratch registers. If you use SI or DI in inline assembly code, the compiler won't use these registers for register variables.

### Inline assembly, offsets, and size overrides

When programming, you don't need to be concerned with the exact offsets of local variables. Simply using the name will include the correct offsets.

However, it may be necessary to include appropriate WORD PTR, BYTE PTR, or other size overrides on assembly instruction. A DWORD PTR override is needed on LES or indirect far call instructions.

### Using C structure members

You can reference structure members in an inline assembly statement in the usual fashion (that is, *variable.member*). In such a case, you are dealing with a variable, and you can store or retrieve values. However, you can also directly reference the member name (without the variable name) as a form of numeric constant. In this situation, the constant equals the offset (in bytes) from the start of the structure containing that member. Consider the following program fragment:

```
struct myStruct {
    int a_a;
    int a_b;
    int a_c;
} myA ;

myfunc()
{
    ...
    asm {mov ax, myA.a_b
        mov bx, [di].a_b
        }
    ...
}
```

We've declared a structure type named *myStruct* with three members, *a\_a*, *a\_b*, and *a\_c*; we've also declared a variable *myA* of type *myStruct*. The first inline assembly statement moves the value contained in *myA.a\_b* into the register AX. The second moves the value at the address  $[di] + \text{offset}(a_c)$  into the register BX (it takes the address stored in DI and adds to it the offset of *a\_c* from the start of *myStruct*). In this sequence, these assembler statements produce the following code:

```
mov ax, DGROUP : myA+2
mov bx, [di+4]
```

Why would you even want to do this? If you load a register (such as DI) with the address of a structure of type *myStruct*, you can use the member names to directly reference the members. The member name actually may be used in any position where a numeric constant is allowed in an assembly statement operand.

The structure member must be preceded by a dot (.) to signal that a member name, rather than a normal C symbol, is being used. Member names are replaced in the assembly output by the numeric offset of the structure member (the numeric offset of *a\_c* is 4), but no type information is retained. Thus members may be used as compile-time constants in assembly statements.

However, there is one restriction. If two structures that you are using in inline assembly have the same member name, you must distinguish between them. Insert the structure type (in parentheses) between the dot and the member name, as if it were a cast. For example,

```
asm mov bx, [di].(struct tm)tm_hour
```

## Using jump instructions and labels

You can use any of the conditional and unconditional jump instructions, plus the loop instructions, in inline assembly. They are only valid inside a function. Since no labels can be defined in the **asm** statements, jump instructions must use C **goto** labels as the object of the jump. Direct far jumps cannot be generated.

In the following code, the jump goes to the C **goto** label *a*.

```
int x()
{
a:                               /* This is the goto label "a" */
  ...
}
```

```

asm jmp a          /* Goes to label "a" */
...
}

```

Indirect jumps are also allowed. To use an indirect jump, you can use a register name as the operand of the jump instruction.

## Interrupt functions

---

The 8086 reserves the first 1024 bytes of memory for a set of 256 far pointers—known as interrupt vectors—to special system routines known as *interrupt handlers*. These routines are called by executing the 8086 instruction

```
int int#
```

where *int#* goes from 0h to FFh. When this happens, the computer saves the code segment (CS), instruction pointer (IP), and status flags, disables the interrupts, then does a far jump to the location pointed to by the corresponding interrupt vector. For example, one interrupt call you're likely to see is

```
int 21h
```

which calls most DOS routines. But many of the interrupt vectors are unused, which means, of course, that you can write your own interrupt handler and put a **far** pointer to it into one of the unused interrupt vectors.

To write an interrupt handler in Turbo C++, you must define the function to be of type **interrupt**; more specifically, it should look like this:

```

void interrupt myhandler(bp, di, si, ds, es, dx,
                        cx, bx, ax, ip, cs, flags, ... );

```

As you can see, all the registers are passed as parameters, so you can use and modify them in your code without using the pseudo-variables discussed earlier in this chapter. You can also pass additional parameters (*flags, ...*) to the handler; those should be defined appropriately.

A function of type **interrupt** will automatically save (in addition to SI, DI, and BP) the registers AX through DX, ES, and DS. These same registers are restored on exit from the interrupt handler.

Interrupt handlers may use floating-point arithmetic in all memory models. Any interrupt handler code that uses an 80x87 must

save the state of the chip on entry and restore it on exit from the handler.

An interrupt function can modify its parameters. Changing the declared parameters will modify the corresponding register when the interrupt handler returns. This may be useful when you are using an interrupt handler to act as a user service, much like the DOS INT 21 services. Also, note that an interrupt function exits with an IRET (return from interrupt) instruction.

So, why would you want to write your own interrupt handler? For one thing, that's how most memory-resident routines work. They install themselves as interrupt handlers. That way, whenever some special or periodic action takes place (clock tick, keyboard press, and so on), these routines can intercept the call to the routine handling the interrupt and see what action needs to take place. Having done that, they can then pass control on to the routine that was there.

---

## Using low-level practices

You've already seen a few examples of how to use these different low-level practices in your code; now it's time to look at a few more. Let's start with an interrupt handler that does something harmless but tangible (or, in this case, audible): It beeps whenever it's called.

First, write the function itself. Here's what it might look like:

```
#include <dos.h>

void interrupt mybeep(unsigned bp, unsigned di, unsigned si,
                    unsigned ds, unsigned es, unsigned dx,
                    unsigned cx, unsigned bx, unsigned ax)
{
    int i, j;
    char originalbits, bits;
    unsigned char bcount = ax >> 8;

    /* Get the current control port setting */
    bits = originalbits = inportb(0x61);

    for (i = 0; i <= bcount; i++){

        /* Turn off the speaker for awhile */
        outportb(0x61, bits & 0xfc);
        for (j = 0; j <= 100; j++)
            ; /* empty statement */

        /* Now turn it on for some more time */
```

```

        outportb(0x61, bits | 2);
        for (j = 0; j <= 100; j++)
            ; /* another empty statement */
    }

    /* Restore the control port setting */
    outportb(0x61, originalbits);
}

```

Next, write a function to install your interrupt handler. Pass it the address of the function and its interrupt number (0 to 255 or 0x00 to 0xFF).

```

void install(void interrupt (*faddr)(), int inum)
{
    setvect(inum, faddr);
}

```

Finally, call your beep routine to test it out. Here's a function to do just that:

```

void testbeep(unsigned char bcount, int inum)
{
    _AH = bcount;
    geninterrupt(inum);
}

```

Your **main** function might look like this:

```

main()
{
    char ch;

    install(mybeep,10);
    testbeep(3,10);
    ch = getch();
}

```

You might also want to preserve the original interrupt vector and restore it when your main program is finished. Use the **getvect** and **setvect** functions to do this.



## *Error messages*

Turbo C++ has two categories of errors: run time and compile time. Run-time error messages immediately follow. Compile-time error messages fall into three classes: fatal errors, nonfatal errors, and warnings. These are explained and discussed in more detail starting on page 283.

The following generic names and values are some of those that appear in the error messages listed in this chapter (most are self-explanatory). When you get an error message, the appropriate name or value is substituted.

<b>What you'll see in the manual</b>	<b>What you'll see on your screen</b>
<i>argument</i>	A command-line or other argument
<i>class</i>	A class name
<i>field</i>	A field reference
<i>filename</i>	A file name (with or without extension)
<i>group</i>	A group name
<i>identifier</i>	An identifier (variable name or other)
<i>language</i>	The name of a programming language
<i>member</i>	The name of a data member or member function
<i>message</i>	A message string
<i>module</i>	A module name
<i>number</i>	An actual number
<i>option</i>	A command-line or other option
<i>parameter</i>	A parameter name

What you'll see in the manual	What you'll see on your screen
<i>segment</i>	A segment name
<i>specifier</i>	A type specifier
<i>symbol</i>	A symbol name
XXXXh	A 4-digit hexadecimal number, followed by <i>h</i>

The error messages are listed in ASCII alphabetic order; messages beginning with symbols (equal signs, commas, braces, and so on) normally come first. Since messages that begin with one of the variables just listed cannot be alphabetized by what you will actually see when you receive such a message, all such messages have been placed at the beginning of each error message list.

For example, if you have a C++ function **goforit**, you might receive the following actual message:

```
goforit must be declared with no arguments
```

In order to look this error message up, you would need to find

***function must be declared with no arguments***

near the beginning of the list of error messages.

If the variable occurs later in the text of the error message (for example, "Incorrect command-line argument: *argument'*"), you can find the message in correct alphabetical order; in this case, under *I*.

## Run-time error messages

Turbo C++ has a small number of run-time error messages. These errors occur after the program has successfully compiled and while it is running. These errors are listed alphabetically and explained in this section.

*These errors can happen through memory overwrites.*

### **Abnormal program termination**

This message can be generated if there is not enough memory to execute the program within. This message is covered in more detail at the end of the floating-point errors section. Calling **abort** activates the message.

### **Divide by 0**

This message is generated by an integer divide by zero, such as

```
int n = 0;
n = 2 / n;
```

It can be trapped with the signal function. Otherwise, **abort** is called and your program terminates.

#### **Floating point error: Divide by 0.**

#### **Floating point error: Domain.**

#### **Floating point error: Overflow.**

These fatal errors result from a floating-point operation for which the result is not finite.

- “Divide by 0” means the result is +INF or –INF exactly, such as 1.0/0.0.
- “Domain” means the result is NAN (not a number), like 0.0 /0.0.
- “Overflow” means the result is +INF (infinity) or –INF with complete loss of precision, such as assigning 1e200\*1e200 to a **double**.

#### **Floating point error: Partial loss of precision.**

#### **Floating point error: Underflow.**

These exceptions are masked by default, and the error messages do not occur. Underflows are converted to zero and losses of precision are ignored. They can be unmasked by calling **\_control87**.

#### **Floating point error: Stack fault.**

This message indicates that the floating-point stack has been overrun. This error does not normally occur and may be due to assembly code using too many registers or due to a misdeclaration of a floating-point function.

These floating-point errors can be avoided by masking the exception so that it doesn’t occur, or by catching the exception with **signal**. See the functions **\_control87** and **signal** (in the *Library Reference*) for details.

In each of the above cases, the program prints the error message and then calls **abort**, which prints

```
Abnormal program termination
```

and calls **\_exit(3)**. See **abort** and **\_exit** for more details.

#### **Null pointer assignment**

When a small or medium memory model program exits, a check is made to determine if the contents of the first few bytes

within the program's data segment have changed. These bytes would never be altered by a working program. If they have been changed, the message "Null pointer assignment" is displayed to inform you that (most likely) a value was stored to an uninitialized pointer. The program may appear to work properly in all other respects; however, this is a serious bug which should be attended to immediately. Failure to correct an uninitialized pointer can lead to unpredictable behavior (including "locking" the computer up in the large, compact, and huge memory models). You can use the integrated debugger to track down null pointers.

### Stack overflow

The default stack size for Turbo C++ programs is 4,096 bytes. This should be enough for most programs, but those which execute recursive functions or store a great deal of local data can overflow the stack. You will only get this message if you have stack checking enabled. If you do get this message, you can try switching to a larger memory model, increasing the stack size, or decreasing your program's dependence on the stack. See Chapter 2, "Global variables," in the *Library Reference* for information on changing the stack size by altering the global variable `_stklen`. To decrease the amount of local data used by a function, look at the example below. The variable `buffer` has been declared static and does not consume stack space like `list` does.

```
void anyfunction( void )
{
    static int buffer[ 2000 ]; /* resides in the data segment */
    int list[ 2000 ];         /* resides on the stack */
}
```

There are two disadvantages to declaring local variables as static.

1. It now takes permanent space away from global variables and the heap. (You have to rob Peter to pay Paul.) This is usually only a minor disadvantage.
2. The function may no longer be reentrant. What this means is that if the function is called recursively or asynchronously and it is important that each call to the function have its own unique copy of the variable, you cannot make it static. This is because every time the function is called, it will use the same exact memory space

for the variable, rather than allocating new space for it on each call. You could have a sharing problem if the function is trying to execute from within itself (recursively) or at the same time as itself (asynchronously). For most DOS programs this is not a problem. If you aren't writing recursive functions or trying to multitask, don't worry. If you are, the preceding explanation will make sense to you.

## Compiler error messages

---

The Turbo C++ compiler diagnostic messages fall into three classes: fatal errors, errors, and warnings.

**Fatal errors** are rare. Some of them indicate an internal compiler error. When a fatal error occurs, compilation stops immediately. You must take appropriate action and then restart compilation.

**Errors** indicate program syntax errors, disk or memory access errors, and command line errors. The compiler will complete the current phase of the compilation and then stop. The compiler attempts to find as many real errors in the source program as possible during each phase (preprocessing, parsing, optimizing and code-generating).

**Warnings** do not prevent the compilation from finishing. They indicate conditions which are suspicious, but which are legitimate as part of the language. The compiler also produces warnings if you use machine-dependent constructs in your source files.

The compiler prints messages with the message class first, then the source file name and line number where the compiler detected the condition, and finally the text of the message itself.

In the following lists, messages are presented alphabetically within message class. With each message, a probable cause and remedy are provided.

You should be aware of one detail about line numbers in error messages: the compiler only generates messages as they are detected. Because C does not force any restrictions on placing statements on a line of text, the true cause of the error may be one or more lines before the line number mentioned. In the following message list, we have indicated those messages which often appear (to the compiler) to be on lines after the real cause.

## Fatal errors

---

### **Bad call of inline function**

You have used an inline function taken from a macro definition, but have called it incorrectly. An inline function in C is one that begins and ends with a double underscore ( `__` ).

### **Irreducible expression tree**

This is a sign of some form of compiler error. Some expression on the indicated line of the source file has caused the code generator to be unable to generate code. Whatever the offending expression is, it should be avoided. Notify Borland if the compiler ever encounters this error.

### **Out of memory**

The total working storage is exhausted. Compile the file on a machine with more memory. If you already have 640K, you may have to simplify the source file.

### **Register allocation failure**

This is a sign of some form of compiler error. Some expression on the indicated line of the source file was so complicated that the code generator could not generate code for it. Simplify the offending expression. If this fails to solve the problem, avoid the expression. Notify Borland if the compiler encounters this error.

## Errors

---

### ***constructor* cannot return a value**

A C++ constructor can't have an expression in a return statement.

### ***constructor* is not a base class of *class***

A C++ class constructor *class* is trying to call a base class constructor *constructor*, or you are trying to change the access rights of *class::constructor*. *constructor* is not a base class of *class*. Check your declarations.

### ***function1* cannot be distinguished from *function2***

The parameter type lists in the declarations of these two functions do not differ enough to tell them apart. Try changing the order of parameters or the type of a parameter in one declaration.

***function is ambiguous***

In this call of function *function*, more than one overloaded function matches the list of arguments (using default argument conversions). Use an explicit cast with one or more of the arguments to resolve the ambiguity.

***function must be declared with no arguments***

This C++ operator function was incorrectly declared with arguments.

***function must be declared with one argument***

This C++ operator function was incorrectly declared with more than one argument.

***function must be declared with two arguments***

This C++ operator function was incorrectly declared with other than two arguments.

***function was previously declared without static***

This function has been declared static here, but was declared **extern** (or global) earlier. ANSI C does not allow the mixing of these declarations.

***function was previously declared with the language *language****

Only one language modifier (**cdecl**, **pascal**, or **interrupt**) can be given for a function. This function has been declared with different language modifiers in two locations.

***identifier cannot be declared in an anonymous union***

The compiler found a declaration for a member function or static member in an anonymous union. Such unions can only contain data members.

***identifier cannot be used in a static member function***

A static member function can only use a static member of its class, though it has full access rights. This error is the result of trying to use a member which requires a **this** pointer.

***identifier is inaccessible because also in *class****

It is not legal to use a class as both a direct and indirect base class, since the fields are automatically ambiguous. Try making the base class virtual in both locations.

***identifier is not a data member and can't be initialized here***

Only data members can be initialized in the initializers of a constructor. This message means that the list includes a static member or function member.

**identifier is not a member of struct**

You are trying to reference *identifier* as a member of **struct**, but it is not a member. Check your declarations.

**identifier is not a parameter**

In the parameter declaration section of an old-style function definition, *identifier* is declared but is not listed as a parameter. Either remove the declaration or add *identifier* as a parameter.

**identifier is not legal here**

Type specifier *identifier* is not legal because it conflicts with or duplicates another type specifier in this declaration, or because *identifier* is being used as a **typedef** name when it is not a **typedef** name in this scope.

**identifier is virtual and cannot be explicitly initialized**

A C++ class constructor is trying to call a base class constructor *identifier*, but *identifier* is a virtual base class. Virtual base classes cannot be explicitly initialized. The compiler implicitly calls the base class default constructor `base::base()` for you.

**identifier must be a member function**

Most C++ operator functions may be members of classes or ordinary nonmember functions, but certain ones are required to be members of classes. These are **operator =**, **operator ->**, **operator ()**, and type conversions. This operator function is not a member function but should be.

**identifier must be a member function or have an argument of class type**

Most C++ operator functions must have an implicit or explicit argument of class type. This operator function was declared outside a class and does not have an explicit argument of class type.

**identifier must be a previously defined class or struct**

You are attempting to declare *identifier* to be a base class, but either it is not a class or it has not yet been fully defined. Correct the name or rearrange the declarations.

**identifier must be a previously defined enumeration tag**

This declaration is attempting to reference *identifier* as the tag of an **enum** type, but it has not been so declared. Correct the name, or rearrange the declarations.

**identifier must be a previously defined structure tag**

This declaration is attempting to reference *identifier* as the tag of a **struct** type, but it has not been so declared. Correct the name, or rearrange the declarations.

**identifier specifies multiple or duplicate access**

A base class may be declared **public** or **private**, but not both. This access specifier may appear no more than once for a base class.

**member is not accessible**

You are trying to reference C++ class member *member*, but it is **private** or **protected** and cannot be referenced from this function. This sometimes happens when attempting to call one accessible overloaded member function (or constructor), but the arguments match an inaccessible function. The check for overload resolution is always made before checking for accessibility. If this is the problem, try an explicit cast of one or more parameters to select the desired accessible function.

**specifier has already been included**

This type specifier occurs more than once in this declaration. Delete or change one of the occurrences.

**= expected**

An assignment operator was expected for initializing a variable.

**, expected**

A comma was expected in a list of declarations, initializations, or parameters.

**{ expected**

A left brace was expected at the start of a block or initialization.

**( expected**

A left parenthesis was expected before a parameter list.

**} expected**

A right brace was expected at the end of a block or initialization.

**) expected**

A right parenthesis was expected at the end of a parameter list.

**: expected after private/protected/public**

When used to begin a **private/protected/public** section of a C++ class, these reserved words must be followed by a colon.

**:: requires a preceding identifier in this context**

A C++ double colon was encountered in a declaration without a preceding qualifying class name. The unqualified double colon can be used only in expressions to indicate global scope, not in declarations.

**. \* operands do not match**

You have not declared the right side of a C++ dot-star (.\* ) operator as a pointer to a member of the class specified by the left side of the operator.

**#operator not followed by macro argument name**

In a macro definition, the # may be used to indicate turning a macro argument into a string. The # must be followed by a macro argument name.

**Access can only be changed to public or protected**

A C++ derived class may modify the access rights of a base class member, but only to **public** or **protected**. A base class member cannot be made **private**.

**Access declarations cannot grant or reduce access**

A C++ derived class can modify the access rights of a base class member, but only by restoring it to the rights in the base class. It cannot add or reduce access rights.

**Access specifier *specifier* found in a union**

The C++ access specifiers (**public**, **private**, or **protected**) cannot be used in unions.

**Ambiguity between *function1* and *function2***

Both of the named overloaded functions could be used with the supplied parameters. This ambiguity is not allowed.

**Ambiguous conversion functions: *type1* and *type2***

The compiler found more than one way to convert the provided type to the desired type. This ambiguity is not allowed.

**Array bounds missing ]**

Your source file declared an array in which the array bounds were not terminated by a right bracket.

**Array must have at least one element**

ANSI C and C++ require that an array be defined to have at least one element (objects of zero size are not allowed). An old programming trick declares an array element of a structure to have zero size, then allocates the space actually needed with

**malloc.** You can still use this trick, but you must declare the array element to have (at least) one element. Declarations (as opposed to definitions) of arrays of unknown size are still allowed, of course.

For example,

```
char ray[];          /* definition of unknown size -- illegal */
char ray[0];        /* definition of 0 size -- illegal */
extern char ray[];  /* declaration of unknown size -- ok */
```

### **Array of references is not allowed**

It is illegal to have an array of references, since pointers to references are not allowed and array names are coerced into pointers.

### **Array size too large**

The declared array is larger than 64K.

### **Assembler statement too long**

Inline assembly statements may not be longer than 480 bytes.

### **Attempting to return a reference to local name *identifier***

This C++ function returns a reference type, and you are trying to return a reference to a local (auto) variable. This is illegal, since the variable referred to disappears when the function exits. You may return a reference to any static or global variable, or you may change the function to return a value instead.

### **Bad file name format in include directive**

Include file names must be surrounded by quotes ("*filename.h*") or angle brackets (<*filename.h*>). The file name was missing the opening quote or angle bracket. If a macro was used, the resulting expansion text is incorrect; that is, not surrounded by quote marks.

### **Bad `ifdef` directive syntax**

An `ifdef` directive must contain a single identifier (and nothing else) as the body of the directive.

### **Bad `ifndef` directive syntax**

An `ifndef` directive must contain a single identifier (and nothing else) as the body of the directive.

### **Bad return type for a type conversion operator**

This C++ type conversion member function specifies a return type different from the type itself. A declaration for conversion function **operator T** may not specify any return type.

**Bad syntax for pure function definition**

Pure virtual functions are specified by appending “= 0” to the declaration. You wrote something similar, but not quite the same.

**Bad undef directive syntax**

An **#undef** directive must contain a single identifier (and nothing else) as the body of the directive.

**Base class *class* is included more than once**

A C++ class may be derived from any number of base classes, but may be directly derived from a given class only once.

**Base class *class* is initialized more than once**

In a C++ class constructor, the list of initializations following the constructor header includes base class *class* more than once.

**Base class cannot be declared protected**

A C++ base class may be **public** or **private**, but not **protected**.

**Bit field cannot be static**

Only ordinary C++ class data members can be declared **static**, not bit fields.

**Bit fields must be signed or unsigned int**

A bit field must be declared to be a signed or unsigned integral type. In ANSI C, bit fields may only be signed or unsigned **int** (not **char** or **long**, for example).

**Bit fields must contain at least one bit**

You cannot declare a named bit field to have 0 (or less than 0) bits. You can declare an unnamed bit field to have 0 bits, a convention used to force alignment of the following bit field to a byte boundary (or word boundary, if the **-a** alignment option is selected).

**Bit field too large**

This error occurs when you supply a bit field with more than 16 bits.

**Body already defined for this function**

A function with this name and type was previously supplied a function body. A function body can only be supplied once.

**Call of non-function**

The name being called is not declared as a function. This is commonly caused by incorrectly declaring the function or misspelling the function name.

**Cannot assign *identifier1* to *identifier2***

Both sides of an assignment (=) operator (or compound assignment operator such as +=) must be compatible and must not be arrays. The right side of this assignment has type *identifier1*, and cannot be assigned to the object on the left, which is type *identifier2*.

**Cannot call 'main' from within the program**

C++ does not allow recursive calls of **main**.

**Cannot cast from *identifier1* to *identifier2***

A cast from type *identifier1* to type *identifier2* is not allowed. In C, a pointer may be cast to an integral type or to another pointer. An integral type may be cast to any integral, floating, or pointer type. A floating type may be cast to an integral or floating type. Structures and arrays may not be cast to or from. You usually cannot cast from a **void** type.

In C++, user-defined conversions and constructors are checked for, and if one cannot be found, then the preceding rules apply (except for pointers to class members). Among integral types, only a constant zero may be cast to a member pointer. A member pointer may be cast to an integral type or to a similar member pointer. A similar member pointer points to a data member if the original does, or to a function member if the original does; the qualifying class of the type being cast to must be the same as or a base class of the original.

**Cannot create a variable for abstract class *class***

Abstract classes—those with pure virtual functions—cannot be used directly, only derived from.

**Cannot define a pointer or reference to a reference**

It is illegal to have a pointer to a reference or a reference to a reference.

**Cannot find *class::class (class &)* to copy a vector**

When a C++ class *class1* contains a vector (array) of class *class2*, and you want to construct an object of type *class1* from another object of type *class1*, there must be a constructor ***class2::class2(class2&)*** so that the elements of the vector can be constructed. This constructor takes just one parameter (which is a reference to its class) and is called a *reference constructor*.

Usually the compiler supplies a reference constructor automatically. However, if you have defined a constructor for class

*class2* that has a parameter of type *class2&* and has additional parameters with default values, the reference constructor cannot exist and cannot be created by the compiler. (This is because `class2::class2(class2&)` and `class2::class2(class2&, int = 1)` cannot be distinguished.) You must redefine this constructor so that not all parameters have default values. You can then define a reference constructor or let the compiler create one.

#### **Cannot find *identifier::identifier()* to initialize a vector**

When a C++ class *class1* contains a vector (array) of class *class2*, and you want to construct an object of type *class1* but not from another object of type *class1*, there must be a constructor **`class2::class2()`** so that the elements of the vector can be constructed. This constructor without parameters is called the default constructor. The compiler will supply a default constructor automatically unless you have defined any constructor for class *class2*; in that case, the compiler will not supply the default constructor automatically—you must supply one.

#### **Cannot find *class::class()* to initialize base class**

Whenever a C++ derived class *class2* is constructed, each base class *class1* must first be constructed. If the constructor for *class2* does not specify a constructor for *class1* (as part of *class2*'s header), there must be a constructor **`class1::class1()`** for the base class. This constructor without parameters is called the default constructor. The compiler will supply a default constructor automatically unless you have defined any constructor for class *class1*; in that case, the compiler will not supply the default constructor automatically—you must supply one.

#### **Cannot find *class::class()* to initialize field *identifier***

When a C++ class *class1* contains a member of class *class2*, and you wish to construct an object of type *class1* but not from another object of type *class1*, there must be a constructor **`class2::class2()`** so that the member can be constructed. This constructor without parameters is called the default constructor. The compiler will supply a default constructor automatically unless you have defined any constructor for class *class2*; in that case, the compiler will not supply the default constructor automatically—you must supply one.

**Cannot find `class::operator=(class&)` to copy a vector**

When a C++ class *class1* contains a vector (array) of class *class2*, and you wish to copy a class of type *class1*, there must be an assignment operator **`class2::operator=(class2&)`** so that the elements of the vector can be copied. Usually the compiler supplies such an operator automatically. However, if you have defined an **`operator=`** for class *class2*, but not one that takes a parameter of type *class2&*, the compiler will not supply it automatically—you must supply one.

**Cannot have a near member in a far class**

All members of a C++ **far** class must be far. This member is a class that was declared (or defaults to) **near**.

**Cannot initialize a field**

Individual fields of **structs**, **unions**, and C++ **classes** may not have initializers. A **struct** or **union** may be initialized as a whole using initializers inside braces. A C++ **class** may only be initialized by the use of a constructor.

**Cannot initialize `type1` with `type2`**

You are attempting to initialize an object of type *type1* with a value of type *type2*, which is not allowed. The rules for initialization are essentially the same as for assignment.

**Cannot modify a const object**

This indicates an illegal operation on an object declared to be **const**, such as an assignment to the object.

**Cannot overload 'main'**

**main** is the only function which cannot be overloaded.

**Cannot specify base classes except when defining the class**

When specifying a C++ class, the base classes from which this class is derived may be specified only at the point of the class definition. When you only declare the class tag, as in `class c;`, you cannot specify the base classes.

**Case outside of switch**

The compiler encountered a **case** statement outside a **switch** statement. This is often caused by mismatched braces.

**Case statement missing :**

A **case** statement must have a constant expression followed by a colon. The expression in the **case** statement either was missing a colon or had an extra symbol before the colon.

**Character constant too long**

Character constants may only be one or two characters long.

**Class *class* has a constructor and cannot be hidden**

C has separate name spaces for structure tags and ordinary names, and C++ usually allows the same sort of confusion. It draws the line at classes with constructors, since constructor declarations look like function declarations, and the confusion could become terminal.

**Classes cannot be initialized with {}**

Ordinary C structures can be initialized with a set of values inside braces. C++ classes can only be initialized with constructors if the class has constructors, private members, functions or base classes which are virtual.

**Class member *member* declared outside its class**

C++ class member functions can be declared only inside the class declaration. Unlike nonmember functions, they cannot be declared multiple times or at other locations.

**Compound statement missing }**

The compiler reached the end of the source file and found no closing brace. This is most commonly caused by mismatched braces.

**Conflicting type modifiers**

This occurs when a declaration is given that includes, for example, both **near** and **far** keywords on the same pointer. Only one addressing modifier may be given for a single pointer, and only one language modifier (**cdecl**, **pascal**, or **interrupt**) may be given for a function.

**Constant expression required**

Arrays must be declared with constant size. This error is commonly caused by misspelling a **#define** constant.

**Constructor cannot have a return type specification**

C++ constructors have an implicit return type used by the compiler, but you cannot declare a return type or return a value.

**Conversion of near pointer not allowed**

A near pointer cannot be converted to a far pointer in the expression evaluation box when a program is not currently running. This is because the conversion needs the current value of DS in the user program, which doesn't exist.

**Could not find a match for *argument(s)***

No C++ function could be found with parameters matching the supplied arguments.

**Could not find file *filename***

The compiler is unable to find the file supplied on the command line.

**Declaration does not specify a tag or an identifier**

This declaration doesn't declare anything. This may be a **struct** or **union** without a tag or a variable in the declaration. C++ requires that something be declared.

**Declaration is not allowed here**

Declarations cannot be used as the control statement for **while**, **for**, **do**, **if**, or **switch** statements.

**Declaration missing ;**

Your source file contained a **struct** or **union** field declaration that was not followed by a semicolon.

**Declaration syntax error**

Your source file contained a declaration that was missing some symbol or had some extra symbol added to it.

**Declaration terminated incorrectly**

A declaration has an extra or incorrect termination symbol, such as a semicolon placed after a function body. A C++ member function declared in a class with a semicolon between the header and the opening left brace also generates this error.

**Declaration was expected**

A declaration was expected here but not found. This is usually caused by a missing delimiter such as a comma, semicolon, right parenthesis, or right brace.

**Declare operator *delete* (void\*) or (void\*, size\_t)**

Declare the operator **delete** with a single **void\*** parameter, or with a second parameter of type **size\_t**. If you use the second version, it will be used in preference to the first version. The global operator **delete** is already declared with two parameters, so be careful if you plan to override this declaration.

**Default outside of switch**

The compiler encountered a **default** statement outside a **switch** statement. This is most commonly caused by mismatched braces.

### **Default value missing**

When a C++ function declares a parameter with a default value, all of the following parameters must also have default values. In this declaration, a parameter with a default value was followed by a parameter without a default value.

### **Define directive needs an identifier**

The first non-whitespace character after a `#define` must be an identifier. The compiler found some other character.

### **Destructor cannot have a return type specification**

C++ destructors never return a value, and you cannot declare a return type or return a value.

### **Destructor for *class* is not accessible**

The destructor for this C++ class is **protected** or **private**, and cannot be accessed here to destroy the class. If a class destructor is **private**, the class cannot be destroyed, and thus can never be used. This is probably an error. A **protected** destructor can be accessed only from derived classes. This is a useful way to ensure that no instance of a base class is ever created, but only classes derived from it.

### **Destructor name must match the class name**

In a C++ class, the tilde (~) introduces a declaration for the class destructor. The name of the destructor must be same as the class name. In your source file, the ~ preceded some other name.

### **Division by zero**

Your source file contained a divide or remainder in a constant expression with a zero divisor.

### **do statement must have while**

Your source file contained a **do** statement that was missing the closing **while** keyword.

### **do-while statement missing (**

In a **do** statement, the compiler found no left parenthesis after the **while** keyword.

### **do-while statement missing )**

In a **do** statement, the compiler found no right parenthesis after the test expression.

### **do-while statement missing ;**

In a **do** statement test expression, the compiler found no semicolon after the right parenthesis.

**Duplicate case**

Each **case** of a **switch** statement must have a unique constant expression value.

**Enum syntax error**

An **enum** declaration did not contain a properly formed list of identifiers.

**Error directive: *message***

This message is issued when an **#error** directive is processed in the source file. The text of this directive is displayed in this message.

**Error writing output file**

A DOS error that prevents Turbo C++ from writing an **.OBJ**, **.EXE**, or temporary file. Check the **-n** or **Options | Directories | Output** directory and make sure that this is a valid directory. Also check that there is enough free disk space.

**Expression expected**

An expression was expected here, but the current symbol cannot begin an expression. This message may occur where the controlling expression of an **if** or **while** clause is expected or where a variable is being initialized. It is often due to an accidentally inserted or deleted symbol in the source code.

**Expression is too complicated**

The compiler can handle very complex expressions, but some expressions with hundreds of terms might prove too complicated. Split the expression into two or more statements.

**Expression of arithmetic type expected**

The unary plus (+) and minus (-) operators require an expression of arithmetic type—only types **char**, **short**, **int**, **long**, **enum**, **float**, **double**, and **long double** are allowed.

**Expression of integral type expected**

The complement (~) operator requires an expression of integral type—only types **char**, **short**, **int**, **long**, or **enum** are allowed.

**Expression of scalar type expected**

The not (!), increment (++), and decrement (--) operators require an expression of scalar type—only types **char**, **short**, **int**, **long**, **enum**, **float**, **double**, **long double**, and pointer types are allowed.

**Expression syntax**

This is a catchall error message when the compiler parses an expression and encounters some serious error. This is most commonly caused by two consecutive operators, mismatched or missing parentheses, or a missing semicolon on the previous statement.

**Expression type does not match the return type**

The type of the **return** expression cannot be converted to the **return** type of the function.

**extern variable cannot be initialized**

The storage class **extern** applied to a variable means that the variable is being declared but not defined here—no storage is being allocated for it. Therefore, you can't initialize the variable as part of the declaration.

**Extra parameter in call**

A call to a function, via a pointer defined with a prototype, had too many arguments given.

**Extra parameter in call to function**

A call to the named function (which was defined with a prototype) had too many arguments given in the call.

**Field *field* cannot be used without an object**

This means that the user has written *class::field* where *field* is an ordinary (non-static) member, and there is no class to associate with that field. For example, it is legal to write *obj.class::field*, but not to write *class::field*.

**Field *field* is ambiguous in class**

You must qualify the field reference with the appropriate base class name. In C++ class *class*, field *field* can be found in more than one base class, and was not qualified to indicate which was meant. This applies only in multiple inheritance, where the field name in each base class is not hidden by the same field name in a derived class on the same path. The C++ language rules require that this test for ambiguity be made before checking for access rights (**private**, **protected**, **public**). It is therefore possible to get this message even though only one (or none) of the fields can be accessed.

**Field identifier expected**

The name of a structure or C++ class field was expected here, but not found. The right side of a dot (.) or arrow (->) operator

must be the name of a field in the structure or class on the left of the operator.

**File must contain at least one external declaration**

This compilation unit was logically empty, containing no declarations. ANSI C and C++ require that something be declared in the compilation unit.

**File name too long**

The file name given in an **#include** directive was too long for the compiler to process. File names in DOS must be no more than 79 characters long.

**For statement missing (**

In a **for** statement, the compiler found no left parenthesis after the **for** keyword.

**For statement missing )**

In a **for** statement, the compiler found no right parenthesis after the control expressions.

**For statement missing ;**

In a **for** statement, the compiler found no semicolon after one of the expressions.

**Found : instead of ::**

You used a colon instead of a double colon (::) to separate a C++ class qualifier from its field in declarations and expressions.

**Friend declarations need a function signature**

If you declare a friend function, you must provide the parameter types as well so the proper function can be found among the overload cohort.

**Friends must be functions or classes, not fields**

A **friend** of a C++ class must be a function or another class; a field cannot be a **friend**.

**Function call missing )**

The function call argument list had some sort of syntax error, such as a missing or mismatched right parenthesis.

**Function calls not supported**

In integrated debugger expression evaluation, calls to functions (including implicit conversion functions, constructors, destructors, overloaded operators, and inline functions) are not supported.

### Function defined inline after use as extern

Functions cannot become inline after they have already been used. Either move the inline definition forward in the file or delete it entirely.

### Function definition cannot be a typedef'ed declaration

Declarations of pointers to functions are made more readable by using **typedefs**. But in C++, such **typedefs** cannot be used to define functions.

For example, here type *F* is a function with no parameters returning an **int**:

```
typedef int F(void);
```

Here it's illegal to define *g* as such a function:

```
F g { /* ... */ }
```

And here it's OK to have *g* as a function returning a pointer to type *F*.

```
F *g(...) { /* ... */ }
```

### Function *function* cannot be static

Only ordinary member functions and the operators **new** and **delete** can be declared static. Constructors, destructors and other operators must not be static.

### Functions cannot return arrays or functions

This function was declared to return a function or array rather than a pointer to a function or a pointer to an array element.

### Function should return a value

This function was declared (maybe implicitly) to return a value. A return statement was found without a return value or the end of the function was reached without a return statement being found. Either return a value or change the function declaration to return **void**.

### Functions may not be part of a struct or union

This C **struct** or **union** field was declared to be of type function rather than pointer to function. Functions as fields are allowed only in C++.

### Global anonymous union not static

In C++, a global anonymous union at the file level must be static.

**Goto statement missing label**

The **goto** keyword must be followed by an identifier.

**Group overflowed maximum size: *name***

The total size of the segments in a group (for example, DGROUP) exceeded 64K.

**Identifier *identifier* cannot have a type qualifier**

A C++ qualifier *class::identifier* may not be applied here. A qualifier is not allowed on **typedef** names, on function declarations (except definitions at the file level), on local variables or parameters of functions, or on a class member except to use its own class as a qualifier (redundant but legal).

**Identifier expected**

An identifier was expected here, but not found. In C, this is in a list of parameters in an old-style function header, after the reserved words **struct** or **union** when the braces are not present, and as the name of a field in a structure or union (except for bit fields of width 0). In C++, an identifier is also expected in a list of base classes from which another class is derived, following a double colon (::) and after the reserved word **operator** when no operator symbol is present.

**If statement missing (**

In an **if** statement, the compiler found no left parenthesis after the **if** keyword.

**If statement missing )**

In an **if** statement, the compiler found no right parenthesis after the test expression.

**Illegal character *character* (0x*value*)**

The compiler encountered some invalid character in the input file. The hexadecimal value of the offending character is printed. This can also be caused by extra parameters passed to a function macro.

**Illegal initialization**

Initializations must be either constant expressions, or else the address of a global **extern** or **static** variable plus or minus a constant.

**Illegal octal digit**

The compiler found an octal constant containing a non-octal digit (8 or 9).

**Illegal parameter to `__emit__`**

There are some restrictions on inserting literal values directly into your code. For example you cannot give a local variable as a parameter to `__emit__`. Refer to the `__emit__` function for further explanation.

**Illegal pointer subtraction**

This is caused by attempting to subtract a pointer from a non-pointer.

**Illegal structure operation**

Structures may only be used with dot (`.`), address-of (`&`) or assignment (`=`) operators, or be passed to or from a function as parameters. The compiler encountered a structure being used with some other operator.

**Illegal to take address of bit field**

It is not legal to take the address of a bit field, although you can take the address of other kinds of fields.

**Illegal use of floating point**

Floating-point operands are not allowed in shift, bitwise Boolean, conditional (`? :`), indirection (`*`), or certain other operators. The compiler found a floating-point operand with one of these prohibited operators.

**Illegal use of pointer**

Pointers can only be used with addition, subtraction, assignment, comparison, indirection (`*`) or arrow (`->`). Your source file used a pointer with some other operator.

**Improper use of typedef *identifier***

Your source file used a **typedef** symbol where a variable should appear in an expression. Check for the declaration of the symbol and possible misspellings.

**Improper use of a typedef symbol**

Your source file used a **typedef** symbol where a variable should appear in an expression. Check for the declaration of the symbol and possible misspellings.

**Incompatible type conversion**

The cast requested can't be done.

**Incorrect command-line option: *option***

The compiler did not recognize the command-line parameter as legal.

**Incorrect configuration file option: *option***

The compiler did not recognize the configuration file parameter as legal; check for a preceding hyphen (-).

**Incorrect number format**

The compiler encountered a decimal point in a hexadecimal number.

**Incorrect use of default**

The compiler found no colon after the **default** keyword.

**Inline assembly not allowed in an inline function**

The compiler cannot handle inline assembly statements in a C++ inline function. You could make this a macro, remove the **inline** storage class, or eliminate the inline assembly code.

**Invalid indirection**

The indirection operator (\*) requires a non-**void** pointer as the operand.

**Invalid macro argument separator**

In a macro definition, arguments must be separated by commas. The compiler encountered some other character after an argument name.

**Invalid pointer addition**

Your source file attempted to add two pointers together.

**Invalid use of dot**

An identifier must immediately follow a period operator (.).

**Items of type *type* need constructors and can't be passed with ...**

It is illegal to pass an object of a type which needs a constructor to a variable argument list (specified with...).

**Left side must be a structure**

The left side of a dot (.) operator (or C++ dot-star operator) must evaluate to a structure type. In this case it did not.

**Linkage specification not allowed**

Linkage specifications such as **extern "C"** are only allowed at the file level. Move this function declaration out to the file level.

**Lvalue required**

The left hand side of an assignment operator must be an addressable expression. These include numeric or pointer variables, structure field references or indirection through a pointer, or a subscripted array element.

**Macro argument syntax error**

An argument in a macro definition must be an identifier. The compiler encountered some non-identifier character where an argument was expected.

**Macro expansion too long**

A macro cannot expand to more than 4,096 characters.

**main must have a return type of int**

Function **main** has special requirements, one of which is that it cannot be declared with any return type other than **int**.

**May compile only one file when an output file name is given**

You have supplied an **-o** command-line option, which allows only one output file name. The first file is compiled but the other files are ignored.

**Member *member* is initialized more than once**

In a C++ class constructor, the list of initializations following the constructor header includes the same field more than once.

**Member functions can only have static storage class**

The only storage class allowed for a member function is **static**.

**Misplaced break**

The compiler encountered a **break** statement outside a **switch** or looping construct.

**Misplaced continue**

The compiler encountered a **continue** statement outside a looping construct.

**Misplaced decimal point**

The compiler encountered a decimal point in a floating-point constant as part of the exponent.

**Misplaced elif directive**

The compiler encountered an **#elif** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.

**Misplaced else**

The compiler encountered an **else** statement without a matching **if** statement. An extra **else** statement could cause this message, but it could also be caused by an extra semicolon, missing braces, or some syntax error in a previous **if** statement.

**Misplaced else directive**

The compiler encountered an **#else** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.

**Misplaced endif directive**

The compiler encountered an **#endif** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.

**Multiple base classes require explicit class names**

In a C++ class constructor, each base class constructor call in the constructor header must include the base class name when there is more than one immediate base class.

**Multiple declaration for identifier**

This identifier was improperly declared more than once. This might be caused by conflicting declarations such as `int a;` `double a;`, a function declared two different ways, or a label repeated in the same function, or some declaration repeated other than an **extern** function or a simple variable.

**Multiple scope qualifiers**

This C++ identifier was qualified with more than one class name; at most one class can qualify an identifier.

**Must take address of a memory location**

Your source file used the address-of operator (**&**) with an expression which cannot be used that way; for example, a register variable.

**Need an identifier to declare**

In this context, an identifier was expected to complete the declaration. This might be a **typedef** with no name, or an extra semicolon at file level. In C++, it might be a class name improperly used as another kind of identifier.

**'new' and 'delete' not supported**

In integrated debugger expression evaluation, the new and delete operators are not supported.

**No : following the ?**

The question mark (**?**) and colon (**:**) operators do not match in this expression. The colon may have been omitted, or parentheses may be improperly nested or missing.

**No base class to initialize**

This C++ class constructor is trying to implicitly call a base class constructor, but this class was declared with no base classes. Check your declarations.

**No body defined for this inline function**

This C++ function is declared inline, but no function body for it is visible. The inline function body normally is put in the same

header file as the function declaration, whether a member function or an ordinary inline function.

**No constructor parameters allowed for array of class**

When you declare an array of C++ classes, no parameters can be passed to the class constructor. The constructor that takes no parameters (the default constructor) must be used to construct each element of the array.

**No file name ending**

The file name in an **#include** statement was missing the correct closing quote or angle bracket.

**No file names given**

The command line of the Turbo C++ command-line compiler (TCC) contained no file names. You have to specify a source file name.

**No matching )**

A left parenthesis has no matching right parenthesis. Check this expression for balanced parentheses.

**Nonportable pointer conversion**

An implicit conversion between a pointer and an integral type is required, but the types are not the same size. This cannot be done without an explicit cast. This conversion may not make any sense, so be sure this is what you want to

**Non-virtual function *function* declared pure**

Only virtual functions can be declared pure, since derived classes must be able to override them.

**Not an allowed type**

Your source file declared some sort of forbidden type; for example, a function returning a function or array.

**Not a valid expression format type**

Invalid format specifier following expression in the evaluate or watch window. A valid format specifier is an optional repeat value followed by a format character (c, d, f[n], h, x, m, p, r, or s).

**No type information**

Debugger has no type information for this variable. Module may have been compiled without debug switch turned on, or by another compiler or assembler.

### **Numeric constant too large**

String and character escape sequences larger than hexadecimal `\xFF` or octal `\377` cannot be generated. Two-byte character constants may be specified by using a second backslash. For example, `\x0D\x0A` represents a two-byte constant. A numeric literal following an escape sequence should be broken up like this:

```
printf("\x0D" "12345");
```

This prints a carriage return followed by 12345.

### **Object must be initialized**

This C++ object is declared **const**, but is not initialized. Since no value may be assigned to it, it must be initialized at the point of declaration.

### **Only one of a set of overloaded functions can be *function***

C++ functions are by default overloaded, and the compiler assigns a new name to each function. If you wish to override the compiler's assigning a new name by declaring the function *function*, you can do this for only one of a set of functions with the same name. (Otherwise the linker would find more than one global function with the same name.)

### **Operand expected**

In evaluating the current expression, the compiler ran out of operands before using up all the operators. Look for extra operator symbols (**+**, **\***, **/**, and so forth) or missing variable names.

### **Operands are of differing or incompatible type**

The left and right side of a binary operator (**+**, **/**, **==**, and so forth), cannot be combined this way.

### **Operator [ ] missing ]**

The C++ **operator [ ]** was declared as **operator [**. You must add the missing **]** or otherwise fix the declaration.

### **operator -> must return a pointer or a class**

The C++ **operator->** function must be declared to either return a a class or a pointer to a **class** (or **struct** or **union**). In either case, it must be something to which the **->** operator can be applied.

### **Operator cannot be applied to these operand types**

The left or right side of a binary operator (**+**, **-**, **==**, and so forth) is not a valid type for the operator; for example, you may be trying to add two arrays.

**operator delete must have a single parameter of type void \***

This C++ overloaded **operator delete** was declared in some other way.

**operator delete must return void**

This C++ overloaded **operator delete** was declared in some other way.

**operator new must have an initial parameter of type size\_t**

Operator **new** can be declared with an arbitrary number of parameters, but it must always have at least one, which is the amount of space to allocate.

**operator new must have a single parameter of type size\_t**

This C++ overloaded **operator new** was declared in some other way.

**operator new must return an object of type void \***

This C++ overloaded **operator new** was declared in some other way.

**Other objects cannot be declared in a function definition**

A function body cannot be followed by a comma to add other declarations to a list.

For example,

```
int f(), j;           /* f declared, comma OK, j declared */
int f(){return 0;}, j; /* f defined here, comma illegal */
```

**Overlays only supported in medium, large, and huge memory models**

As explained in Chapter 4, only programs using the medium, large, or huge memory models may be overlaid.

**Overloadable operator expected**

Almost all C++ operators can be overloaded. The only ones that can't be overloaded are the field-selection dot (.), dot-star (.\*), double colon (::), and conditional expression (? :). The preprocessor operators (# and ##) are not C or C++ language operators and thus cannot be overloaded. Other nonoperator punctuation, such as semicolon (;), of course, cannot be overloaded.

**Overloaded function is not allowed here**

When you change the access protection of a member of a C++ base class in a derived class, that member cannot be an overloaded function.

### **Overloaded function resolution not supported**

In integrated debugger expression evaluation, resolution of overloaded functions or operators is not supported, not even to take an address.

### **Parameter *parameter* missing name**

In a function definition header, this parameter consisted only of a type specifier with no parameter name. This is not legal in C. (It is allowed in C++, but there's no way to refer to the parameter in the function.)

### **Parameter names are used only with a function body**

When declaring a function (not defining it with a function body), you must use either empty parentheses or a function prototype. A list of parameter names only is not allowed.

Example declarations include:

```
int func();           /* declaration without prototype--OK */
int func(int, int);  /* declaration with prototype--OK */
int func(int i, int j); /* parameter names in prototype--OK */
int func(i, j);      /* parameter names only--illegal */
```

### **Pointer required on left side of ->**

Nothing but a pointer is allowed on the left side of the arrow (->).

### **Pointer to a static member cannot be created**

C++ class member pointers may be created only for ordinary data and function members. You may not create a member pointer to a static member.

### **Previously specified default argument value cannot be changed**

When a parameter of a C++ function is declared to have a default value, this value cannot be changed or omitted in any other declaration for the same function.

### **Pure function *function* not overridden in *class***

A pure virtual function must be either overridden (given a new declaration) or re-declared pure in a derived class.

### **Reference member *member* is not initialized**

References must always be initialized. A class member of reference type must have an initializer provided in all constructors for that class. This means that you cannot depend on the compiler to generate constructors for such a class, since it has no way of knowing how to initialize the references.

**Reference member *member* needs a temporary for initialization**

The user provided an initial value for a reference type which was not an lvalue of the referenced type. This requires the compiler to create a temporary for the initialization. Since there is no obvious place to store this temporary, the initialization is illegal.

**register is the only storage class allowed**

The only storage class allowed for function parameters is **register**.

**Repeat count needs an lvalue**

The expression before the comma (,) in the Watch or Evaluate window must be a manipulable region of storage. For example, expressions like this one are not valid:

```
i++, 10d  
x = y, 10m
```

**Right side of .\* is not a member pointer**

The right side of a C++ dot-star (.\* operator must be declared as a pointer to a member of the class specified by the left side of the operator. In this case, the right side is not a member pointer.

**Side effects are not allowed**

Side effects such as assignments, ++, or -- are not allowed in the Watch window. A common error is to use  $x = y$  (not allowed) instead of  $x == y$  to test the equality of  $x$  and  $y$ .

**Size of *identifier* is unknown or zero**

This identifier was used in a context where its size was needed. A **struct** tag may only be declared (the **struct** not defined yet), or an **extern** array may be declared without a size. It's illegal then to have some references to such an item (like **sizeof**) or to dereference a pointer to this type. Rearrange your declaration so that the size of *identifier* is available.

**sizeof may not be applied to a bit field**

**sizeof** returns the size of a data object in bytes, which does not apply to a bit field.

**sizeof may not be applied to a function**

**sizeof** may be applied only to data objects, not functions. You may request the size of a pointer to a function.

**Size of the type is unknown or zero**

This type was used in a context where its size was needed. For example, a **struct** tag may only be declared (the **struct** not de-

defined yet). It's illegal then to have some references to such an item (like **sizeof**) or to dereference a pointer to this type. Rearrange your declarations so that the size of this type is available.

**Size of this expression is unknown or zero**

This expression involved a type or variable whose size is not known, and is used in a context where the size is needed. A **struct** tag may only be declared (the **struct** not defined yet) or an **extern** array may be declared without a size. It's illegal then to have some references to such an item (like **sizeof**) or to dereference a pointer to this type. Rearrange your declarations so that the needed size is available.

**Statement is required here**

Some parts of C and C++ programs require a statement (even just a semicolon); it's placed between a label and the end of the block it occurs in, and after an **if**, **do**, **while**, or **for** clause.

**Statement missing ;**

The compiler encountered an expression statement without a semicolon following it.

**Static and union members cannot require initialization**

A C++ class that has a constructor or any virtual functions, or is derived from a class that does, must be initialized. A static data member of a class cannot be initialized, and so may not be of a type that requires initialization.

**Storage class *storage class* not allowed for a field**

In C, no storage class is allowed in a field declaration. In C++, a field may be a **typedef**, a data field may be **static**, and a function field may be **inline**. Nothing else is allowed.

**Storage class *storage class* not allowed for a function**

In C and C++ a function may be **extern** or **static**. In C++, a function may be **inline**. Nothing else is allowed, and only one storage class may be present.

**Storage class *storage class* is not allowed here**

The given storage class is not allowed here. Probably two storage classes were specified, and only one may be given.

**Structure size too large**

Your source file declared a structure larger than 64K.

**Subscripting missing ]**

The compiler encountered a subscripting expression which was missing its closing bracket. This could be caused by a missing or extra operator, or mismatched parentheses.

**Switch selection expression must be of integral type**

The selection expression in parentheses in a **switch** statement must evaluate to an integral type (**char, short, int, long, enum**). You may be able to use an explicit cast to satisfy this requirement.

**Switch statement missing (**

In a **switch** statement, the compiler found no left parenthesis after the **switch** keyword.

**Switch statement missing )**

In a **switch** statement, the compiler found no right parenthesis after the test expression.

**'this' can only be used within a member function**

In C++, **this** is a reserved word that can be used only within class member functions.

**Too few parameters in call**

A call to a function with a prototype (via a function pointer) had too few arguments. Prototypes require that all parameters be given.

**Too few parameters in call to function**

A call to the named function (declared using a prototype) had too few arguments.

**Too many decimal points**

The compiler encountered a floating-point constant with more than one decimal point.

**Too many default cases**

The compiler encountered more than one **default** statement in a single **switch**.

**Too many error or warning messages**

A maximum of 255 errors and warnings can be set before the compiler stops.

**Too many exponents**

The compiler encountered more than one exponent in a floating-point constant.

**Too many initializers**

The compiler encountered more initializers than were allowed by the declaration being initialized.

**Too many storage classes in declaration**

A declaration may never have more than one storage class.

**Too many types in declaration**

A declaration may never have more than one of the basic types: **char**, **int**, **float**, **double**, **struct**, **union**, **enum**, or **typedef-name**.

**Too much global data defined in file**

The sum of the global data declarations exceeds 64K bytes.

Check the declarations for any array that may be too large.

Also consider reorganizing the program or using **far** variables if all the declarations are needed.

**Trying to derive a far class from a near base**

If a class is declared (or defaults to) **near**, all derived classes must also be **near**.

**Trying to derive a near class from a far base**

If a class is declared (or defaults to) **far**, all derived classes must also be **far**.

**Two consecutive dots**

Because an ellipsis contains three dots (...), and a decimal point or member selection operator uses one dot (.), there is no way two consecutive dots can legally occur in a C program.

**Two operands must evaluate to the same type**

The types of the expressions on both sides of the colon in the conditional expression operator (?:) must be the same, except for the usual conversions like **char** to **int** or **float** to **double**, or **void\*** to a particular pointer. In this expression, the two sides evaluate to different types that are not automatically converted. This may be an error or you may merely need to cast one side to the type of the other.

**Type mismatch in parameter *number***

The function called, via a function pointer, was declared with a prototype; the given parameter *number* (counting left to right from 1) could not be converted to the declared parameter type.

**Type mismatch in parameter *number* in call to *function***

Your source file declared the named function with a prototype, and the given parameter *number* (counting left to right from 1) could not be converted to the declared parameter type.

**Type mismatch in parameter *parameter***

Your source file declared the function called via a function pointer with a prototype, and the named parameter could not be converted to the declared parameter type.

**Type mismatch in parameter *parameter* in call to function**

Your source file declared the named function with a prototype, and the named parameter could not be converted to the declared parameter type.

**Type mismatch in redeclaration of *identifier***

Your source file redeclared a variable with a different type than was originally declared for the variable. This can occur if a function is called and subsequently declared to return something other than an integer. If this has happened, you must declare the function before the first call to it.

**Type name expected**

One of these errors has occurred:

- In declaring a file-level variable or a **struct** field, neither a type name nor a storage class was given.
- In declaring a **typedef**, no type for the name was supplied.
- In declaring a destructor for a C++ class, the destructor name was not a type name (it must be the same name as its class).
- In supplying a C++ base class name, the name was not the name of a class.

**Type qualifier *identifier* must be a struct or class name**

The C++ qualifier in the construction *qual::identifier* is not the name of a **struct** or **class**.

**Unable to create output file *filename***

This error occurs if the work disk is full or write protected. If the disk is full, try deleting unneeded files and restarting the compilation. If the disk is write-protected, move the source files to a writable disk and restart the compilation. This error also occurs if the output directory does not exist.

**Unable to create `turboc.$ln`**

The compiler cannot create the temporary file TURBOC.\$LN because it cannot access the disk or the disk is full.

**Unable to execute command *command***

TLINK or TASM cannot be found, or possibly the disk is bad.

**Unable to open include file *filename***

The compiler could not find the named file. This could also be caused if an **#include** file included itself, or if you do not have **FILES** set in **CONFIG.SYS** on your root directory (try **FILES=20**). Check whether the named file exists.

**Unable to open input file *filename***

This error occurs if the source file cannot be found. Check the spelling of the name and whether the file is on the proper disk or directory.

**Undefined label *identifier***

The named label has a **goto** in the function, but no label definition.

**Undefined structure *structure***

Your source file used the named structure on some line before where the error is indicated (probably on a pointer to a structure) but had no definition for the structure. This is probably caused by a misspelled structure name or a missing declaration.

**Undefined symbol *identifier***

The named identifier has no declaration. This could be caused by a misspelling either at this point or at the declaration. This could also be caused if there was an error in the declaration of the identifier.

**Unexpected }**

An extra right brace was encountered where none was expected. Check for a missing {.

**Unexpected )—check for matching parenthesis**

An extra right parenthesis was encountered where none was expected. Check for a missing (.

**Unexpected : found**

An extra colon was encountered where none was expected. Check for a missing or misplaced ?.

**Unexpected end of file in comment started on *line number***

The source file ended in the middle of a comment. This is normally caused by a missing close of comment (\*/\*).

**Unexpected end of file in conditional started on *line number***

The source file ended before the compiler encountered **#endif**. The **#endif** either was missing or misspelled.

**union cannot have a base type**

In general, a C++ class may be of **union** type, but such a class cannot be derived from any other class.

**Union members cannot require initialization**

Since the lifetime of a union member is indeterminate, it is not legal to declare one which requires initialization of some sort.

**Unknown language, must be C or C++**

In the C++ construction

```
extern name type func( /*...*/ );
```

The given name in quotes must be C or C++; other language names are not recognized. You can declare an external Pascal function without the compiler's renaming like this:

```
extern "C" int pascal func( /*...*/ );
```

A C++ (possibly overloaded) function may be declared Pascal and allow the usual compiler renaming (to allow overloading) like this:

```
extern int pascal func( /*...*/ );
```

**Unknown preprocessor directive: *identifier***

The compiler encountered a # character at the beginning of a line, and the directive name following was not one of these: **define, undef, line, if, ifdef, ifndef, include, else, or endif.**

**Unterminated string or character constant**

The compiler found no terminating quote after the beginning of a string or character constant.

**Use . or -> to call *function***

You attempted to call a member function without providing an object.

**Use :: to take the address of a member function**

If *f* is a member function of class *c*, you take its address with the syntax *&c::f*. Note the use of the class type name, not the name of an object, and the :: separating the class name from the function name. (Member function pointers are not true pointer types, and do not refer to any particular instance of a class.)

**Use ; to terminate declarations**

This declaration has not been terminated with a comma or a semicolon.

**User break**

You typed a *Ctrl-Break* while compiling or linking in the integrated environment. (This is not an error, just a confirmation.)

**Value of type void is not allowed**

A value of type **void** is really not a value at all, and thus may not appear in any context where an actual value is required. Such contexts include the right side of an assignment, an argument of a function, and the controlling expression of an **if**, **for**, or **while** statement.

**Variable *identifier* is initialized twice**

This variable has more than one initialization. It is legal to declare a file level variable more than once, but it may have only one initialization (even if two are the same).

**Variable name expected**

When using the address-of operator (**&**), or when in C++ returning a reference to an object, an actual object must be supplied. This is typically the name of a variable. In this case, the compiler is being asked to take the address of something inappropriate.

**Vectors of classes must use the default constructor**

When initializing a vector (array) of classes, you must use the constructor that has no arguments. This is called the *default constructor*, which means that you may not supply constructor arguments when initializing such a vector.

**Virtual function *function1* conflicts with *function2***

A virtual function has the same argument types as one in a base class, but a different return type. This is illegal.

**virtual specified more than once**

The C++ reserved word **virtual** may appear only once in one member function declaration.

**void & is not a valid type**

An explicit message for an obvious restriction. This was always caught before when you tried to initialize or use the reference type. This is clearer.

**While statement missing (**

In a **while** statement, the compiler found no left parenthesis after the **while** keyword.

**While statement missing )**

In a **while** statement, the compiler found no right parenthesis after the test expression.

**Wrong number of arguments in call of macro**

Your source file called the named macro with an incorrect number of arguments.

## Warnings

---

***function1* hides virtual function *function2***

A virtual function in a base class is usually overridden by a declaration in a derived class. In this case, a declaration with the same name but different argument types makes the virtual functions inaccessible to further derived classes.

***identifier* is declared as both external and static**

This identifier appeared in a declaration that implicitly or explicitly marked it as global or external, and also in a static declaration. The identifier is taken as static. You should review all declarations for this identifier.

***identifier* declared but never used**

Your source file declared the named variable as part of the block just ending, but the variable was never used. The warning is indicated when the compiler encounters the closing brace of the compound statement or function. The declaration of the variable occurs at the beginning of the compound statement or function.

***identifier* is assigned a value that is never used**

The variable appears in an assignment, but is never used anywhere else in the function just ending. The warning is indicated only when the compiler encounters the closing brace.

***identifier* is both a structure tag and a name, now obsolete**

In C, it is perfectly valid to use an identifier as both the tag of a structure and as a variable or typedef name, as in

```
struct s { int i, j; } s;
```

or

```
typedef struct s { int i, j; } s;
```

This is not appropriate in C++.

### **Ambiguous operators need parentheses**

This warning is displayed whenever two shift, relational or bitwise-Boolean operators are used together without parentheses. Also, an addition or subtraction operator that appears unparenthesized with a shift operator will produce this warning. Programmers frequently confuse the precedence of these operators, since the precedence assigned to them is somewhat counter-intuitive.

### **Assigning type to enumeration**

Assigning an integer value to an **enum** type. This is an error, but is reduced to a warning to give existing programs a chance to work.

### **Assignment to this is obsolete, use X::operator new instead**

In early versions of C++, the only way to control allocation of class of objects was by assigning to the **this** parameter inside a constructor. This practice is now discouraged, since a better, safer, and more general technique is to define a member function **operator new** instead.

### **Base initialization without a class name is now obsolete**

Early versions of C++ provided for initialization of a base class by following the constructor header with just the base class constructor parameter list. It is now recommended to include the base class name.

This makes the code much clearer, and is required when there are multiple base classes.

Old way:

```
derived::derived(int i) : (i, 10) { ... }
```

New way:

```
derived::derived(int i) : base(i, 10) { ... }
```

### **Bit fields must be signed or unsigned int**

A bit field must be declared to be a signed or unsigned integral type. In ANSI C, bit fields may only be signed or unsigned **int** (not **char** or **long**, for example).

### **Both return and return with a value used**

The current function has **return** statements with and without values. This is legal C, but almost always an error. Possibly a **return** statement was omitted from the end of the function.

### Call to function with no prototype

This message is given if the “Prototypes required” warning is enabled and you call a function without first giving a prototype for that function.

### Call to function *function* with no prototype

This message is given if the “Prototypes required” warning is enabled and you call function *function* without first giving a prototype for that function.

### Code has no effect

This warning is issued when the compiler encounters a statement with some operators which have no effect. For example the statement

```
a + b;
```

has no effect on either variable. The operation is unnecessary and probably indicates a bug.

### Constant is long

The compiler encountered either a decimal constant greater than 32767 or an octal (or hexadecimal) constant greater than 65535 without a letter *l* or *L* following it. The constant is treated as a **long**.

### Constant member *member* is not initialized

This C++ class contains a constant member *member*, which does not have an initialization. Note that constant members may be initialized only, not assigned to.

### Constant out of range in comparison

Your source file includes a comparison involving a constant sub-expression that was outside the range allowed by the other sub-expression's type. For example, comparing an **unsigned** quantity to  $-1$  makes no sense. To get an **unsigned** constant greater than 32767 (in decimal), you should either cast the constant to **unsigned** [for example, **(unsigned)65535**] or append a letter *u* or *U* to the constant (for example, 65535u).

Whenever this message is issued, the compiler will still generate code to do the comparison. If this code ends up always giving the same result, such as comparing a **char** expression to 4000, the code will still perform the test.

### Conversion may lose significant digits

For an assignment operator or some other circumstance, your source file requires a conversion from **long** or **unsigned long** to

**int** or **unsigned int** type. Since **int** type and **long** type variables don't have the same size, this kind of conversion may alter the behavior of a program.

#### **Declaration does not specify a tag or an identifier**

This declaration doesn't declare anything. This is usually a **struct** or a **union** without a tag or a variable in the declaration.

Some early C compilers allowed a declaration like this:

```
struct { int a; int b; };
```

and further allowed *a* and *b* to be used as generic fields for any variable: An expression like *x.b* was allowed even if *x* was not a structure type. This practice is now discouraged, and this warning can help you find such abuses.

#### **Declare function prior to use in prototype**

When a function prototype refers to a structure type which has not previously been declared, the declaration inside the prototype is not the same as a declaration outside the prototype. For example:

```
int func(struct s *ps);
struct s { /* ... */};
```

Since there is no `struct s` in scope at the prototype for **func**, the type of parameter *ps* is pointer to undefined `struct s`, and is not the same as the `struct s` which is later declared. This will result in later warning and error messages about incompatible types, which would be very mysterious without this warning message. To fix the problem, you can move the declaration for `struct s` ahead of any prototype which references it, or add the incomplete type declaration `struct s;` ahead of any prototype which references `struct s`. If the function parameter is a **struct**, rather than a pointer to **struct**, the incomplete declaration is not sufficient; you must then place the `struct` declaration ahead of the prototype.

#### **Division by zero**

A divide or remainder expression had a literal zero as a divisor.

#### **Functions containing reserved word are not expanded inline**

Functions containing any of the reserved words **do**, **for**, **while**, **goto**, **switch**, **break**, **continue**, and **case** cannot be expanded inline, even when specified as **inline**. The function is still perfectly legal, but will be treated as an ordinary static (not global)

function. A copy of the function will appear in each compilation unit where it is called.

### **Function should return a value**

Your source file declared the current function to return some type other than **int** or **void**, but the compiler encountered a return with no value. This is usually some sort of error. **int** functions are exempt, since in old versions of C there was no **void** type to indicate functions which return nothing.

### **Hexadecimal value contains more than 3 digits**

Under older versions of C, a hexadecimal escape sequence could contain no more than three digits. The new ANSI standard allows any number of digits to appear as long as the value fits in a byte. This warning results when you have a long hexadecimal escape sequence with many leading zero digits (such as “\x00045”). Older versions of C would interpret such a string differently.

### **Ill-formed pragma**

A pragma does not match one of the pragmas expected by the Turbo C++ compiler.

### **Initialization is only partially bracketed**

When structures are initialized, braces can be used to mark the initialization of each member of the structure. If a member itself is an array or structure, nested pairs of braces may be used. This ensures that your idea and the compiler's idea of what value goes with which member are the same. When some of the optional braces are omitted, the compiler issues this warning.

### **Initialization with inappropriate type**

A variable of type **enum** is being initialized with a value of a different type. For example,

```
enum count { zero, one, two } x = 2;
```

will result in this warning, because 2 is of type **int**, not type **enum count**. It is better programming practice to use an **enum** identifier instead of a literal integer when assigning to or initializing **enum** types.

### **Initializing *identifier* with *type***

You're trying to initialize an **enum** variable to a different type. This is an error, but is reduced to a warning to give existing programs a chance to work.

### **Mixing pointers to signed and unsigned char**

You converted a **char** pointer to an **unsigned char** pointer, or vice versa, without using an explicit cast. (Strictly speaking, this is incorrect, but on the 8086, it is often harmless.)

### **No declaration for function *function***

This message is given if you call a function without first declaring that function. In C, you can declare a function without presenting a prototype, as in “int func()”;. In C++, every function declaration is also a prototype; this example is equivalent to “int func(void);”. The declaration can be either classic or modern (prototype) style.

### **Non-const function *function* called for const object**

A non-**const** member function was called for a **const** object. This is an error, but was reduced to a warning to give existing programs a chance to work.

### **Nonportable pointer comparison**

Your source file compared a pointer to a non-pointer other than the constant zero. You should use a cast to suppress this warning if the comparison is proper.

### **Nonportable pointer conversion**

A nonzero integral value is used in a context where a pointer is needed or where an integral value is needed; the sizes of the integral type and pointer are the same. Use an explicit cast if this is what you really meant to do.

### **Obsolete syntax; use :: instead**

Early versions of C++ used a dot (.) or a colon (:) to separate a member name from its class name in a declaration or definition. This is now obsolete; the double colon (::) must be used.

Old way:

```
void myclass.func(int i) { /* ... */ }
```

New way:

```
void myclass::func(int i) { /* ... */ }
```

### **overload is now unnecessary and obsolete**

Early versions of C++ required the reserved word **overload** to mark overloaded function names. C++ now uses a “type-safe linkage” scheme, whereby all functions are assumed overloaded unless marked otherwise. The use of **overload** should be discontinued.

**Parameter *parameter* is never used**

The named parameter, declared in the function, was never used in the body of the function. This may or may not be an error and is often caused by misspelling the parameter. This warning can also occur if the identifier is redeclared as an automatic (local) variable in the body of the function. The parameter is masked by the automatic variable and remains unused.

**Possible use of *identifier* before definition**

Your source file used the named variable in an expression before it was assigned a value. The compiler uses a simple scan of the program to determine this condition. If the use of a variable occurs physically before any assignment, this warning will be generated. Of course, the actual flow of the program may assign the value before the program uses it.

**Possibly incorrect assignment**

This warning is generated when the compiler encounters an assignment operator as the main operator of a conditional expression (that is, part of an **if**, **while** or **do-while** statement). More often than not, this is a typographical error for the equality operator. If you wish to suppress this warning, enclose the assignment in parentheses and compare the whole thing to zero explicitly. Thus,

```
if (a = b) ...
```

should be rewritten as

```
if ((a = b) != 0) ...
```

**Program flow can skip this initialization; try using { }**

This variable's initialization is controlled by an **if** statement, and thus might be skipped. You probably need a pair of braces around a block to control the scope of this variable.

**Redefinition of *macro* is not identical**

Your source file redefined the named macro using text that was not exactly the same as the first definition of the macro. The new text replaces the old.

**Restarting compile using assembly**

The compiler encountered an **asm** with no accompanying **-B** command line option or **#pragma inline** statement. The compile restarts using assembly language capabilities.

### Structure passed by value

If “Structure passed by value” warning is enabled, this warning is generated anytime a structure is passed by value as an argument. It is a frequent programming mistake to leave an address-of operator (&) off a structure when passing it as an argument. Because structures can be passed by value, this omission is acceptable. This warning provides a way for the compiler to warn you of this mistake.

### Style of function definition is now obsolete

In C++, this old C style of function definition is illegal:

```
int func(p1, p2) int p1, p2; { /* ... */ }
```

This practice may not be allowed by other C++ compilers.

### Superfluous & with function

An address-of operator (&) is not needed with function name; any such operators are discarded.

### Suspicious pointer conversion

The compiler encountered some conversion of a pointer which caused the pointer to point to a different type. You should use a cast to suppress this warning if the conversion is proper.

### Temporary used to initialize *identifier*

### Temporary used for parameter *number* in call to *identifier*

### Temporary used for parameter *parameter* in call to *identifier*

### Temporary used for parameter *number*

### Temporary used for parameter *parameter*

In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type. If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter. The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.

For example, here function **f** requires a reference to an **int**, and **c** is a **char**:

```
f(int&);  
char c;  
f(c);
```

Instead of calling **f** with the address of **c**, the compiler generates code equivalent to the C++ source code:

```
int X = c, f(X);
```

**Undefined structure identifier**

The named structure was used in the source file, probably on a pointer to a structure, but had no definition in the source file. This is probably caused by a misspelled structure name or a missing declaration.

**Unknown assembler instruction**

The compiler encountered an inline assembly statement with a disallowed opcode. Check the spelling of the opcode (in Chapter 6, "Interfacing with assembly language," page 271). This warning is off by default.

**Unreachable code**

A **break**, **continue**, **goto** or **return** statement was not followed by a label or the end of a loop or function. The compiler checks **while**, **do** and **for** loops with a constant test condition, and attempts to recognize loops which cannot fall through.

**Untyped bit field assumed signed int**

This bit field had no type specification, and is taken to be signed **int**. Some compilers default to unsigned **int**. You should supply a declaration of either **int** or unsigned **int**.

**Void functions may not return a value**

Your source file declared the current function as returning **void**, but the compiler encountered a return statement with a value. The value of the return statement will be ignored.

## ANSI implementation-specific standards

Certain aspects of the ANSI C standard are not defined exactly by ANSI. Instead, each implementor of a C compiler is free to define these aspects individually. This chapter tells how Borland has chosen to define these implementation-specific standards. The section numbers refer to the ANSI Standard December 1988 draft, which is the most recent working draft. Remember that there are differences between C and C++; this appendix addresses C only.

### 2.1.1.3 How to identify a diagnostic.

When run with the correct combination of options, any message issued by the compiler beginning with the words *Fatal*, *Error*, or *Warning* are diagnostics in the sense that ANSI specifies. The options needed to insure this interpretation are as follows:

Table A.1  
Identifying diagnostics in  
Turbo C++

Option	Action
-A	Enable only ANSI keywords.
-C-	No nested comments allowed.
-p-	Use C calling conventions.
-i32	At least 32 significant characters in identifiers.
-w-	Turn off all warnings except the following.
-wbei	Turn on warning about inappropriate initializers.
-wdcl	Turn on warning about declarations without type or storage class.
-wcpt	Turn on warning about non-portable pointer comparisons.
-wdup	Turn on warning about duplicate non-identical macro definitions.
-wsus	Turn on warning about suspicious pointer conversion.
-wrpt	Turn on warning about non-portable pointer conversion.

Table A.1: Identifying diagnostics in Turbo C++ (continued)

-wvrt	Turn on warning about void functions returning a value.
-wbig	Turn on warning about constants being too large.
-wucp	Turn on warning about mixing pointers to signed and unsigned char.
-wstu	Turn on warning about undefined structures.
-wext	Turn on warning about variables declared both as external and as static.
-wfdt	Turn on warning about function definitions using a typedef.

None of the following options can be used:

-ms!	SS must be the same as DS for small data models.
-mm!	SS must be the same as DS for small data models.
-mt!	SS must be the same as DS for small data models.
-zGxx	The BSS group name may not be changed.
-zSxx	The data group name may not be changed.

Other options not specifically mentioned here can be set to whatever you desire.

### 2.1.2.2 The semantics of the arguments to main.

The value of *argv*[0] is a pointer to a null byte when the program is run on DOS versions prior to version 3.0. For DOS version 3.0 or later, *argv*[0] points to the program name.

The remaining *argv* strings point to each component of the DOS command-line arguments. Whitespace separating arguments is removed, and each sequence of contiguous nonwhitespace characters is treated as a single argument. Quoted strings are handled correctly (that is, as one string containing spaces).

### 2.1.2.3 What constitutes an interactive device.

Any device that looks like the console.

### 2.2.1 Members of the source and execution character sets.

The source and execution character sets are the extended ASCII set supported by the IBM PC. Any character other than ^Z (Control-Z) can appear in string literals, character constants, or comments.

### 2.2.1.2 Shift states for multibyte characters.

No multibyte characters are supported in Turbo C++.

**2.2.2 The direction of printing.**

Printing is from left-to-right, the normal direction for the PC.

**2.2.4.2 The number of bits in a character in the execution character set.**

There are 8 bits per character in the execution character set.

**3.1.2 The number of significant initial characters in identifiers.**

The first 32 characters are significant, although you can use a command-line option (-i) to change that number. Both internal and external identifiers use the same number of significant digits. (The number of significant characters in C++ identifiers is unlimited.)

**3.1.2 Whether case distinctions are significant in external identifiers.**

The compiler will normally force the linker to distinguish between uppercase and lowercase. You can use a command-line option (-l-c) to suppress the distinction.

**3.1.2.5 The representations and sets of values of the various types of integers.**

Type	Minimum value	Maximum value
signed char	-128	127
unsigned char	0	255
signed short	-32,768	32,767
unsigned short	0	65,535
signed int	-32,768	32,767
unsigned int	0	65,535
signed long	-2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295

All **char** types use one 8-bit byte for storage.

All **short** and **int** types use 2 bytes.

All **long** types use 4 bytes.

If alignment is requested (-a), all non**char** integer type objects will be aligned to even byte boundaries. Character types are never aligned.

**3.1.2.5 The representations and sets of values of the various types of floating-point numbers.**

The IEEE floating-point formats as used by the Intel 8087 are used for all Turbo C++ floating-point types. The **float** type uses 32-bit IEEE real format. The **double** type uses 64-bit IEEE real format. The **long double** type uses 80-bit IEEE extended real format.

**3.1.3.4 The mapping between source and execution character sets.**

Any characters in string literals or character constants will remain unchanged in the executing program. The source and execution character sets are the same.

**3.1.3.4 The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant.**

Wide characters are not supported. They are treated as normal characters. All legal escape sequences map onto one or another character. If a hex or octal escape sequence is used that exceeds the range of a character, the compiler issues a message.

**3.1.3.4 The value of an integer constant that contains more than one character, or a wide character constant that contains more than one multibyte character.**

Character constants can contain one or two characters. If two characters are included, the first character occupies the low-order byte of the constant, and the second character occupies the high-order byte.

**3.1.3.4 The current locale used to convert multibyte characters into corresponding wide characters for a wide character constant.**

Wide character constants are recognized, but treated in all ways like normal character constants. In that sense, the locale is the "C" locale.

**3.2.1.2 The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented.**

These conversions are performed by simply truncating the high-order bits. Signed integers are stored as 2's-complement values, so the resulting number is interpreted as such a value. If the high-order bit of the smaller integer is nonzero, the value is interpreted as a negative value; otherwise, it is positive.

**3.2.1.3 The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value.**

The integer value is rounded to the nearest representable value. Thus, for example, the **long** value ( $2^{31-1}$ ) is converted to the **float** value  $2^{31}$ . Ties are broken according to the rules of IEEE standard arithmetic.

**3.2.1.4 The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number.**

The value is rounded to the nearest representable value. Ties are broken according to the rules of IEEE standard arithmetic.

**3.3 The results of bitwise operations on signed integers.**

The bitwise operators apply to signed integers as if they were their corresponding unsigned types. The sign bit is treated as a normal data bit. The result is then interpreted as a normal 2's complement signed integer.

**3.3.2.3 What happens when a member of a union object is accessed using a member of a different type.**

The access is allowed and will simply access the bits stored there. You'll need a detailed understanding of the bit encodings of floating-point values in order to understand how to access a floating-type member using a different member. If the member stored is shorter than the member used to access the value, the excess bits have the value they had before the short member was stored.

**3.3.3.4 The type of integer required to hold the maximum size of an array.**

For a normal array, the type is **unsigned int**, and for huge arrays the type is **signed long**.

- 3.3.4 The result of casting a pointer to an integer or vice versa.** When converting between integers and pointers of the same size, no bits are changed. When converting from a longer type to a shorter, the high-order bits are truncated. When converting from a shorter integer type to a longer pointer type, the integer is first widened to an integer type that is the same size as the pointer type. Thus signed integers will sign-extend to fill the new bytes. Similarly, smaller pointer types being converted to larger integer types will first be widened to a pointer type that is as wide as the integer type.
- 3.3.5 The sign of the remainder on integer division.** The sign of the remainder is negative when only one of the operands is negative. If neither or both operands are negative, the remainder is positive.
- 3.3.6 The type of integer required to hold the difference between two pointers to elements of the same array, ptrdiff\_t.** The type is **signed int** when the pointers are near, or **signed long** when the pointers are far or huge. The type of *ptrdiff\_t* depends on the memory model in use. In small data models, the type is **int**. In large data models, the type is **long**.
- 3.3.7 The result of a right shift of a negative signed integral type.** A negative signed value is sign-extended when right shifted.
- 3.5.1 The extent to which objects can actually be placed in registers by using the register storage-class specifier.** Objects declared with any two-byte integer or pointer types can be placed in registers. The compiler will place any small auto objects into registers, but objects explicitly declared as *register* will take precedence. At least two and as many as six registers are available. The number of registers actually used depends on what registers are needed for temporary values in the function.
- 3.5.2.1 The padding and alignment of members of structures.** By default, no padding is used in structures. If you use the alignment option (**-a**), structures are padded to even size, and any members that do not have character or character array type will be aligned to an even offset.
- 3.5.2.1 Whether a plain int bit-field is treated as a signed int or as an unsigned int bit field.** Plain **int** bit fields are treated as **signed int** bit fields.

**3.5.2.1 The order of allocation of bit fields within an int.**

Bit fields are allocated from the low-order bit position to the high-order.

**3.5.2.1 Whether a bit-field can straddle a storage-unit boundary.**

When alignment (**-a**) is not requested, bit fields can straddle word boundaries, but will never be stored in more than two adjacent bytes.

**3.5.2.2 The integer type chosen to represent the values of an enumeration type.**

If all enumerators can fit in an **unsigned char**, that is the type chosen. Otherwise, the type is **signed int**.

**3.5.4 The maximum number of declarators that can modify an arithmetic, structure, or union type.**

There is no specific limit on the number of declarators. The number of declarators allowed is fairly large, but when nested deeply within a set of blocks in a function, the number of declarators will be reduced. The number allowed at file level is at least 50.

**3.5.5.3 What constitutes an access to an object that has volatile-qualified type.**

Any reference to a volatile object will access the object. Whether accessing adjacent memory locations will also access an object depends on how the memory is constructed in the hardware. For special device memory, like video display memory, it depends on how the device is constructed. For normal PC memory, volatile objects are only used for memory that might be accessed by asynchronous interrupts, so accessing adjacent objects has no effect.

**3.6.4.2 The maximum number of case values in a switch statement.**

There is no specific limit on the number of cases in a switch. As long as there is enough memory to hold the case information, the compiler will accept them.

**3.8.1 Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant can have a negative value.**

All character constants, even constants in conditional directives use the same character set (execution). Single-character character constants will be negative if the character type is signed (default and **-K** not requested).

**3.8.2 The method for locating includable source files.**

For include file names given with angle brackets, if include directories are given in the command line, then the file is searched for in each of the include directories. Include directories are searched in this order: First, using directories specified on the command line, then using directories specified in **TURBOC.CFG**. If no include directories are specified, then only the current directory is searched.

**3.8.2 The support for quoted names for includable source files.**

For quoted file names, the file is first searched for in the current directory. If not found, Turbo C++ searches for the file as if it were in angle brackets.

**3.8.2 The mapping of source file name character sequences.**

Backslashes in include file names are treated as distinct characters, not as escape characters. Case differences are ignored for letters.

**3.8.8 The definitions for `__DATE__` and `__TIME__` when they are unavailable.**

The date and time are always available, and will use the DOS date and time.

**4.1.1 The type of the `sizeof` operator, `size_t`.**

The type `size_t` is **unsigned int**.

**4.1.1 The decimal point character.**

It is a period (`.`).

**4.1.5 The null pointer constant to which the macro NULL expands.**

An integer or a long 0, depending upon the memory model.

**4.2 The diagnostic printed by and the termination behavior or the assert function.**

The diagnostic message printed is “Assertion failed: *expression*, file *filename*, line *nn*”, where *expression* is the asserted expression which failed, *filename* is the source file name, and *nn* is the line number where the assertion took place.

**abort** is called immediately after the assertion message is displayed.

**4.3 The implementation-defined aspects of character testing and case mapping functions.**

None, other than what is mentioned in 4.3.1.

**4.3.1 The sets of characters tested for by the isalnum, isalpha, iscntrl, islower, isprint and isupper functions.**

First 128 ASCII characters.

**4.5.1 The values returned by the mathematics functions on domain errors.**

An IEEE NAN (not a number).

**4.5.1 Whether the mathematics functions set the integer expression *errno* to the value of the macro ERANGE on underflow range errors.**

No, only for the other errors—domain, singularity, overflow, and total loss of precision.

- 4.5.6.4 Whether a domain error occurs or zero is returned when the fmod function has a second argument of zero.** No. `fmod(x, 0)` returns 0.
- 4.7.1.1 The set of signals for the signal function.** SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM.
- 4.7.1.1 The semantics for each signal recognized by the signal function.** See the description of **signal** in the *Library Reference*.
- 4.7.1.1 The default handling and the handling at program startup for each signal recognized by the signal function.** See the description of **signal** in the *Library Reference*.
- 4.7.1.1 If the equivalent of `signal(sig, SIG_DFL)`; is not executed prior to the call of a signal handler, the blocking of the signal that is performed.** The equivalent of **signal** (`sig, SIG_DFL`) is always executed.
- 4.7.1.1 Whether the default handling is reset if the SIGILL signal is received by a handler specified to the signal function.** No.
- 4.9.2 Whether the last line of a text stream requires a terminating newline character.** No, none is required.

- 4.9.2 Whether space characters that are written out to a text stream immediately before a newline character appear when read in.** Yes, they do.
- 4.9.2 The number of null characters that may be appended to data written to a binary stream.** None.
- 4.9.3 Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file.** The file position indicator of an append-mode stream is initially placed at the beginning of the file. It is reset to the end of the file before each write.
- 4.9.3 Whether a write on a text stream causes the associated file to be truncated beyond that point.** A write of 0 bytes *may* or *may not* truncate the file, depending upon how the file is buffered. It is safest to classify a zero-length write as having indeterminate behavior.
- 4.9.3 The characteristics of file buffering.** Files can be fully buffered, line buffered, or unbuffered. If a file is buffered, a default buffer of 512 bytes is created upon opening the file.
- 4.9.3 Whether a zero-length file actually exists.** Yes, it does.
- 4.9.3 Whether the same file can be open multiple times.** Yes, it can.
- 4.9.4.1 The effect of the remove function on an open file.** No special checking for an already open file is performed; the responsibility is left up to the programmer.

**4.9.4.2 The effect if a file with the new name exists prior to a call to `rename`.** `rename` will return a `-1` and `errno` will be set to `EEXIST`.

**4.9.6.1 The output for `%p` conversion in `printf`.** In near data models, four hex digits (`XXXX`). In far data models, four hex digits, colon, four hex digits (`XXXX:XXXX`).

**4.9.6.2 The input for `%p` conversion in `fscanf`.** See 4.9.6.1.

**4.9.6.2 The interpretation of an `-` (hyphen) character that is neither the first nor the last character in the scanlist for a `%[` conversion in `fscanf`.** See the description of `scanf` in the *Library Reference*.

**4.9.9.1 The value to which the macro `errno` is set by the `fgetpos` or `ftell` function on failure.** EBADF Bad file number.

**4.9.10.4 The messages generated by `perror`.**

---

Error 0	Invalid data
Invalid function number	No such device
No such file or directory	Attempted to remove current directory
Path not found	Not same device
Too many open files	No more files
Permission denied	Invalid argument
Bad file number	Arg list too big
Memory arena trashed	Exec format error
Not enough memory	Cross-device link
Invalid memory block address	Math argument
Invalid environment	Result too large
Invalid format	File already exists
Invalid access code	

---

See `perror` in the *Library Reference* for details.

**4.10.3 The behavior of `calloc`, `malloc`, or `realloc` if the size requested is zero.**

`calloc` and `malloc` will ignore the request. `realloc` will free the block.

**4.10.4.1 The behavior of the abort function with regard to open and temporary files.**

The file buffers are not flushed and the files are not closed.

**4.10.4.3 The status returned by `exit` if the value of the argument is other than zero, `EXIT_SUCCESS`, or `EXIT_FAILURE`.**

Nothing special. The status is returned exactly as it is passed. The status is represented as a **signed char**.

**4.10.4.4 The set of environment names and the method for altering the environment list used by `getenv`.**

The environment strings are those defined in DOS with the `SET` command. `putenv` can be used to change the strings for the duration of the current program, but the DOS `SET` command must be used to change an environment string permanently.

**4.10.4.5 The contents and mode of execution of the string by the system function.**

The string is interpreted as a DOS command. `COMMAND.COM` is executed and the argument string is passed as a command to execute. Any DOS built-in command, as well as batch files and executable programs, can be executed.

**4.11.4.4 The collation sequence of the execution character set.**

The collation sequence for the execution character set uses the signed value of the character in ASCII.

**4.11.6.2 The contents of the error message strings returned by `strerror`.**

See 4.9.10.4.

**4.12.1 The local time zone and Daylight Saving Time.**

Defined as local PC time and date.

**4.12.2.1 The era for clock.** Represented as clock ticks, with the origin being the beginning of the program execution.

**4.12.3.5 The formats for date and time.** Turbo C++ implements ANSI formats.

- ;
- null statement 24, 93
- statement terminator 24, 93
- /\* \*/ (comments) 6
- /\*\*/ (token pasting) 6
- // (comments) 7
- operator
  - decrement 21, 79
- ? : operator
  - conditional expression 23, 89
- :: (scope resolution operator) 23, 100
- \* and ->\* (dereference pointers) 23
- 1 command-line compiler option 271
- 87 environment variable 206
- \\ escape sequence (display backslash character) 15
- \" escape sequence (display double quote) 15
- \? escape sequence (display question mark) 15
- \' escape sequence (display single quote) 15
- : (labeled statement) 25
- != operator
  - huge pointer comparison and 193
  - not equal to 23, 87
- && operator
  - logical AND 22, 88
- ++ operator
  - increment 21, 78, 79
- << operator
  - put to *See* overloaded operators, >> (put to)
  - shift bits left 22, 84
- <= operator
  - less than or equal to 22, 85
- == operator
  - equal to 86
  - huge pointer comparison and 193
- >= operator
  - greater than or equal to 22, 86
- >> operator
  - character conversions and 176
  - floating-point conversion and 176
  - get from *See* overloaded operators, << (get from)
  - integer conversion and 176
  - shift bits right 22, 84
- || operator
  - logical OR 22, 88
- > operator (selection) 23
- overloading 128
- structure member access 66, 78
- union member access 78
- \* (pointer declarator) 25
- \ (string continuation character) 18
- ## symbol
  - overloading and 124
  - preprocessor directives 21
  - token pasting 6, 139
- ! operator
  - logical negation 21, 81
- % operator
  - modulus 22, 82
  - remainder 22, 82
- & operator
  - address 21, 80
  - pseudovariables and 266
  - bitwise AND 22, 87
  - truth table 87
- \* operator
  - indirection 21, 80
  - pointers and 57
  - multiplication 22, 82
- + operator
  - addition 22, 83
  - unary plus 21, 81
- , operator
  - evaluation 23, 91
  - function argument lists and 24
- operator
  - subtraction 22, 83

- unary minus *21, 81*
- / operator
  - division *22, 82*
  - rounding *82*
- < operator
  - less than *22, 85*
- = operator
  - assignment *22, 90*
  - compound *90*
  - overloading *127*
  - equal to *23*
  - initializer *26*
- > operator
  - greater than *22, 85*
- ^ operator
  - bitwise XOR *22, 87*
  - truth table *87*
- | operator
  - bitwise inclusive OR *22, 88*
  - truth table *87*
- ~ operator
  - bitwise complement *21, 81*
- ! operator (not)
  - overloaded *183*
- . operator (selection) *23*
  - structure member access *66, 78*
- 1's complement (~) *21, 81*
- # symbol
  - conditional compilation and *142*
  - converting strings and *139*
  - null directive *135*
  - overloading and *124*
  - preprocessor directives *21, 26, 134*
- 80x87 coprocessors *See numeric coprocessors*
- 80x86 processors
  - address segment:offset notation *191*
  - functions (list) *160*
  - inline assembly language and *271*
  - registers *188-190, 265*

## A

- \a escape sequence (audible bell) *15*
- A TCC option (ANSI keywords) *152*
- a TCC option (word alignment) *67, 332, 333*
- abort (function)
  - destructors and *122*

- open and temporary files and *339*
- abstract classes *166, See classes, abstract*
- access
  - classes *110-111*
    - base *110*
    - default *110*
    - derived *110*
    - qualified names and *111*
  - data members and member functions *108*
  - friend classes *110*
  - friend functions *109*
  - overriding *109*
  - structure members *66, 78, 109*
  - unions
    - members *78, 109*
    - objects *331*
    - volatile objects *333*
- accounting applications *209*
- active page
  - defined *236*
  - setting *236*
- adapters, video *See video adapters*
- addition operator (+) *22, 83*
- address operator (&) *21, 80*
  - pseudovariabes and *266*
- addresses, memory *See memory addresses*
- aggregate data types *See data types*
- \_AH pseudovariabes *266*
- AH register
  - assembly language and *266*
- \_AL pseudovariabes *266*
- AL register
  - assembly language and *266*
- alert (\a) *15*
- algorithms
  - #include directive *141*
- aliases *See referencing and dereferencing*
- alignment
  - bit fields and *333*
  - structure members *332*
  - word *67, 332, 333*
- alloc.h (header file) *155*
- allocation, memory *See memory, allocation*
- ancestors *See classes, base*
- AND operator (&) *22, 87*
  - truth table *87*
- AND operator (&&) *22, 88*

- anonymous unions
  - member functions and 71
- ANSI
  - C standard
    - Turbo C++ and 3
  - date and time formats 340
  - diagnostics 327
  - extended character sets 328
  - implementation-specific items 327-340
  - integer values 329
  - keywords 8
    - predefined macro 152
  - main function
    - semantics of arguments to 328
  - multibyte characters 328
- argsused pragma 146
- arguments *See also* parameters
  - actual
    - calling sequence 63
  - command line *See* command-line compiler
  - conversions 63
  - converting to strings 139
  - default
    - constructors and 114, 116
  - to #define directive 138
  - fmod function and 336
  - function calls and 63
  - functions taking none 61
  - matching number of 64
  - parameters vs. 4
  - passing
    - C-language style 47
    - type checking 60
    - variable number of 25
      - functions (list) 164
      - Pascal and 50
- arithmetic, pointers *See* pointers, arithmetic
- arithmetic types 40
- arrays 58
  - of classes
    - initializing 102
  - constructors for
    - order of calling 118
  - elements
    - comparing 85
  - in-memory formatting and 166
  - indeterminate 58
    - structures and 59
  - initialization 43
  - integer types for 331
    - pointers to 332
  - multidimensional 58
  - new operator and 102
  - sizeof and 82
  - subscripts 23, 77
    - overloading 128
- ASCII codes
  - extended character sets 328
  - functions
    - list 157
- asm (keyword) 268
  - how to use 92
- .ASM files *See* assembly language
- asm statement
  - inline pragma and -B TCC option and 147
- aspect ratio
  - determining current 245
  - setting 236
- assembly language *See also* opcodes
  - DD and DW statements 253
  - files
    - command-line compiler option (-S) 252
  - huge functions and 51, 149
  - identifiers
    - C 253
    - defining 253
    - making visible 254
    - Pascal 253
  - inline 267
    - 80186 instructions 271
    - advantages of 270
    - C structure members and 273
      - restrictions 274
    - calling functions 272
    - commenting 268
    - directives 272
    - floating point in 207
    - goto in 274
    - inline pragma and 147
    - jump instructions 272, 274
    - option (-B) 147, 267
    - referencing data in 272
    - register variables in 273
    - semicolons and 269

- size overrides in 273
- syntax 268
- variable offsets in 273
- interfacing with 247-264
- interrupt functions 275-277
- layout of source files 252
- memory models and 252
- Pascal parameter-passing sequences and 258
- registers and 266
- routines
  - calling C functions from 255, 262
  - constants and variables in 253
  - example of 256
  - huge memory model and 252
  - linking into C programs 251
  - overlays and 217
  - passing parameters to 256
  - referencing C data from 255
  - referencing C functions from 255
  - register conventions in 261
  - return values in 257
    - examples 257
- segment directives
  - simple 251, 262
- statement syntax 92
- template 252
- viewing C code compiled as 252
- assert (function)
  - message and behavior 335
- assert.h (header file) 155
- assignment operator
  - overloading 127
- assignment operator (=) 22, 90
  - compound 90
- associativity 76, *See also* precedence
  - expressions 75
- asterisk (\*) 25
- atexit (function)
  - destructors and 122
- attributes
  - cell
    - blink 230
    - colors 229
  - control functions 225
  - screen cells 221, 229
- auto (keyword) 45
  - class members and 104

- external declarations and 36
  - register keyword and 31
- automatic objects 31
- auxiliary carry flag 189
- \_AX pseudovisible 266
- AX register 188
  - assembly language and 257, 266

## B

- \b escape sequence (backspace) 15
- b TCC option (enumerations) 71
- B TCC option (inline assembler code) 267
  - inline pragma and 147
- background color *See* graphics, colors, background
- backslash character
  - hexadecimal and octal numbers and 13
  - line continuation 140
- backslash character (\\) 15
- backspace character (\b) 15
- bad (member function) 183
- badbit (C++ error bit) 182
- banker's rounding 211
- base address register 189
- base classes *See* classes
- BCD 209
  - converting 210
  - number of decimal digits 210
  - range 210
  - rounding errors and 210
- bcd.h (header file) 155
- bell (\a) 15
- BGI OBJ (graphics converter)
  - initgraph function and 234
- \_BH pseudovisible 266
- BH register
  - assembly language and 266
- binary coded decimal *See* BCD
- binary operators *See* operators
- binary streams
  - null characters and 337
- BIOS
  - functions (list) 160
  - video output and 230
- bios.h (header file) 155
- bit fields
  - alignment and 333

- hardware registers and 69
- how treated 332
- integer 69
- order of allocation 333
- portable code and 70
- structures and 69
- unions and 71
- bit images
  - functions for 236
- bit-mapped fonts *See* fonts
- bits
  - blink enable 225
  - C++ error 182
  - color 225
  - shifting 22, 83
- bitwise
  - AND operator (&) 22, 87
    - truth table 87
  - complement operator (~) 21, 81
  - operators
    - signed integers and 331
  - OR operator (|) 22, 88
    - truth table 87
  - XOR operator (^) 22, 87
    - truth table 87
- \_BL pseudovisible 266
- BL register
  - assembly language and 266
- blink enable bit 225
- block
  - scope 29
  - statements 92
- Boolean data type 93
- \_BP pseudovisible 266
- BP register 189, 261
  - assembly language and 257, 266
  - overlays and 217
- braces 24
- brackets 23, 77
  - overloading 128
- break statements 97
  - loops and 97
- buffered files 337
- buffers
  - C++ streams and 184
  - overlays
    - default size 216

- \_BX pseudovisible 266
- BX register 188
  - assembly language and 266
- BYTE (assembler) 255

## C

- C++ 98-133
  - binary numbers 170
  - C code and 151, 179
  - character extractors 176
  - classes *See* classes
  - comments 7
  - complex numbers *See* complex numbers
  - constants *See* constants
  - constructors *See* constructors
    - ifstream 179
    - ofstream 179
    - stream classes 178
  - conversions *See* conversions, C++
  - data members *See* data members
  - declarations *See* declarations
  - destructors *See* destructors
  - enumerations *See* enumerations
  - file operations *See* files
  - fill characters 173
  - floating-point precision 173
  - for loops *See* loops, for, C++
  - formatting *See* formatting, C++
  - Fourier transforms example 208
  - functions
    - friend 105
      - access 109
    - inline *See* functions, inline
    - pointers to 54
    - taking no arguments 61
    - virtual 128
      - pure keyword and 130
  - inheritance *See* inheritance
  - initializers 44
  - input 175-178
  - keywords 8
  - member functions *See* member functions
  - members *See* data members; member
    - functions
  - name spaces 68

- operators *See* overloaded operators
- output *See* output, C++
- parameters *See* parameters
- referencing and dereferencing *See*
  - referencing and dereferencing
- scope *See* scope
- streams *See* streams, C++
- structures *See* structures
- this
  - nonstatic member functions and 105
  - static member functions and 106
- unions *See* unions
- visibility *See* visibility
- C language
  - argument passing 47
  - C++ code and 151, 179
  - calling conventions 150, 152, 250
  - Pascal calling conventions vs. 250
- CALL statement (assembler)
  - extension in TASM 2.0 262-264
- calling conventions *See also* parameters,
  - passing; Pascal
  - Pascal
    - C vs. 250
    - command-line compiler option (-p) 250
    - reasons for using 250
- calloc (function)
  - zero-size memory allocation and 338
- calls
  - far, functions using 51
  - near, functions using 51
- carriage return character 15
- carry flag 189
- case
  - preserving 50
  - sensitivity
    - external identifiers and 254, 329
    - forcing 47
    - global variables and 47
    - identifiers and 9
    - linking with no 250
    - pascal identifiers and 10
    - Turbo Assembler option 254
  - statements *See* switch statements
- cast expressions
  - syntax 79
- \_\_CDECL\_\_ macro 150
- cdecl (keyword) 47, 49, 250
  - function modifiers and 51
- cells, screen *See* screens, cells
- cerr (stream) 168
  - C++ streams and 184
- \_CH pseudovariable 266
- CH register
  - assembly language and 266
- characters
  - blinking 230
  - colors 229, 230
  - constants *See* constants, character
  - data type char *See* data types, char
  - decimal point 334
  - extractors for (C++) 176
  - fill
    - setting 173
  - functions (list) 157
  - in screen cells 221
  - input and output
    - in memory 179
  - intensity
    - setting 225
  - manipulating
    - header file 155
  - multibyte 328
  - newline
    - inserting 173
    - text streams and 336, 337
  - null
    - binary stream and 337
- sets
  - execution 328
    - collation sequence 339
    - number of bits in 329
    - source and 330
  - extended 328
  - for character constants 333
  - testing for 335
- unsigned char data type
  - range 19
- whitespace
  - extracting 173
- wide 330

- child processes
- functions (list) 163
- header file 156

- .CHR files *See* fonts, files
- cin (stream) 168
  - C++ streams and 184
- circles
  - roundness of 236
- \_CL pseudovariable 266
- CL register
  - assembly language and 266
- classes 102-113, *See also* C++; individual class names, *See also* inheritance
  - abstract 130, 166
  - access 110-111
    - default 110
    - qualified names and 111
  - arrays of
    - initialization 102
  - auto keyword and 104
  - base
    - calling constructor from derived class 120
    - constructors 120
    - pointers to
      - destructors and 123
    - private
      - friend keyword and 110
      - protected keyword and 110
      - unions and 110
    - virtual 112
      - constructors and 117
  - class names and 103
  - data types and 38
  - declarations
    - incomplete 104
  - derived
    - base class access and 110
    - calling base class constructor from 120
    - constructors 120
  - extern keyword and 104
  - file I/O 179
  - friends 112-113
    - access 110
  - initialization *See* initialization, classes
  - member functions *See* member functions
  - members, defined 104
  - naming *See* identifiers
  - objects 103, 104
    - initialization *See* initialization, classes, objects
    - register keyword and 104
    - scope *See* scope, classes
    - sizeof operator and 82
    - streambuf 167
    - streams 166, 167
    - syntax 102
    - unions and 71
    - withassign 167, 184
- \_clear87 (function)
  - floating point exceptions and 207
- clear (member function) 182
- clipping, defined 238
- clock (function)
  - era 340
- clog (stream) 168
- .CODE segment directive 251
- Code Generation dialog box 67
- code models *See* memory models
- code segment 190
- colons 25
- color *See* graphics, colors
- Color/Graphics Adapter (CGA) *See also* graphics; graphics drivers; video adapters
  - background and foreground colors 241
  - color palettes 240, 241
  - resolution 240
    - high 242
- colors *See* graphics, colors
- .COM files
  - memory models and 194
- comma
  - operator 23, 91
  - separator 24
- command-line compiler
  - nested comments 7
  - options
    - 1 (80186 instructions) 271
    - b (enumerations) 71
    - alignment (-a) 67
      - bit fields and 333
    - ANSI diagnostics and 327
    - ANSI keywords (-A) 152
    - assembly language and 267
    - assembly language output file (-S) 252
    - B (inline assembler code) 267

- inline pragma and 147
- define identifiers (-D) 137
- enumerations (-b) 71
- floating point
  - code generation (-f87) 205
  - default 206
  - emulation (-f) 205
  - fast (-ff) 205
- inline assembler code (-B) 267
  - inline pragma and 147
- overlays (-Y) 152, 216
- overlays (-Yo) 214
- Pascal
  - conventions (-p) 150, 152
- Pascal calling conventions (-p) 50, 250
- pragmas for 148
- S (produce .ASM output file) 252
- undefine (-U) 137
- underscores for C identifiers (-u) 253
- word alignment (-a) 67
- Y (overlays) 152, 216
- commas
  - nested
    - macros and 139
- comments 6
  - // 7
  - /\* \*/ 6
  - as whitespace 5
  - inline assembly language code 268
  - nested 6
  - token pasting and 6
  - whitespace and 7
- \_\_COMPACT\_\_ macro 150
- compact memory model *See* memory models, compact
- compatibility
  - streams 165
- compile-time limitations
  - header file 156
- compiler
  - diagnostic messages 283-326
- compiling
  - conditional
    - # symbol and 142
- complement
  - bitwise 21, 81
- complex declarations *See* declarations
- complex.h (header file) 155
  - complex numbers and 208
- complex numbers
  - << and >> operators and 208
  - C++ operator overloading and 208
  - example 208
  - functions (list) 161
  - header file 155, 208
  - using 208
- component selection *See* operators, selection (. and ->)
- compound assignment operators 90
- concatenating strings *See* strings, concatenating
- conditional compilation
  - # symbol and 142
  - \_\_cplusplus macro and 151
- conditional directives *See* directives, conditional
- conditional operator (? :) 89
- conforming extensions 3
- conio.h (header file) 155
  - console control and 223
- console
  - header file 155
  - I/O
    - functions 223
- const (keyword) 47, 48
  - C++ and 48
  - formal parameters and 63
  - pointers and 48, 56
- constant expressions 19
- constants 10, 47, 48, *See also* numbers
  - assembly language routines and 253
  - C++ 48
  - case statement
    - duplicate 94
  - character 11, 13
    - character set 333
    - extending 15
    - integer and 42
    - two-character 15
    - values 330
    - wide 15, 330
  - data types 12
  - decimal 11, 12
    - data types 12
    - suffixes 12

- DOS
  - header file 155
- enumerations *See* enumerations
- expressions *See* constant expressions
- floating point 11, 16
  - data types 16
  - negative 16
  - ranges 16
- fractional 11
- hexadecimal 11, 12
- integer 11
  - negative 12
- internal representations of 18
- manifest 150
- negative 12
- null pointer
  - NULL macro and 335
- octal 11, 12
- open function
  - header file 155
- pointers and 56
- string *See* strings, literal
- suffixes and 12
- symbolic
  - header file 157
- syntax 11
- Turbo C++ 15
- ULONG\_MAX and UINT\_MAX 84
- UNIX compatible
  - header file 157
- constructors 114-118, *See also* initialization
  - arrays
    - order of calling 118
  - base class
    - calling
      - from derived class 120
      - order 120
  - calling 114
  - class initialization and 119
  - classes
    - virtual base 117
  - copy 116
    - class object initialization and 119
  - default 115
  - default arguments and 114, 116
  - default parameters 116
  - delete operator and 114
  - derived class
    - order of calling 120
  - inheritance and 114
  - invoking 114
  - new operator and 114
  - non-inline
    - placement of 121
  - order of calling 117
  - overloaded 117
  - unions and 114
  - virtual 114
- continue statements 97
  - loops and 97
- continuing lines 5, 18, 140
- \_control87 (function)
  - floating point exceptions and 207
- control lines *See* directives
- conversions 41
  - arguments to strings 139
  - arrays 59
  - BCD 210
  - C++ 173
    - classes for 167
    - setting base for 173
    - streams and 169
  - character
    - C++ 176
    - integers and 42
  - date and time
    - header file 157
  - decimal 173
  - floating point
    - C++ 176
    - to smaller floating point 331
  - functions (list) 158
  - header file 156
  - hexadecimal 173
  - integers
    - C++ 176
    - character and 42
    - to floating point 331
    - to pointers 332
  - octal 173
  - pointers 58
    - to integers 332
  - sign extension and 42
  - special 42

- standard 42
- when value can't be represented 331
- coordinates
  - origin 223
  - returning 226
  - starting positions 222, 227
- coprocessors *See* numeric coprocessors
- copy constructors *See* constructors, copy
- cout (stream) 168
  - C++ streams and 184
- `__cplusplus` macro 151
- CPP.EXE (preprocessor) 133
- CPU (central processing unit) *See* 80x86 processors
  - `_cs` (keyword) 47, 199
  - `_CS` pseudovisible 266
- CS register 190, 192
  - assembly language and 266
- ctype.h (header file) 155
- current position, files *See* files, position within
- cursor
  - changing 227
  - control
    - header file 223
    - manipulating onscreen 224
  - position
    - setting 224
- `_CX` pseudovisible 266
- CX register 188
  - assembly language and 266

## D

- `\D` escape sequence (display a string of octal digits) 15
- `-D` TCC option (define identifier) 137
- data, binary, reading 177
- `.DATA?` segment directive 251
- `.DATA` segment directive 251
- data members *See also* member functions
  - access 108
  - dereference pointers 23
  - private 108
  - protected 109
  - public 108
  - scope 107-109
  - static 106
    - declaration 107

- definition 107
  - uses 107
- data models *See* memory models
- data segments 190
- data structures *See* structures
- data types 27, *See also* constants; floating point; integers; numbers
  - aggregate 38
  - arithmetic 40
  - BCD *See* BCD
  - Boolean 93
  - C++ streams and 169
  - char 40
    - range 19
      - signed and unsigned 15, 40
  - classes and 38
  - conversions *See* conversions
  - declarations 39
  - declaring 38
  - default 38
  - defining
    - header file 156
  - derived 38
  - enumerations *See* enumerations
    - range 19
  - function return types 60
  - fundamental 38, 39
    - creating 40
  - identifiers and 27, 28
  - integers *See* integers
  - integral 40
  - internal representations 40
  - memory use 81
  - new, defining 46
  - ranges 19
  - scalar 38
    - initializing 43
  - `size_t` 82, 125, 126
  - `sizeof` operator 334
  - table of 19
  - taxonomy 38
  - `text_modes` 228
  - `time_t`
    - header file 157
  - types of 38
  - unsigned char
    - range 19

- void 39
- wchar\_t 15
- date
  - formats 340
  - functions (list) 164
  - local
    - how defined 339
  - macro 151
  - \_\_DATE\_\_ macro 151
  - availability 334
  - #define and #undef directives and 137
- DD statement (assembler) 253
- deallocation, memory *See* memory, allocation
- debugging
  - macros
    - header file 155
  - overlays 217
- dec (manipulator) 173
- decimal constants *See* constants, decimal
- decimal point
  - how displayed 334
- declarations 26
  - arrays 58
  - C++ 38
    - incomplete 104
  - complex 52
    - examples 52, 53
  - data types 38
    - default 38
  - defining 27, 32, 33, 44
    - extern keyword and 45
  - examples 39
  - external 32, 36
    - storage class specifiers and 36
  - function *See* functions, declaring
  - incomplete class 104
  - with initializers
    - bypassing 97
  - mixed languages 49
  - modifiers and 46
  - objects 34
  - Pascal 49
  - pointers 55
  - referencing 27, 33
    - extern keyword and 45
  - simple 44
  - static data members 107
  - structures *See* structures, declaring
  - syntax 33, 34
  - tentative definitions and 34
  - unions 71
- declarators
  - number of 333
  - pointers (\*) 25
  - syntax 53
- decrement operator (–) 21, 79
- default (label)
  - switch statements and 94
- default constructors *See* constructors, default
- #define directive 135
  - argument lists 138
  - global identifiers and 137
  - keywords and 137
  - redefining macros with 136
  - with no parameters 135
  - with parameters 137
- defined operator 143
- defining declarations *See* declarations, defining
- definitions *See* declarations, defining
  - function *See* functions, definitions
  - tentative 34
- delete (operator) 101
  - constructors and destructors and 114
  - destructors and 122
  - dynamic duration objects and 32
  - overloading 125
  - pointers and 122
- dereferencing *See* referencing and dereferencing
- derived classes *See* classes
- derived data types *See* data types
- descendants *See* classes, derived
- destructors 114, 121-124, *See also* initialization
  - abort function and 122
  - atexit function and 122
  - base class pointers and 123
  - calling 114
  - class initialization and 119
  - delete operator and 114, 122
  - exit function and 122
  - global variables and 122
  - inheritance and 114
  - invoking 114, 122
    - explicitly 122

- new operator and 114, 122
- pointers and 122
- #pragma exit and 122
- unions and 114
- virtual 114, 123
- \_DH pseudovisible 266
- DH register
  - assembly language and 266
- \_DI pseudovisible 266
- DI register 189
  - assembly language and 261, 266
- diagnostic messages
  - ANSI 327
  - compiler 283-326
- digits
  - hexadecimal 11
  - nonzero 11
  - octal 11
- dir.h (header file) 155
- direct member selector *See* operators, selection (. and ->)
- direct video output 230
- direction flag 189
- directives 133-152, *See also* individual directive names; macros
  - ## symbol
    - overloading and 124
  - # symbol 26
    - overloading and 124
  - conditional 142
    - nesting 142
  - conditional compilation and 142
  - error messages 145
  - keywords and 137
  - line control 144
  - pragmas *See* pragmas
  - segment 251, 262
  - sizeof and 82
  - syntax 134
  - usefulness of 133
- directories
  - functions (list) 158
  - header file 155
  - include files
    - how searched 334
- division operator (/) 22, 82
  - rounding 82

- \_DL pseudovisible 266
- DL register
  - assembly language and 266
- do while loops *See* loops, do while
- domain errors
  - mathematics functions and 335
- DOS
  - environment
    - 87 variable 206
    - strings
      - changing permanently 339
    - functions (list) 160
    - header file 155
  - dos.h (header file) 155
  - dot operator (selection) *See* operators, selection (. and ->)
  - double quote character
    - displaying 15
  - drawing color *See* graphics, colors
  - drawing functions 234
  - \_ds (keyword) 47, 199
  - \_DS pseudovisible 266
  - DS register 190, 192
    - assembly language and 266
  - duplicate case constants 94
  - duration 31
    - dynamic 32
      - memory allocation and 32
    - local 31
      - local scope and 31
    - pointers 55
    - static 31
  - DW statement (assembler) 253
  - DWORD (assembler) 255
  - \_DX pseudovisible 266
  - DX register 188
    - assembly language and 257, 266
  - dynamic duration 32
    - memory allocation and 32
  - dynamic memory allocation *See* memory, allocation

## E

- elaborated type specifier 104
- #elif directive 142
- ellipsis (...) 25
  - prototypes and 61, 64

- #else directive 142
- empty statements 93
- EMU.LIB 205
- emulating the 80x87 math coprocessor *See*
  - floating point, emulating
- enclosing block 29
- #endif directive 142
- endl (manipulator) 173
- ends (manipulator) 173
- Enhanced Graphics Adapter (EGA) *See also*
  - graphics drivers; video adapters
    - color control on 242
- enum (keyword) *See* enumerations
- enumerations 71
  - C++ 72
  - class names and 103
  - command-line option (-b) 71
  - constants 11, 17, 72
    - default values 17
  - conversions 42
  - default type 71
  - name space 30
  - open\_mode (C++) 180
  - range 19
  - scope, C++ 73
  - structures and
    - name space in C++ 68
  - tags 72
    - name spaces 73
  - values 333
- environment
  - DOS
    - 87 variable 206
    - header file 156
- eof (member function) 182
- eofbit (C++ error bit) 182
- equal to operator (=) 23
- equal-to operator (==) 86
- equality operators *See* operators, equality
- era, clock function and 340
- errno.h (header file) 155
- #error directive 145
- errors
  - C++ streams 181
    - member functions for testing 182
  - command line
    - defined 283
  - compiler 283-318
    - fatal 284
  - defined 283
  - disk access 283
  - domain
    - mathematics functions and 335
  - expressions 77
  - fatal 283
  - floating point
    - disabling 207
  - graphics, functions for handling 243
  - math, masking 207
  - memory access
    - defined 283
  - messages
    - assert function 335
    - graphics 243
    - list 283-318
    - perror function 338
    - strerror function 339
  - mnemonics for codes 155
  - preprocessor directive for 145
  - run time 280
  - syntax
    - defined 283
  - underflow range
    - mathematics functions and 335
- \_es (keyword) 47, 199
- \_ES pseudovisible 266
- ES register 190
  - assembly language and 266
- escape sequences 11, 13
  - length 14
  - number of digits in 14
- octal
  - non-octal digits and 14
- source files and 334
- table of 15
- evaluation order *See* precedence
- exclusive OR operator (^) 22, 87
  - truth table 87
- execution character sets *See* characters, sets,
  - execution
- exit (functions) 339
  - destructors and 122
- exit pragma 146

- expanded and extended memory
  - overlays and 218
- exponents 11
- \_export (keyword) 47, 51, 52
- expressions
  - associativity 75
  - cast, syntax 79
  - constant 19
  - conversions and 41
  - decrementing 79
  - empty (null statement) 24, 93
  - errors and overflows 77
  - floating point
    - precedence 76
  - function
    - sizeof and 82
  - grouping 23
  - incrementing 79
  - precedence 75, 76
  - statements 24, 93
  - syntax 74
  - table 74
- extent *See* duration
- extern (keyword) 45, *See also* identifiers, external
  - arrays and 58
  - class members and 104
  - const keyword and 48
  - linkage and 32
- external
  - declarations 32
  - identifiers *See* identifiers, external
  - linkage *See* linkage
- ExternFunc 217
- extra segment 190
- extraction operator (<<) *See* overloaded operators, << (get from)
- extractors *See* input, C++
- EXTRN statement (assembler) 255, 262, 264
  - extension in TASM 2.0 262-264
  - huge memory model and 256
  - variable size and 255

## F

- f87 TCC option (generate floating-point code) 205
- \f escape sequence (formfeed) 15

- f TCC option (emulate floating point) 205
- fail (member function) 182
- failbit (C++ error bit) 182
- far
  - calls
    - memory model and 216
    - requirement 216
  - functions *See* functions, far
  - pointers *See* pointers, far
- far (keyword) 47, 192, 199, 204
- fcntl.h (header file) 155
- ff TCC option (fast floating point) 205
- fgetpos (function)
  - errno value on failure of 338
- field width *See* formatting, width (C++)
- \_FILE\_ macro 151
  - #define and #undef directives and 137
- file-position indicator
  - initial position 337
- file scope *See* scope
- filebuf (class)
  - base class of 167, 179
- files *See also* individual file-name extensions
  - appending
    - file-position indicator and 337
  - .ASM *See* assembly language
  - buffering 337
  - closing
    - C++ 179
  - current
    - macro 151
  - font *See* fonts
  - graphics driver, linking 233
  - header *See* header files
  - include *See* include files
  - including 140
  - input and output
    - C++ 179
  - names
    - searching for 334
  - open
    - abort function and 339
    - remove function and 337
  - opening
    - C++ 179, 180
    - default mode 180
    - multiple times 337

- position within
  - C++ 181
- position within, C++ 181
- project
  - graphics library listed in 231
- reading
  - header file 155
- renaming
  - preexisting file name and 338
- scope *See* scope
- searching
  - C++ 179
- sharing
  - header file 156
- source
  - escape sequences and 334
- streams
  - declaring 179
- temporary
  - abort function and 339
- truncation while writing to 337
- writing
  - header file 155
- zero-length 337
- fill characters
  - C++ 173, 174
- filling functions 234
- financial applications 209
- flags
  - format state *See* formatting, format state
  - flags (C++)
  - ios (class)
    - setting 173
  - register 187, 189
- float.h (header file) 155
- floating point *See also* data types; integers;
  - numbers
    - arithmetic
      - interrupt functions and 275
    - constants *See* constants
    - conversions *See* conversions
    - decimal point character 334
    - double
      - range 19
    - emulating 205
    - exceptions
      - disabling 207
    - expressions
      - precedence 76
    - extractors for (C++) 176
    - fast 205
    - formats 329
    - functions (list) 161
    - header file 155
    - libraries 204
    - long double
      - range 19
    - precision
      - setting 173
    - ranges 19
    - registers and 207
    - using 204
- flow-control statements *See* if statements;
  - switch statements
- flush (manipulator) 173
- fmod (function)
  - second argument of zero 336
- fonts
  - bit-mapped
    - stroked vs. 238
    - when to use 238
  - clipping 238
  - files
    - loading and registering 238
  - height and width 238
  - information on current settings 245
  - registering 239
  - setting size 238
  - stroked
    - advantages of 238
- for loops *See* loops, for
- foreground color *See* graphics, colors,
  - foreground
- formal parameters *See* parameters, formal
- format state flags *See* formatting, C++, format
  - state flags
- formatting
  - C++
    - fill character 173, 174
    - format state flags 170
    - I/O 173, *See also* manipulators
    - output 170
    - padding 174
    - width functions 171, *See also* manipulators

- setting 173, 177
  - in memory 166
  - streams and 166
    - clearing 173
- formfeed character 15
- forward references 27
- Fourier transforms
  - complex number example 208
- FP\_OFF 202
- FP\_SEG 202
- fprintf (function)
  - %p conversion output 338
- free (function)
  - delete operator and 101
  - dynamic duration objects and 32
- free union variant record *See* unions
- friend (keyword) 105, 112-113
  - base class access and 110
  - functions and *See* C++, functions, friend
- fscanf (function)
  - %p conversion input 338
- fstream.h (header file) 155
- ftell (function)
  - errno value on failure of 338
- function call operator *See* parentheses
- function operators *See* overloaded operators
- functions 59-64
  - 8086 160
  - arguments
    - no 61
  - assembly language and
    - return values 257
  - attribute control 225
  - bcd
    - header file 155
  - BIOS 160
    - header file 155
  - calling 63, *See also* parentheses
    - from assembly language routines 262
    - in inline assembly code 272
    - operators ( ) 78
    - overloading operator for 128
    - rules 63
  - cdecl and 50
  - child processes 163
    - header file 156
  - class names and 103

- classification 157
- color control 239
- comparing 87
- complex numbers 161
  - header file 155
- console
  - header file 155
  - I/O 223
- conversion 158
- date and time 164
  - header file 157
- declaring 59, 60
  - as near or far 200
- default types for memory models 51
- definitions 59, 62
- diagnostic 158
- directories 158
  - header file 155
- DOS 160
- drawing 234
- duration 31
- error-handling, graphics 243
- exit 146
- external 45
  - declarations 32
- far 51
  - declaring 201
  - memory model size and 200
- file sharing
  - header file 156
- filling 234
- floating point
  - header file 155
- friend *See* C++, functions, friend
- fstream
  - header file 155
- generic
  - header file 155
- goto 163
  - header file 156
- graphics 158, *See also* graphics
  - drawing operations 234
  - fill operations 235
  - header file 155
  - using 231-245
- graphics system control 231
- huge 51

- assembly language and 51
- `_loadds` and 52
- saving registers 149
- image manipulation 236
- inline
  - assembly language *See* assembly language,
  - inline
  - C++ 105
  - linkage 107
- integer 161
- internal linkage 45
- international
  - header file 156
  - information 163
- interrupt *See* interrupts, functions
- I/O 159
  - header file 155
- iomanip
  - header file 155
- iostream
  - header file 156
- listed by topic 157-164
- locale 163
- main 59
- mathematical 161
  - domain errors 335
  - header file 156
  - underflow range errors 335
- member *See* member functions
- memory 161
  - allocating and checking 162
  - header file 156
  - models and 47
- mode control 225
- near 51
  - declaring 201
  - memory models and 200
- no arguments 39
- not returning values 39
- operators *See* overloaded operators
- overloaded *See* overloaded functions
- Pascal
  - calling 49
- pixel manipulation 236
- pointers 54
  - calling overlaid routines 217
  - object pointers vs. 53

- pointers to
  - void 54
- process control 163
- prototypes *See* prototypes
- recursive
  - memory models and 200
- return statements and 97
- return types 60
- scope *See* scope
- screen manipulation 236
- signals
  - header file 156
- sizeof and 82
- sound 163
- standard routines 163
- startup 146
- state queries 226, 243
- static 32
- stdarg.h header file and 61
- stdiostr
  - header file 156
- storage class specifiers and 33
- strings 161
- strstrea
  - header file 157
- structures and 66
- text
  - manipulation 224
  - output
    - graphics mode 237
- Turbo C++
  - licensing 154
- type
  - modifying 51
- variable argument lists 164
- viewport manipulation 236
- width (C++) 171
- windows 163, 225

fundamental data types *See* data types

## G

- generate underbars option 250
- generic.h (header file) 155
- generic pointers 39, 55
- get (function)
  - C++ input and 177

- get from operator (>>) *See* overloaded operators, >> (get from)
- getenv (function)
  - environment names and methods 339
- global identifiers *See* identifiers, global
- global variables 29, *See also* variables
  - case sensitivity and 47
  - destructors and 122
  - \_ovrbuffer 213, 217
  - underscores and 47
  - \_wscroll 224
- good (member function) 182
- goodbit (C++ error bit) 182
- goto statement 97
- goto statements
  - assembly language and 274
  - functions (list) 163
  - header file 156
  - labels
    - name space 29
- grammar
  - lexical 4
  - parsing 5
  - phrase structure 4
  - tokens *See* tokens
- graphics *See also* graphics drivers
  - buffers 236
  - circles
    - aspect ratio 236
  - colors *See also* graphics, palettes
    - background 225
      - CGA 241
      - defined 229, 240
      - list 230
      - setting 225
    - CGA 240, 241
    - drawing 240
    - EGA/VGA 242
    - foreground 225
      - CGA 241
      - defined 229
      - list 230
      - setting 225
    - functions 239
    - information on current settings 245
  - coordinates *See* coordinates
  - default settings
    - restoring 233
  - displaying 240
  - drawing functions 234
  - errors
    - functions to handle 243
  - fill
    - operations 235
    - patterns 235
      - using 245
  - functions
    - list 158
    - using 231-245
  - header file 155, 231
  - library 231
  - line style 235
  - memory for 234
  - page
    - active
      - defined 236
      - setting 236
    - visual
      - defined 236
      - setting 236
  - palettes *See also* graphics, colors
    - defined 239
    - functions 239
    - information on current 245
  - pixels *See also* screens, resolution
    - colors
      - current 245
      - functions for 236
      - setting color of 239
  - setting
    - clearing screen and 236
  - state queries 243
  - system
    - control functions 231
    - shutting down 233
    - state queries 245
  - text and 237
  - viewports
    - defined 223
    - functions 236
    - information on current 245

graphics drivers *See also* Color/Graphics Adapter (CGA); Enhanced Graphics Adapter (EGA); graphics; video adapters; Video Graphics Array Adapter (VGA)  
 current 233, 245  
   returning information on 245  
 linking 233  
 loading and selecting 232, 233  
 new  
   adding 233  
   registering 234  
   returning information on 245  
   supported by Turbo C++ 232  
 graphics.h (header file) 155, 231  
 greater-than operator (>) 22, 85  
 greater-than or equal-to operator (>=) 22, 86

## H

hardfail (C++ error bit) 182  
 header files 155-157, *See also* include files  
   C++ streams 166  
   complex numbers 208  
   described 155  
   floating point 155  
   function prototypes and 61  
   graphics 231  
   #include directive and 140  
   prototypes and 59, 153  
   reading and writing 155  
   sharing 156  
   variable parameters 61  
 heap 32  
 Hercules card *See* graphics drivers; video adapters  
 hex (manipulator) 173  
 hexadecimal  
   constants *See* constants, hexadecimal  
   digit 11  
 hidden objects 30  
 hiding *See* scope, C++  
 horizontal tab 15  
 huge  
   functions  
     saving registers and 149  
   memory model *See* memory models  
   pointers *See* pointers, huge  
 \_\_HUGE\_\_ macro 150

huge (keyword) 47, 192, 199  
   assembly language and 51

## I

identifiers 9  
   assembly language  
     making visible 254  
   C++  
     reading from standard input 178  
   case 50  
     sensitivity and 9  
   classes 103  
   data types and 27, 28  
   declarations and 27  
   declaring 44  
   defined operator and 143  
   defining 137  
   defining in assembly language routines 253  
   duplicate 30  
   duration 31  
   enumeration constants 17  
   external 253, *See also* extern (keyword)  
     case sensitivity and 254, 329  
   global 150  
     #define and #undef directives and 137  
   length 329  
   linkage 32, 33  
     attributes 32  
   mixed languages 49  
   name spaces *See* name spaces  
   no linkage attributes 33  
   Pascal 49  
     assembly language and 253  
   pascal (keyword)  
     case sensitivity and 10  
   rules for creating 9  
   scope *See* scope  
   significant characters in 329  
   storage class and 28  
   testing for definition 143  
   Turbo C++ keywords as 3  
   undefining 137  
   underscores and 253  
     command-line compiler option 253  
   unique 33  
 IEEE  
   floating-point formats 41, 329

- rounding 211, 331
- #if directive 142
- if statements 93
  - nested 93
- #ifdef directive 143
- #ifndef directive 143
- ifstream (class)
  - constructor 179
  - insertion operations 179
- implementation-specific ANSI items 327-340
- include files *See also* header files
  - #include directive and 140
  - search algorithm for 141
  - searching for 334
- #include directive 140
  - search algorithm 141
- inclusive OR operator (|) 22, 88
  - truth table 87
- incomplete declarations
  - classes 104
  - structures 68
- increment operator (++) 21, 78, 79
- indeterminate arrays 58
  - structures and 59
- indirect member selector *See* operators, selection
- indirection operator (\*) 21, 80
  - pointers and 57
- inequality operator (!=) 23, 87
- inheritance *See also* classes
  - constructors and destructors 114
  - multiple
    - base classes and 112
  - overloaded assignment operator and 127
  - overloaded operators and 125
- initialization 42, *See also* constructors; destructors
  - arrays 43
  - classes 119
    - objects 119
    - copy constructor and 119
  - operator 26
  - pointers 55
  - static member definitions and 107
  - structures 43
  - unions 43, 71
  - variables 44

- initializers
  - automatic objects 44
  - C++ 44
  - new operator and 102
- inline
  - assembly language code *See* assembly language, inline
  - expansion 105
  - functions *See* functions, inline
  - pragma 147
- inline (keyword) 105
- inline pragma 268
- input
  - C++
    - binary numbers 170
    - characters and 176
    - floating point and 176
    - integers and 176
    - user-defined types 178
  - chaining operations 175
- inserters *See* output, C++
- insertion operator *See* overloaded operators, << (put to)
- instances *See* classes, objects
- INT instruction 275
- integers 40, *See also* data types; floating point; numbers
  - arrays and 331
  - C++
    - default number base 171
    - C++ streams and 169
    - casting to pointer 332
    - constants *See* constants
    - conversions *See* conversions
    - division
      - sign of remainder 332
    - enumerations and 333
    - expressions
      - precedence 76
    - extractors for (C++) 176
    - functions (list) 161
    - long 40
      - range 19
    - memory use 40
    - pointers and 332
    - range 19

- ranges
    - header file *156*
  - right shifted *332*
  - short *40*
  - signed
    - bitwise operators and *331*
  - sizes *40*
  - suffix *11*
  - unsigned
    - range *19*
  - values *329*
  - integral data types *See* characters; integers
  - integrated environment
    - nested comments command *7*
  - intensity
    - setting *225*
  - interfacing with assembly code *247-264*
  - internal linkage *See* linkage
  - internal representations of data types *40*
  - international information
    - functions (list) *163*
    - header file *156*
  - interrupt (keyword) *47, 48, 275*
  - interrupts
    - beep
      - example *276*
    - flag *189*
    - functions
      - assembly language and *275-277*
      - example of *276*
      - floating-point arithmetic in *275*
      - memory models and *48*
      - void *48*
    - handlers *47*
      - assembly language and *275-277*
      - calling *277*
      - installing *277*
      - modules and *217*
      - programming *275*
    - registers and *48*
  - io.h (header file) *155*
  - I/O
    - C++
      - errors *181*
      - formatting *173*
      - memory and *179*
      - precision *173*
    - functions (list) *159*
    - low level
      - header file *155*
  - ioanip.h (header file) *155*
  - ios (class)
    - contents *167*
    - derived classes of *167*
    - error bits *182*
    - flags
      - format state *170*
      - setting *173*
    - open\_mode enumeration *180*
  - iostream (class)
    - base class of *167*
    - bidirectional operations and *167*
  - iostream.h (header file) *156, 166*
  - iostream library *166*
  - IP (instruction pointer) register *187*
  - isalnum (function) *335*
  - isalpha (function) *335*
  - iscntrl (function) *335*
  - islower (function) *335*
  - isprint (function) *335*
  - istream (class)
    - base class of *167*
    - constructors *178*
    - contents of *167*
    - derived classes of *179*
    - old vs. new streams *184*
    - overflowing *177*
  - isupper (function) *335*
  - iteration statements *See* loops
- ## J
- jump instructions, inline assembly language
    - table *272*
    - using *274*
  - jump statements *See* break statements; continue statements; goto statements; return statements
- ## K
- keywords *8, See also* individual keyword names
    - ANSI
      - predefined macro *152*
    - C++ *8*

- combining 40
- macros and 137
- Turbo C++
  - using as identifiers 3

## L

- labeled statements 92
- labels
  - creating 25
  - default 94
  - function scope and 29
  - goto statement and 97
  - in inline assembly code 274
- language extensions
  - conforming 3
  - `__LARGE__` macro 150
- large code
  - data
    - and memory models *See* memory models
- less-than operator (<) 22, 85
- less-than or equal-to operator (<=) 22, 85
- lexical grammar *See* grammar
- libraries
  - files (list) 153
  - floating point
    - using 204
  - graphics 231
  - iostream 166
  - prototypes and 64
  - stream 165
  - underscores on identifiers 253
- limits.h (header file) 156
- `__LINE__` macro 151
- `#define` and `#undef` directives and 137
- `#line` directive 144
- lines
  - continuing 5, 18, 140
  - numbers 144
    - macro 151
- linkage 32
  - external 32, 33
    - C++ constants and 48
  - internal 32, 33
  - no 32, 33
  - rules 33
  - static member functions 107
  - storage class specifiers and 32

- linker
  - mixed modules and 203
  - using directly 203
- literal strings *See* strings, literal
- `_loadds` (keyword) 47, 51, 52
  - huge functions and 52
- local duration 31
- locale
  - functions (list) 163
- locale.h (header file) 156
- logical AND operator (&&) 22, 88
- logical negation operator (!) 21, 81
- logical OR operator (||) 22, 88
- long integers *See* integers, long
- longjmp (function)
  - header file 156
- loops 95
  - break statement and 97
  - continue statement and 97
  - do while 95
  - for 95
    - C++ 96
    - while 95
      - string scanning and 95
- low-level programming 264-277
  - setting registers and 265
- lvalues 28, *See also* rvalues
  - examples 52
  - modifiable 28

## M

- macros *See also* directives
  - argument lists 138
    - header file 156
  - assert 155
  - calling 138
  - character conversion
    - header file 155
  - commas and
    - nested 139
  - debugging
    - assert
      - header file 155
  - defining 135
    - conflicts 136
    - global identifiers and 137
    - header file 156

- directory manipulation
  - header file 155
- expansion 135
- far pointer creation 202
- keywords and 137
- MK\_FP 202
- NULL
  - expansion 335
- parameters and 137
  - none 135
- parentheses and
  - nested 139
- precedence in
  - controlling 24
- predefined 150, *See also* individual macro names
  - ANSI keywords 152
  - C calling conventions 150
  - conditional compilation 151
  - current file 151
  - current line number 151
  - date 151
  - DOS 151
  - memory models 150
  - overlays 152
  - Pascal calling conventions 152
  - time 152
  - Turbo C++ version number 152
- redefining 136
- side effects and 140
- undefining 136
  - global identifiers and 137
- main (function) 59
  - pascal keyword and 50
  - semantics of arguments to 328
- malloc (function)
  - dynamic duration objects and 32
  - new operator and 101
  - zero-size memory allocation and 338
- manifest constants 150
- manipulators 172, *See also* C++, formatting, width; individual manipulator names
  - parameterized 172
  - syntax 173
- math
  - BCD *See* BCD
  - coprocessors *See* numeric coprocessors
  - errors
    - masking 207
  - functions
    - domain errors and 335
    - list 161
    - underflow range errors and 335
- math.h (header file) 156
- matherr (function)
  - proper use of 207
- \_\_MEDIUM\_\_ macro 150
- medium memory model *See* memory models
- mem.h (header file) 156
- member functions 105, *See also* data members
  - access 108
  - constructors *See* constructors
  - defined 104
  - destructors *See* destructors
  - error testing 182
  - friend 105
  - inline *See* functions, inline, C++
  - nonstatic 105
  - private 108
  - protected 109
  - public 108
  - pushing characters 177
  - scope 107-109
  - static 106
    - linkage 107
    - this keyword and 106
  - structures and 65
  - this keyword and 105, 106
  - unions and 71
- members, classes *See* data members; member functions
- members, structures *See* structures, members
- memory *See also* memory addresses
  - allocation 32
    - assembly language code and huge functions and 51
    - functions (list) 162
    - graphics system 234
    - new and delete operators and 101
    - structures 67
  - C++ streams and 179

- checking 162
- data types 81
- expanded and extended *See* expanded and extended memory
- formatting in 166
- functions (list) 161
- header file 155, 156
- heap 32
- memory models and 195
- overlays and 213
- paragraphs 191
  - boundary 191
- segments in 190
- Turbo C++'s usage of 211
- word alignment and structures 67
- memory addresses *See also* memory calculating 189, 191
  - constructors and destructors 114
  - far pointers and 192
  - near pointers and 192
  - pointing to 202
  - pseudovariables and 266
  - segment:offset notation 191
  - standard notation for 191
- memory models 198, 187-204
  - assembly language code and 252
  - changing 202
  - compact 194
    - default function type 51
  - comparison 198
  - default
    - overriding 51
  - defined 194
  - function pointers and 54
  - functions
    - default type
    - overriding 47
    - list 162
  - graphics library 231
  - huge 195
    - assembly language routines and 252
    - default function type 51
    - EXTRN statement and (assembler) 256
  - illustrations 195-198
  - interrupt functions and 48
  - large 195
    - default function type 51
    - libraries 153
    - macros and 150
    - math files for 153
    - medium 194
      - default function type 51
    - memory apportionment and 195
    - mixing 203
      - function prototypes and 203
    - overlays and 214, 216
    - pointers
      - modifiers and 51
    - pointers and 192, 200
    - predefined macros and 150
    - segment directives and 251
    - small 194
      - default function type 51
    - tiny 194
      - default function type 51
  - memory-resident routines 276
  - methods *See* member functions
  - mixed-language programming 247
  - mixed modules
    - linking 203
  - MK\_FP (run-time library macro) 202
  - mnemonics, error code 155
  - .MODEL segment directive 251
  - modifiable lvalues *See* lvalues
  - modifiable objects *See* objects
  - modifiers 46
    - function type 51
    - pointers 51, 200
    - table 47
  - Modula-2
    - variant record types 70
  - modules
    - linking mixed 203
    - size limit 198
  - modulus operator (%) 22, 82
  - Monochrome Display Adapter *See* graphics drivers; video adapters
  - \_\_MSDOS\_\_ macro 151
  - multibyte characters 328
  - multidimensional arrays *See* arrays
  - multiple inheritance *See* inheritance
  - multiplication operator (\*) 22, 82
  - /mx Turbo Assembler option 254

## N

- \n (newline character) 15
  - name spaces
    - scope and 29
    - structures 68
      - C++ 68
  - names *See* identifiers
    - qualified 108
  - near (keyword) 47, 192, 199
  - near functions *See* functions, near
  - near pointers *See* pointers, near
  - negation
    - logical (!) 21, 81
  - negative offsets 189
  - nested
    - comments 6, 7
    - conditional directives 142
    - declarators 333
  - new (operator) 101
    - arrays and 102
    - constructors and destructors and 114
    - destructors and 122
    - dynamic duration objects and 32
    - initializers and 102
    - overloading 102, 125
    - prototypes and 101
  - new lines, creating in output 15
  - newline characters
    - inserting 173
  - no linkage *See* linkage
  - nondefining declarations *See* declarations, referencing
  - nonzero digit 11
  - normalized pointers *See* pointers, normalized
  - not equal to operator (!=) 23, 87
  - not operator (!) 21, 81
    - overloaded 183
  - NULL
    - macro 335
    - using 55
  - null
    - characters
      - binary stream and 337
    - directive (#) 135
    - inserting in string 173
    - pointer constant 335
    - pointers 55
      - statement 24, 93
  - number of arguments 25
  - numbers *See also* constants; data types; floating point; integers
    - base
      - setting for conversion 173
    - BCD *See* BCD
    - binary
      - C++ and 170
      - reading (C++) 177
    - converting *See* conversions
    - decimal
      - conversions 173
    - functions (list) 161
    - hexadecimal 11
      - backslash and 13
      - conversions 173
      - displaying 15
    - lines *See* lines, numbers
    - octal 11
      - backslash and 13
      - conversions 173
      - displaying 15
      - escape sequence 14
  - numeric coprocessors *See also* 80x86 processors
    - autodetecting 206
    - built in 205
    - floating-point emulation 205
    - floating-point format 329
    - registers and 207
- ## O
- .OBJ files
    - converting .BGI files to 234
  - objects 27, *See also* classes
    - aliases 98
    - automatic 31
      - initializers 44
    - class names and 103
    - duration 31
    - hidden 30
    - initializers 44
    - list of declarable 34
    - modifiable 48
    - pointers 54
      - function pointers vs. 53

- static
  - initializers 44
  - temporary 100
  - volatile 48
    - accessing 333
- oct (manipulator) 173
- octal constants *See* constants, octal
- octal digit 11
- offsets 192
  - component of a pointer 202
- ofstream (class)
  - base class 179
  - constructor 179
  - insertion operations 179
- opcodes *See also* assembly language
  - defined 268
  - mnemonics
    - command-line compiler option (-1) 271
    - table 270
  - repeat prefixes 272
- open (function)
  - header file 155
- open mode *See* files, opening, C++
- open\_mode (enumeration) 180
- operands (assembly language) 268
- operating mode of screen *See* screens, modes
- operator (keyword)
  - overloading and 124
- operator functions *See* overloaded operators
- operators 20-23, 77
  - 1's complement (~) 21, 81
  - addition (+) 22, 83
  - address
    - pseudovariables and 266
  - address (&) 21, 80
  - AND (&) 22, 87
    - truth table 87
  - AND (&&) 22, 88
  - assignment (=) 22, 90
    - compound 90
    - overloading 127
  - binary 22
    - overloading 127
  - bitwise
    - AND (&) 22, 87
      - truth table 87
    - complement (~) 21, 81
    - inclusive OR (|) 22, 88
      - truth table 87
    - signed integers and 331
    - truth table 87
    - XOR (^) 22, 87
      - truth table 87
  - C++ 21
    - delete 101, *See* delete (operator)
    - dereference pointers 23
    - new *See* new (operator)
    - scope (::) 23, 100
  - conditional (? :) 23, 89
  - context and meaning 21
  - decrement (-- ) 21, 79
  - defined operator 143
  - division (/) 22, 82
    - rounding 82
  - equality 23, 86
  - evaluation (comma) 23, 91
  - exclusive OR (^) 22, 87
    - truth table 87
  - function call ( ) 78
  - inclusive OR (|) 22, 88
    - truth table 87
  - increment (++) 21, 78, 79
  - indirection (\*) 21, 80
    - pointers and 57
  - inequality (!=) 23, 87
  - list 20
  - logical
    - AND (&&) 22, 88
    - negation (!) 21, 81
    - OR (| |) 22, 88
  - manipulators *See* manipulators
  - modulus (%) 22, 82
  - multiplication (\*) 22, 82
  - not (!) 183
  - OR (^) 22, 87
    - truth table 87
  - OR (|) 22, 88
    - truth table 87
  - OR (| |) 22, 88
  - overloading *See* overloaded operators
  - postfix 77
  - prefix 77
  - relational 22, 84
  - remainder (%) 22, 82

- selection (. and ->) 23, 78
    - overloading 128
    - structure member access and 66, 78
  - shift bits (<< and >>) 22, 83
  - sizeof 81
    - data type 334
  - subtraction (-) 22, 83
  - unary
    - overloading 127
    - unary minus (-) 21, 81
    - unary plus (+) 21, 81
  - option pragma 148
  - OR operator
    - bitwise inclusive (|) 22, 88
    - truth table 87
    - logical (||) 22, 88
  - OS/2
    - compatibility 52
  - ostream (class)
    - base class of 167
    - constructors 178
    - contents of 167
    - derived classes of 179
    - old vs. new streams 184
  - output
    - C++ 168-175
      - binary numbers 170
      - user-defined types 174
    - directing 230
    - functions 224
  - overflows
    - expressions and 77
    - flag 189
  - \_\_OVERLAY\_\_ macro 152
  - overlays 211-220
    - assembly language routines and 217
    - BP register and 217
    - buffers
      - default size 216
    - cautions 217
    - command-line options (-Yo) 214
    - debugging 217
    - designing programs for 216
    - expanded and extended memory and 218
    - how they work 212
    - large programs 211
    - memory map 213
    - memory models and 214, 216
    - predefined macro 152
    - routines, calling via function pointers 217
  - overloaded constructors *See* constructors, overloaded
  - overloaded functions
    - defined 104
  - overloaded operators 20, 75, 124-128
    - >> (get from) 175
      - complex numbers and 208
      - put and write functions and 170
    - << (put to) 168, 174
      - complex numbers and 208
      - put and write functions and 170
    - assignment (=) 127
    - binary 127
    - brackets 128
    - complex numbers and 208
    - creating 105
    - defined 104
    - delete 125
    - functions and 75
    - inheritance and 125
    - new 102, 125
    - operator functions and 124, 125
    - operator keyword and 124
    - parentheses 128
    - precedence and 75, 169
    - selection (->) 128
    - unary 127
  - \_ovrbuffer (global variable) 213, 217
- ## P
- p TCC option (Pascal calling convention) 50, 150, 152, 250
    - cdecl and 50
  - pages
    - active
      - defined 236
      - setting 236
    - buffers 236
    - visual
      - defined 236
      - setting 236
  - painting *See* graphics, fill, operations
  - palettes *See* graphics, palettes
  - paragraphs *See* memory, paragraphs

- parameterized manipulators *See* manipulators
- parameters *See also* arguments
  - arguments vs. 4
  - default
    - constructors 116
  - ellipsis and 25
  - empty lists 39
  - fixed 61
  - formal 62
    - C++ 63
    - scope 63
  - function calls and 63
  - order on stack 248
  - passing
    - C 47, 49, 248
    - Pascal vs. 247-250
    - Pascal 47, 49, 249, 258, 263
  - variable 61
- parentheses 23
  - as function call operators 78
  - macros and 24
  - nested
    - macros and 139
  - overloading 128
- parity flag 189
- parsing 5
- Pascal
  - calling conventions
    - C calling convention vs. 250
    - command-line compiler option (-p) 50, 250
    - prototypes and 250
    - reasons for using 250
  - functions 49
  - identifiers 49
    - assembly language and 253
    - case sensitivity and 10
  - parameter-passing sequence 47, 249, 258, 263
    - assembly language and 263
  - variant record types 70
  - \_\_PASCAL\_\_ macro 152
- pascal (keyword) 47, 49
  - function modifiers and 51
  - preserving case while using 50
- pass-by-address, pass-by-value, and pass-by-var
  - See* parameters; referencing and dereferencing
- period as an operator *See* operators, selection (. and ->)
- perror (function)
  - messages generated by 338
- phrase structure grammar *See* grammar
- pointers 53, *See also* referencing and dereferencing
  - advancing 57
  - arithmetic 57, 193
  - assignments 55
  - base class
    - destructors and 123
  - C++ 98
    - inserter 169
    - reference declarations 58
  - casting to integer 332
  - changing memory models and 202
  - to class members 23
  - comparing 85, 87, 93, 193
    - while loops 95
  - const 48
  - constants and 56
  - conversions *See* conversions
  - declarations 55
  - declarator (\*) 25
  - default data 198
  - delete operator and 122
  - dereference 23
  - far 47
    - adding values to 193
    - comparing 192
    - declaring 201-202
    - function prototypes and 202
    - memory model size and 201
    - registers and 192
  - far memory model and 192
  - function 54
    - C++ 54
    - modifying 51
    - object pointers vs. 53
    - void 54
  - generic 39, 55
  - huge 47, 193

- comparing
  - != operator 193
  - == operator 193
- declaring 201-202
- overhead of 194
- huge memory model and 192
- initializing 55
- integer type for 332
- keywords for 47
- manipulating 192
- memory models and 192, 200
- to memory addresses 202
- modifiers 51, 199, 200
- modifying 51
- near 47, *See also* segments, pointers
  - declaring 201-202
  - function prototypes and 202
  - memory model size and 201
  - registers and 192
- near memory model and 192
- normalized 193
- null 55
  - NULL macro and 335
- operator (→)
  - overloading 128
  - structure and union access 23, 66, 78
- overlays and 217
- pointers to 54
- range 19
- reassigning 55
- referencing and dereferencing 80
- segment 47, 199
- stack 189
- structure members as 65
- typecasting 58
- void 55
- portable code
  - bit fields and 70
- positive offsets 189
- postdecrement operator (--) 21, 79
- postfix operators 77
- postincrement operator (++) 21, 78
- #pragma exit
  - destructors and 122
- #pragma directives 146
  - argsused 146
  - exit 146
  - inline 147, 268
  - option pragma 148
  - saveregs 149
  - startup 146
  - warn 150
- precedence 76, *See also* associativity
  - controlling 23
  - expressions 75
    - floating point 76
    - integer 76
  - overloading and 169
  - operators 75
- predecrement operator (--) 21, 79
- predefined macros *See* macros, predefined
- prefix opcodes, repeat 272
- prefix operators 77
- preincrement operator (++) 21, 79
- preprocessor directives *See* directives
- printers
  - printing direction 329
- printf (function)
  - C++ streams and 166, 169
- private (keyword)
  - data members and member functions 108
  - derived classes and 110
  - unions and 71
- PROC statement (assembler)
  - extension in TASM 2.0 261
- procedures *See* functions
- process control
  - functions (list) 163
- process.h (header file) 156
- profilers 217
- programs
  - creating 4
  - mixed C and C++ 179
  - performance
    - improving 45
  - size
    - reducing 45
  - terminate and stay resident
    - interrupt handlers and 276
  - very large
    - overlying 211
- projects
  - files
    - graphics library listed in 231

- promotions *See* conversions
- protected (keyword)
  - base classes and 110
  - data members and member functions 109
  - derived classes and 110
  - unions and 71
- prototypes 60-61
  - arguments and
    - matching number of 64
  - C++ 59
  - ellipsis and 61, 64
  - examples 60, 61
  - far and near pointers and 202
  - function calls and 63
  - function definitions and
    - not matching 64
  - graphics functions
    - header file 155
  - header files and 61, 153
  - libraries and 64
  - mixing modules and 203
  - new operator and 101
  - Pascal calling convention and 250
  - scope *See* scope
- pseudovariables
  - accumulator 266
  - addresses 266
  - base pointer 266
  - counting and loops 266
  - defined 265
  - holding data 266
  - indexing 266
  - memory addresses and 266
  - register 9
  - registers and 265
  - stack pointer 266
- public (keyword)
  - data members and member functions 108
  - derived classes and 110
  - unions and 71
- PUBLIC statement (assembler)
  - extension in TASM 2.0 261
- punctuators 23, 23-26
- pure (keyword)
  - virtual functions and 130
- pure specifier 37

- put (function)
  - << and >> overloaded operators and 170
  - C++ output and 170
- put to operator (<<) *See* overloaded operators, >> (put to)
- putback (member function) 177
- putenv (function)
  - environment names and methods 339

## Q

- qualified names 108
- question mark, displaying 15
- question mark colon conditional operator 23, 89
- quotes, displaying 15
- QWORD (assembler) 255

## R

- \r (carriage return character) 15
- r TCC option (register variables) 261
- raise (function)
  - header file 156
- RAM
  - Turbo C++'s use of 211
- ranges
  - floating-point constants 16
- rdstate (member function) 182
- read (function)
  - C++ input and 177
- realloc (function)
  - zero-size memory allocation and 338
- records *See* structures
- recursive functions
  - memory models and 200
- reference declarations 58
- references
  - forward 27
- referencing and dereferencing 80, *See also*
  - pointers
    - asterisk and 25
    - C++ 98
    - functions 99
    - simple 99
    - pointers 23
  - referencing data in inline assembly code 272
  - referencing declarations *See* declarations
  - register (keyword) 45

- class members and 104
- external declarations and 36
- formal parameters and 63
- local duration and 31
- registers
  - 8086 188-190
  - 80x87 top-of-stack 257
  - AH 266
  - AL 266
  - assembly language and 266
  - AX 188
    - assembly language and 257, 266
  - base point 189
  - BH 266
  - BL 266
  - BP 189
    - assembly language and 257, 266
    - overlays and 217
  - BX 188
    - assembly language and 266
  - CH 266
  - CL 266
  - conventions 261
  - CS 190, 192
    - assembly language and 266
  - CX 188
    - assembly language and 266
  - defined 265
  - DH 266
  - DI 189
    - assembly language and 261, 266, 273
  - DL 266
  - DS 190, 192
    - assembly language and 266
    - \_loads and 52
  - DX 188
    - assembly language and 257, 266
  - ES 190
    - assembly language and 266
  - flags 187, 189
  - hardware
    - bit fields and 69
  - index 188, 189
  - interrupts and 48
  - IP (instruction pointer) 187
  - LOOP and string instruction 188
  - math operations 188
  - numeric coprocessors and 207
  - objects and 332
  - pseudovariables 9, 265
  - return values and 257
  - saving with huge functions 149
  - segment 189, 190
  - setting for low-level programming 265
  - SI 189
    - assembly language and 261, 266, 273
  - SP 189
    - assembly language and 257, 266
  - special-purpose 189
  - SS 190
    - assembly language and 266
  - values
    - preserving 52
    - variable declarations and 45
    - variables 45
      - in inline assembly code 273
- relational operators *See* operators, relational
- remainder operator (%) 22, 82
- remove (function)
  - open files and 337
- rename (function)
  - preexisting file name and 338
- repeat prefix opcodes 272
- resetiosflags (manipulator) 173
- resolution *See* screens, resolution
- return
  - statements
    - functions and 97
  - types 60
  - values
    - assembly language and 257
- rounding
  - banker's 211
  - direction
    - division 82
  - errors 209
  - rules 331
- routines, assembly language *See* assembly language, routines
- run-time library
  - functions by category 157
  - source code, licensing 154
- rvalues 28, *See also* lvalues

## S

- S TCC option (produce .ASM output file) 252
- saveregs pragma 149
- \_saveregs (keyword) 47, 51, 52
- scalar data types *See* data types
- scaling factor
  - graphics 236
- scanf (function)
  - >> operator and 175
  - C++ streams and 166
- scope 29-30, *See also* visibility
  - block 29
  - block statements and 92
  - C++ 30, 131-133
    - hiding 132
    - operator (::) 23, 100
    - rules 132
  - classes 29
    - names 103
  - enumerations 30
    - C++ 73
  - file 29
    - static storage class specifier and 33
  - formal parameters 63
  - function 29
    - prototype 29
  - global 29
  - goto and 29
  - identifiers and 10
  - local
    - local duration and 31
  - members 107-109
  - name spaces and 29
  - pointers 55
  - storage class specifiers and 45-46
  - structures 30
  - unions 30
  - variables 30
  - visibility and 30
- screens *See also* graphics; text; windows
  - aspect ratio 236
  - attributes, controlling 225
  - cells
    - attributes 229
    - blinking 230
    - characters in 221
    - colors 229
    - clearing 236
    - colors 229, 239
    - coordinates 223
      - starting positions 222
    - cursor
      - changing 227
      - manipulating 224
    - modes
      - controlling 225
      - defining 221
      - graphics 222, 230, 232, 233
      - selecting 233
      - text 221, 228, 233
    - resolution 222, *See also* graphics, pixels
    - viewports *See* graphics
  - searches
    - header file 156
    - #include directive algorithm 141
  - seekg (member function)
    - current "get" position 181
  - seekp (member function)
    - current "put" position 181
  - \_seg (keyword) 47, 199, 200
  - segment:offset address notation 191
    - making far pointers from 202
  - segmented memory architecture 190
  - segments 191, 194
    - component of a pointer 202
    - directives 251, 262
    - memory 190
    - pointers 47, 199
    - registers 189, 190
  - selection
    - operators *See* operators, selection
    - statements *See* if statements; switch statements
  - semicolons 24, 93
  - setbase (manipulator) 173
  - setbkcolor (function)
    - CGA vs. EGA 242
  - setf (function) 174
  - setfill (manipulator) 173
  - setiosflags (manipulator) 173
  - setjmp (function)
    - header file 156
  - setjmp.h (header file) 156
  - setprecision (manipulator) 173

- setw (manipulator) 173
  - field width and 177
- shapes *See* graphics
- share.h (header file) 156
- shift bits operators (<< and >>) 22, 83
- short integers *See* integers, short
- \_SI pseudovisible 266
- SI register 189
  - assembly language and 261, 266
- side effects
  - macro calls and 140
- sign 11
  - extending 15
    - conversions and 42
  - flag 189
  - integer constants and 12
- signal (function) 336
  - header file 156
  - signal set 336
  - signals 336
- signal.h (header file) 156
- signed (keyword) 40
- simplified segment directives 251, 262
- single quote character
  - displaying 15
- size overrides in inline assembly code 273
- size\_t (data type) 82, 125, 126
- sizeof (operator) 81
  - arrays and 82
  - classes and 82
  - data type 334
  - example 29
  - function-type expressions and 82
  - functions and 82
  - preprocessor directives and 82
  - unions and 70
- \_\_SMALL\_\_ macro 150
- small code
  - data
    - and memory models *See* memory models
- software interrupt instruction 275
- sounds
  - beep 276
  - functions (list) 163
- source code 4
  - run-time library
    - licensing 154
- \_SP pseudovisible 266
- SP register 189
  - assembly language and 257, 266
- special-purpose registers (8086) 189
- specifiers *See* type specifiers
- splicing lines 5, 18
- \_SS pseudovisible 266
- SS register 190
  - assembly language and 266
- \_ss (keyword) 47, 199
- stack
  - pointers 189
  - segment 190
  - top of
    - register 257
- standard conversions *See* conversions
- startup pragma 146
- state queries 243-245
- statements 91-98, *See also* individual statement names
  - assembly language 92
  - block 92
    - marking start and end 24
  - default 94
  - do while *See* loops, do while
  - expression 24, 93
  - for *See* loops, for
  - if *See* if statements
  - iteration *See* loops
  - jump *See* break statements; continue statements; goto statements; return statements
  - labeled 92
  - null 93
  - syntax 92
  - while *See* loops, while
- static
  - duration 31
  - functions 32
  - members *See* data members, static; member functions, static
  - objects *See* objects, static
  - variables *See* variables, static
- static (keyword) 45
  - linkage and 32
- \_status87 (function)
  - floating point exceptions and 207

- stdarg.h (header file)
  - user-defined functions and 61
- stdargs.h (header file) 156
- \_\_STDC\_\_ macro 152
  - #define and #undef directives and 137
- stddef.h (header file) 156
- stderr (header file) 156
- stdin (header file) 156
- stdio.h (header file) 156
- stdiobuf (class)
  - base class of 179
- stdiostr.h (header file) 156
- stdlib.h (header file) 156
- stdout (header file) 156
- stdprn (header file) 156
- storage class
  - identifiers and 28
  - specifiers 45
    - functions and 33
    - linkage and 32
    - register
      - objects and 332
    - static
      - file scope and 33
- stream.h (header file) 157
- streambuf (class) 167
  - derived classes of 167, 179
  - old vs. new streams 184
- streams
  - bidirectional operations 167
  - binary
    - null characters and 337
  - C++
    - accessing 166
    - assigning to streams 184
    - constructors 178
    - data types 169
    - differences between versions 184
    - errors 179, 181
      - member functions for testing 182
    - flushing 173
    - header file 157
    - initializing 178
    - input
      - chaining operations 175
      - manipulators and *See* manipulators
      - old vs. new 184
      - standard 168
      - stdio.h and 184
      - upgrading from 1.x 184
      - using 165-183
    - C vs. C++ 184
    - classes 166
      - streambuf 167
    - clearing 173
    - compatibility 165
    - defined 166
    - header file 156
    - libraries 165
    - memory and 179
    - text
      - newline character and 336, 337
  - strerror (function)
    - messages generated by 339
  - string.h (header file) 157
  - strings
    - clipping 238
    - concatenating 18
    - continuing across line boundaries 18
    - converting arguments to 139
    - functions (list) 161
    - header file 157
    - inserting terminal null into 173
    - instructions
      - registers 188
    - literal 5, 17
    - scanning
      - while loops and 95
  - stroked fonts *See* fonts
  - strstrea.h (header file) 157
  - strstream (class)
    - base class of 179
  - strstreambuf (class)
    - base class of 167
  - struct (keyword) 64, *See also* structures
    - C++ and 65, 103
  - structures 64-70
    - access
      - C++ 110
    - bit fields *See* bit fields
    - C++ 102
      - C vs. 103
    - complex 208
    - declaring 64

- functions and 66
- incomplete declarations of 68
- indeterminate arrays and 59
- initializing 43
- member functions and 65
- members
  - access 66, 78, 109
  - as pointers 65
  - C++ 65
  - comparing 85
  - declaring 65
  - in inline assembly code 273
    - restrictions 274
  - names 68
  - padding and alignment 332
- memory allocation 67
- name spaces 30, 68
- tags 64, 65
  - typedefs and 65
- typedefs and 65
- unions vs. 70
- untagged 65
  - typedefs and 65
- within structures 65
- word alignment
  - memory and 67
- subscripting operator *See* brackets
- subscripts for arrays 23, 77
  - overloading 128
- subtraction operator (-) 22, 83
- switch statements 94
  - case statement and
    - duplicate case constants 94
  - case values
    - number of allowed 333
  - default label and 94
- symbolic constants *See* constants, symbolic
- syntax
  - assembly language statements 92
  - classes 102
  - declarations 33, 34
  - declarator 53
  - directives 134
  - expressions 74
  - manipulators 173
  - notation 4
  - statements 92

- sys\stat.h (header file) 157
- sys\types.h (header file) 157
- system (function) 339
- system control, graphics 231

## T

- \t (horizontal tab character) 15
- tags
  - enumerations 72
    - name spaces 73
  - structure *See* structures, tags
- TASM *See* Turbo Assembler
- taxonomy
  - types 38
- TBYTE (assembler) 255
- tellg (function)
  - current "get" position 181
- tellp (function)
  - current "put" position 181
- template, assembly language 252
- temporary objects 100
- tentative
  - definitions 34
- terminate and stay resident programs
  - interrupt handlers and 276
- text
  - blocks
    - moving in and out of memory 224
  - capturing to memory 225
  - colors 229
  - in graphics mode 237
  - information on current settings 245
  - justifying 238
  - manipulation
    - functions 224
    - onscreen 224
    - output and 224
  - mode types 228
  - output
    - header file 223
  - reading and writing 224
  - scrolling 224
  - streams
    - writing
      - truncation and 337
  - strings
    - clipping 238

- size 238
- writing to screen 225
- this (keyword)
  - nonstatic member functions and 105
  - static member functions and 106
- time
  - formats 340
  - functions (list) 164
  - local
    - how defined 339
  - macro 152
  - \_\_TIME\_\_ macro 152
    - availability 334
    - #define and #undef directives and 137
- time.h (header file) 157
- \_\_TINY\_\_ macro 150
- tiny memory model *See* memory models
- TLINK (linker)
  - using directly 203
- tokens 4
  - continuing long lines of 140
  - kinds of 7
  - parsing 5
  - pasting 6, 139
  - replacement 135
  - replacing and merging 26
- top of stack (TOS) register
  - assembly language and 257
- translation units 32
- trap flag 189
- truth table
  - bitwise operators 87
- Turbo Assembler 268
  - language specifier and
    - example 262
  - language specifiers and 254, 255
    - example 256
  - /mx option (case sensitivity) 254
  - PUBLIC statement extension 261
  - referencing function parameters and 257
  - simplified segment directives 251, 262
- Turbo C++
  - ANSI implementation-specific items 327-340
  - functions
    - licensing 154
  - keywords
    - using as identifiers 3

- version number 152
- Turbo Profiler 217
- \_\_TURBOC\_\_ macro 152
- type specifiers
  - elaborated 104
  - pure 37
- type taxonomy 38
- typecasting
  - pointers 58
- typed constants *See* constants
- typedef (keyword) 46
  - name space 30
  - structure tags and 65
  - structures and 65
- typedefs
  - untagged structures and 65
- types *See* data types

## U

- U TCC option (undefine) 137
- u TCC option (underscores) 253
- UINT\_MAX (constant) 84
- ULONG\_MAX (constant) 84
- unary operators 21
  - minus (-) 21, 81
  - plus (+) 21, 81
  - syntax 79
- #undef directive 136
  - global identifiers and 137
- underbars *See* underscores
- underflow range errors
  - mathematics functions and 335
- underscores
  - generating 47
  - ignoring 47
  - leading, in assembly language routines 253
- union (keyword)
  - C++ 103
- unions 70
  - accessing 331
  - anonymous
    - member functions and 71
  - base classes and 110
  - bit fields and *See* bit fields
  - C++ 71, 102
    - C vs. 103
  - classes and 71

- constructors and destructors and 114
- declarations 71
- initialization 43, 71
- members
  - access 78, 109
  - name space 30
  - sizeof and 70
  - structures vs. 70
- units, translation *See* translation units
- UNIX
  - constants
    - header file 157
- unsetf (function) 174
- unsigned (keyword) 40
- untagged structures *See* structures, untagged
- UTIL.DOC 239

## V

- \v (vertical tab character) 15
- value, passing by *See* parameters
- values
  - comparing 84
- values.h (header file) 157
- var, passing by *See* parameters
- variable number of arguments 25
- variables
  - automatic *See* auto (keyword)
  - declaring 44
  - defining in assembly code routines 253
  - external 45
  - global *See* global variables
  - initializing 44
  - internal linkage 45
  - name space 30
  - offsets in inline assembly code 273
  - pseudo *See* pseudovariables
  - register *See* registers, variables
  - volatile 49
- variant record types *See* unions
- vectors, interrupt *See* interrupts
- version number
  - Turbo C++ 152
- vertical tab 15
- video
  - adapters
    - graphics, compatible with Turbo C++ 232

- video, adapters
  - graphics, compatible with Turbo C++ 232
- video adapters *See also* Color/Graphics Adapter (CGA); Enhanced Graphics Adapter (EGA); graphics drivers; Video Graphics Array Adapter (VGA)
  - modes 221
  - output
    - directing 230
    - using 221-245
- Video Graphics Array Adapter (VGA) *See also* graphics drivers; video adapters
  - color control 242
- viewports *See* graphics
- virtual
  - base classes *See* classes, base, virtual
  - destructors *See* destructors, virtual
  - functions *See* C++, functions, virtual
- virtual (keyword)
  - constructors and destructors and 114
  - functions and 129
- visibility 30, *See also* scope
  - C++ 30
  - pointers 55
  - scope and 30
- visual page
  - defined 236
  - setting 236
- void (keyword) 39
  - function pointers and 54
  - functions and 61
  - interrupt functions and 48
  - pointers 55
  - typecasting expressions and 39
- volatile (keyword) 47, 48
  - formal parameters and 63
- VROOMM 212

## W

- warn pragma 150
- warnings
  - defined 283
  - disabling 146
  - list 318-326
  - overriding 150
  - pragma warn and 150
- wchar\_t (wide character constants) 15, 330

- arrays and 44
- while loops *See* loops, while
- whitespace 5
  - comments and 7
  - comments as 5
  - extracting 173
- wide character constants (wchar\_t) 15, 330
- width functions (C++) 171
- window (function)
  - default window and 222
  - example 227
- windows
  - active
    - erasing 224
    - controlling 225
    - creating 225
    - default type 222
    - defined 222
    - functions (list) 163
    - managing
      - header file 223
    - output in 225
    - scrolling 224
  - text
    - creating 227
    - default size 227
- withassign classes 167, 184
- WORD (assembler) 255

- word alignment 67, 332, 333
  - memory and
  - structures 67
- write (function)
  - << and >> overloaded operators and 170
- write (functions)
  - C++ output and 170
- ws (manipulator) 173
- \_wscroll (global variable) 224
- wxxx options (warnings)
  - warn pragma and 150

## X

- \xH (display a string of hexadecimal digits) 15
- XOR operator (^) 22, 87
  - truth table 87

## Y

- Y TCC option (compiler generated code for overlays) 216
- Y TCC option (overlays) 152
- Yo option (overlays) 214

## Z

- zero flag 189
- zero-length files 337

PROGRAMMER'S  
GUIDE

# TURBO C<sup>®</sup>++

**B O R L A N D**

1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001, (408) 438-5300 ■ PART # 14MN-CPP04-10 ■ BOR 1509  
UNIT 8 PAVILIONS, RUSCOMBE BUSINESS PARK, TWYFORD, BERKSHIRE RG10 9NN, ENGLAND  
43 AVENUE DE L'EUROPE—BP 6, 78141 VELIZY VILLACOUBLAY CEDEX FRANCE