Appendices

Appendix A Reserved Words

The following identifiers are *reserved words*—identifiers with predefined meanings in the C++ language. The programmer cannot declare them for other uses (for example, variable names) in a C++ program.

and	double	not	this
and_eq	dynamic_cast	not_eq	throw
asm	else	operator	true
auto	enum	or	try
bitand	explicit	or_eq	typedef
bitor	export	private	typeid
bool	extern	protected	typename
break	false	public	union
case	float	register	unsigned
catch	for	reinterpret_cast	using
char	friend	return	virtual
class	goto	short	void
compl	if	signed	volatile
const	inline	sizeof	wchar_t
const_cast	int	static	while
continue	long	static_cast	xor
default	mutable	struct	xor_eq
delete	namespace	switch	
do	new	template	

71560_APPX_DaleNEW.indd 985 6/12/09 2:22:12 PM

Appendix B Operator Precedence

The following table summarizes C++ operator precedence. Several operators are not discussed in this book (`typeid`, the comma operator, `->*`, and `.*`, for instance). For information on these operators, see Stroustrup's *The C++ Programming Language*, Third Edition (Addison-Wesley, 1997).

In the table, the operators are grouped by precedence level (highest to lowest), and a horizontal line separates each precedence level from the next-lower level.

In general, the binary operators group from left to right; the unary operators, from right to left; and the `?:` operator, from right to left. Exception: The assignment operators group from right to left.

<table>
<tr><th colspan="3">Precedence (highest to lowest)</th></tr>
<tr><th>Operator</th><th>Associativity</th><th>Remarks</th></tr>
<tr><td>::</td><td>Left to right</td><td>Scope resolution (binary)</td></tr>
<tr><td>::</td><td>Right to left</td><td>Global access (unary)</td></tr>
<tr><td>()</td><td>Left to right</td><td>Function call and function-style cast</td></tr>
<tr><td>[] -> .</td><td>Left to right</td><td></td></tr>
<tr><td>++ --</td><td>Right to left</td><td>++ and -- as postfix operators</td></tr>
<tr><td>typeid dynamic_cast</td><td>Right to left</td><td></td></tr>
<tr><td>static_cast const_cast</td><td>Right to left</td><td></td></tr>
<tr><td>reinterpret_cast</td><td>Right to left</td><td></td></tr>
<tr><td>++ -- ! Unary + Unary –</td><td>Right to left</td><td>++ and -- as prefix operators</td></tr>
<tr><td>~ Unary * Unary &</td><td>Right to left</td><td></td></tr>
<tr><td>(cast) sizeof new delete</td><td>Right to left</td><td></td></tr>
<tr><td>->* .*</td><td>Left to right</td><td></td></tr>
<tr><td>* / %</td><td>Left to right</td><td></td></tr>
<tr><td>+ –</td><td>Left to right</td><td></td></tr>
<tr><td><< >></td><td>Left to right</td><td></td></tr>
<tr><td>< <= > >=</td><td>Left to right</td><td></td></tr>
<tr><td>== !=</td><td>Left to right</td><td></td></tr>
<tr><td>&</td><td>Left to right</td><td></td></tr>
<tr><td>^</td><td>Left to right</td><td></td></tr>
<tr><td>|</td><td>Left to right</td><td></td></tr>
<tr><td>&&</td><td>Left to right</td><td></td></tr>
<tr><td>||</td><td>Left to right</td><td></td></tr>
<tr><td>?:</td><td>Right to left</td><td></td></tr>
<tr><td>= += -= *= /= %=</td><td>Right to left</td><td></td></tr>
<tr><td><<= >>= &= |= ^=</td><td>Right to left</td><td></td></tr>
<tr><td>throw</td><td>Right to left</td><td></td></tr>
<tr><td>,</td><td>Left to right</td><td>The sequencing operator, not the separator</td></tr>
</table>

71560_APPX_DaleNEW.indd 986 6/12/09 2:22:12 PM

Appendix C A Selection of Standard Library Routines

The C++ standard library provides a wealth of data types, functions, and named constants. This appendix details only some of the more widely used library facilities. It is a good idea to consult the manual for your particular system to see what other types, functions, and constants the standard library provides.

This appendix is organized alphabetically according to the header file your program must **#include** before accessing the listed items. For example, to use a mathematics routine such as **sqrt**, you would **#include** the header file **cmath** as follows:

```
#include <cmath>
using namespace std;
  ⋮
y = sqrt(x);
```

Note that every identifier in the standard library is defined to be in the namespace **std**. Without the **using** directive above, you would write

```
y = std::sqrt(x);
```

C.1 The Header File cassert

assert(booleanExpr)

Argument: A logical (Boolean) expression

Effect: If the value of **booleanExpr** is **true**, execution of the program simply continues. If the value of **booleanExpr** is **false**, execution terminates immediately with a message stating the Boolean expression, the name of the file containing the source code, and the line number in the source code.

Function return value: None (a void function)

Note: If the preprocessor directive **#define NDEBUG** is placed before the directive **#include <cassert>**, all **assert** statements are ignored.

C.2 The Header File cctype

isalnum(ch)

Argument: A **char** value **ch**

Function return value: An **int** value that is

- nonzero (**true**), if **ch** is a letter or a digit character (‘A’–‘Z’, ‘a’–‘z’, ‘0’–‘9’)
- 0 (**false**), otherwise

isalpha(ch)

Argument: A **char** value **ch**

Function return value: An **int** value that is

- nonzero (**true**), if **ch** is a letter (‘A’–‘Z’, ‘a’–‘z’)
- 0 (**false**), otherwise

`iscntrl(ch)`
Argument: A `char` value `ch`
Function return value: An `int` value that is
- nonzero (`true`), if `ch` is a control character (in ASCII, a character with the value 0–31 or 127)
- 0 (`false`), otherwise

`isdigit(ch)`
Argument: A `char` value `ch`
Function return value: An `int` value that is
- nonzero (`true`), if `ch` is a digit character ('0'–'9')
- 0 (`false`), otherwise

`isgraph(ch)`
Argument: A `char` value `ch`
Function return value: An `int` value that is
- nonzero (`true`), if `ch` is a nonblank printable character (in ASCII, '!' through '~')
- 0 (`false`), otherwise

`islower(ch)`
Argument: A `char` value `ch`
Function return value: An `int` value that is
- nonzero (`true`), if `ch` is a lowercase letter ('a'–'z')
- 0 (`false`), otherwise

`isprint(ch)`
Argument: A `char` value `ch`
Function return value: An `int` value that is
- nonzero (`true`), if `ch` is a printable character, including the blank (in ASCII, ' ' through '~')
- 0 (`false`), otherwise

`ispunct(ch)`
Argument: A `char` value `ch`
Function return value: An `int` value that is
- nonzero (`true`), if `ch` is a punctuation character (equivalent to `isgraph(ch) && !isalnum(ch)`)
- 0 (`false`), otherwise

`isspace(ch)`
Argument: A `char` value `ch`
Function return value: An `int` value that is
- nonzero (`true`), if `ch` is a whitespace character (blank, newline, tab, carriage return, form feed)
- 0 (`false`), otherwise

`isupper(ch)`
Argument: A `char` value `ch`
Function return value: An `int` value that is
- nonzero (`true`), if `ch` is an uppercase letter ('A'–'Z')
- 0 (`false`), otherwise

`isxdigit(ch)`

Argument:	A `char` value `ch`
Function return value:	An `int` value that is ▪ nonzero (`true`), if `ch` is a hexadecimal digit ('0'–'9', 'A'–'F', 'a'–'f ') ▪ 0 (`false`), otherwise

`tolower(ch)`

Argument:	A `char` value `ch`
Function return value:	A character that is ▪ the lowercase equivalent of `ch`, if `ch` is an uppercase letter ▪ `ch`, otherwise

`toupper(ch)`

Argument:	A `char` value `ch`
Function return value:	A character that is ▪ the uppercase equivalent of `ch`, if `ch` is a lowercase letter ▪ `ch`, otherwise

C.3 The Header File `cfloat`

This header file supplies named constants that define the characteristics of floating-point numbers on your particular machine. Among these constants are the following:

`FLT_DIG`	Approximate number of significant digits in a `float` value on your machine
`FLT_MAX`	Maximum positive `float` value on your machine
`FLT_MIN`	Minimum positive `float` value on your machine
`DBL_DIG`	Approximate number of significant digits in a `double` value on your machine
`DBL_MAX`	Maximum positive `double` value on your machine
`DBL_MIN`	Minimum positive `double` value on your machine
`LDBL_DIG`	Approximate number of significant digits in a `long double` value on your machine
`LDBL_MAX`	Maximum positive `long double` value on your machine
`LDBL_MIN`	Minimum positive `long double` value on your machine

C.4 The Header File `climits`

This header file supplies named constants that define the limits of integer values on your particular machine. Among these constants are the following:

`CHAR_BITS`	Number of bits in a byte on your machine (8, for example)
`CHAR_MAX`	Maximum `char` value on your machine
`CHAR_MIN`	Minimum `char` value on your machine
`SHRT_MAX`	Maximum `short` value on your machine
`SHRT_MIN`	Minimum `short` value on your machine
`INT_MAX`	Maximum `int` value on your machine
`INT_MIN`	Minimum `int` value on your machine
`LONG_MAX`	Maximum `long` value on your machine
`LONG_MIN`	Minimum `long` value on your machine
`UCHAR_MAX`	Maximum `unsigned char` value on your machine
`USHRT_MAX`	Maximum `unsigned short` value on your machine
`UINT_MAX`	Maximum `unsigned int` value on your machine
`ULONG_MAX`	Maximum `unsigned long` value on your machine

C.5 The Header File `cmath`

In the **`math`** routines listed below, the following notes apply.

1. Error handling for incalculable or out-of-range results is system dependent.
2. All arguments and function return values are technically of type **`double`** (double-precision floating-point). However, single-precision (**`float`**) values may be passed to the functions.

`acos(x)`
Argument: A floating-point expression **`x`**, where $-1.0 \leq$ **`x`** ≤ 1.0
Function return value: Arc cosine of **`x`**, in the range 0.0 through π

`asin(x)`
Argument: A floating-point expression **`x`**, where $-1.0 \leq$ **`x`** ≤ 1.0
Function return value: Arc sine of **`x`**, in the range $-\pi/2$ through $\pi/2$

`atan(x)`
Argument: A floating-point expression **`x`**
Function return value: Arc tangent of **`x`**, in the range $-\pi/2$ through $\pi/2$

`ceil(x)`
Argument: A floating-point expression **`x`**
Function return value: "Ceiling" of **`x`** (the smallest whole number $\geq$ **`x`**)

`cos(angle)`
Argument: A floating-point expression **`angle`**, measured in radians
Function return value: Trigonometric cosine of **`angle`**

`cosh(x)`
Argument: A floating-point expression **`x`**
Function return value: Hyperbolic cosine of **`x`**

`exp(x)`
Argument: A floating-point expression **`x`**
Function return value: The value *e* (2.718...) raised to the power **`x`**

`fabs(x)`
Argument: A floating-point expression **`x`**
Function return value: Absolute value of **`x`**

`floor(x)`
Argument: A floating-point expression **`x`**
Function return value: "Floor" of **`x`** (the largest whole number $\leq$ **`x`**)

`log(x)`
Argument: A floating-point expression **`x`**, where **`x`** > 0.0
Function return value: Natural logarithm (base *e*) of **`x`**

`log10(x)`
Argument: A floating-point expression **`x`**, where **`x`** > 0.0
Function return value: Common logarithm (base 10) of **`x`**

`pow(x, y)`
Arguments: Floating-point expressions `x` and `y`. If `x` = 0.0, `y` must be positive; if `x` ≤ 0.0, `y` must be a whole number
Function return value: `x` raised to the power `y`

`sin(angle)`
Argument: A floating-point expression `angle`, measured in radians
Function return value: Trigonometric sine of `angle`

`sinh(x)`
Argument: A floating-point expression `x`
Function return value: Hyperbolic sine of `x`

`sqrt(x)`
Argument: A floating-point expression `x`, where `x` ≥ 0.0
Function return value: Square root of `x`

`tan(angle)`
Argument: A floating-point expression `angle`, measured in radians
Function return value: Trigonometric tangent of `angle`

`tanh(x)`
Argument: A floating-point expression `x`
Function return value: Hyperbolic tangent of `x`

C.6 The Header File `cstddef`

This header file defines a few system-dependent constants and data types. From this header file, the only item we use in this book is the following symbolic constant:

`NULL` The null pointer constant `0`

C.7 The Header File `cstdlib`

`abs(i)`
Argument: An `int` expression `i`
Function return value: An `int` value that is the absolute value of `i`

`atof(str)`
Argument: A C string (null-terminated `char` array) `str` representing a floating point number, possibly preceded by whitespace characters and a '+' or '–'
Function return value: A `double` value that is the floating-point equivalent of the characters in `str`
Note: Conversion stops at the first character in `str` that is inappropriate for a floating-point number. If no appropriate characters were found, the return value is system dependent.

atoi(str)

Argument:	A C string (null-terminated `char` array) `str` representing an integer number, possibly preceded by whitespace characters and a '+' or '–'
Function return value:	An `int` value that is the integer equivalent of the characters in `str`
Note:	Conversion stops at the first character in `str` that is inappropriate for an integer number. If no appropriate characters were found, the return value is system dependent.

atol(str)

Argument:	A C string (null-terminated `char` array) `str` representing a long integer, possibly preceded by whitespace characters and a '+' or '–'
Function return value:	A `long` value that is the long integer equivalent of the characters in `str`
Note:	Conversion stops at the first character in `str` that is inappropriate for a `long` integer number. If no appropriate characters were found, the return value is system dependent.

exit(exitStatus)

Argument:	An `int` expression `exitStatus`
Effect:	Program execution terminates immediately with all files properly closed
Function return value:	None (a void function)
Note:	By convention, `exitStatus` is 0 to indicate normal program completion and is nonzero to indicate an abnormal termination.

labs(i)

Argument:	A `long` expression `i`
Function return value:	A `long` value that is the absolute value of `i`

rand()

Argument:	None
Function return value:	A random `int` value in the range 0 through `RAND_MAX`, a constant defined in `cstdlib` (`RAND_MAX` is usually the same as `INT_MAX`)
Note:	See `srand` below.

srand(seed)

Argument:	An `int` expression `seed`, where `seed` ≥ 0
Effect:	Using `seed`, the random number generator is initialized in preparation for subsequent calls to the `rand` function.
Function return value:	None (a void function)
Note:	If `srand` is not called before the first call to `rand`, a `seed` value of 1 is assumed.

`system(str)`

Argument:	A C string (null-terminated `char` array) `str` representing an operating system command, exactly as it would be typed by a user on the operating system command line
Effect:	The operating system command represented by `str` is executed.
Function return value:	An `int` value that is system dependent
Note:	Programmers often ignore the function return value, using the syntax of a void function call rather than a value-returning function call.

C.8 The Header File `cstring`

The header file `cstring` (not to be confused with the header file named `string`) supports manipulation of C strings (null-terminated `char` arrays).

`strcat(toStr, fromStr)`

Arguments:	C strings (null-terminated `char` arrays) `toStr` and `fromStr`, where `toStr` must be large enough to hold the result
Effect:	`fromStr`, including the null character '\0', is concatenated (joined) to the end of `toStr`.
Function return value:	The base address of `toStr`
Note:	Programmers usually ignore the function return value, using the syntax of a void function call rather than a value-returning function call.

`strcmp(str1, str2)`

Arguments:	C strings (null-terminated `char` arrays) `str1` and `str2`
Function return value:	An `int` value < 0, if `str1` < `str2` lexicographically The `int` value 0, if `str1` = `str2` lexicographically An `int` value > 0, if `str1` > `str2` lexicographically

`strcpy(toStr, fromStr)`

Arguments:	`toStr` is a `char` array and `fromStr` is a C string (null-terminated `char` array), and `toStr` must be large enough to hold the result
Effect:	`fromStr`, including the null character '\0', is copied to `toStr`, overwriting what was there.
Function return value:	The base address of `toStr`
Note:	Programmers usually ignore the function return value, using the syntax of a void function call rather than a value-returning function call.

`strlen(str)`

Argument:	A C string (null-terminated `char` array) `str`
Function return value:	An `int` value ≥ 0 that is the length of `str` (excluding the '\0')

C.9 The Header File `string`

This header file supplies a programmer-defined data type (specifically, a *class*) named `string`. Associated with the `string` type are a data type `string::size_type` and a named constant `string::npos`, defined as follows:

`string::size_type`	An unsigned integer type related to the number of characters in a string
`string::npos`	The maximum value of type `string::size_type`

There are dozens of functions associated with the `string` type. Below are several of the most important ones. In the descriptions, `s` is assumed to be a variable (an *object*) of type `string`.

`s.c_str()`

Arguments:	None
Function return value:	The base address of a C string (null-terminated `char` array) corresponding to the characters stored in `s`

`s.find(arg)`

Argument:	An expression of type `string` or `char`, or a C string (such as a literal string)
Function return value:	A value of type `string::size_type` that gives the starting position in `s` where `arg` was found. If `arg` was not found, the return value is `string::npos`.
Note:	Positions of characters within a string are numbered starting at 0.

`getline(inStream, s)`

Arguments:	An input stream `inStream` (of type `istream` or `ifstream`) and a `string` object `s`
Effect:	Characters are input from `inStream` and stored into `s` until the newline character is encountered. (The newline character is consumed but not stored into `s`.)
Function return value:	Although the function technically returns a value (which we do not discuss here), programmers usually invoke the function as though it were a void function.

`s.length()`

Arguments:	None
Function return value:	A value of type `string::size_type` that gives the number of characters in the string

`s.size()`

Arguments:	None
Function return value:	The same as `s.length()`

`s.substr(pos, len)`

Arguments: Two unsigned integers, `pos` and `len`, representing a position and a length. The value of `pos` must be less than `s.length()`.

Function return value: A temporary `string` object that holds a substring of at most `len` characters, starting at position `pos` of `s`. If `len` is too large, it means "to the end" of the string in `s`.

Note: Positions of characters within a string are numbered starting at 0.

C.10 The Header File `sstream`

This header file supplies various classes derived from `iostream` that enable the user to apply stream-like operations to strings instead of files. In this text, we use only the `ostringstream` class, which enables us to convert numeric types to formatted string values. We can apply the insertion operator (<<) to an `ostringstream` object to append values to it, and then read back its contents in string form, using the `str` member function. In the following, `oss` is an `ostringstream` object.

`ostringstream()`

Argument: None

Return value: An empty `osstringstream` object

`ostringstream(arg)`

Argument: A string

Return value: An `ostringstream` object initialized to the argument string <<

Left argument: An `ostringstream` object

Right argument: An expression that can be converted to a string, including output manipulators.

Return value: An `ostringstream` object

Evaluated: Left to right

`oss.str()`

Argument: None

Return value: A string containing whatever has been appended to `oss`

`oss.str(arg)`

Argument: A string

Effect: Assigns the value of the argument to `oss`. Using the empty string as the argument effectively sets the content of the `ostringstream` object to be empty.

Appendix D Using This Book with a Prestandard Version of C++

D.1 The `string` Type

Prior to the ISO/ANSI C++ language standard, the standard library did not provide a `string` data type. Compiler vendors often supplied their own programmer-defined types with names like `String`, `StringType`, and so on. The syntax and semantics of string operations often varied from vendor to vendor.

71560_APPX_DaleNEW.indd 995 6/12/09 2:22:13 PM

For readers with prestandard compilers, the authors of this book have created a data type named **StrType** that mimics a subset of the standard **string** type. The subset is sufficient to match the string operations displayed throughout this book.

The files related to **StrType** are available for download from the publisher's Web site (www.jbpub.com). Among the files is one called **README.TXT**, which explains how to compile the source code and link it with the programs you write. Another file is the header file **strtype.h**, which contains important declarations that define the **StrType** type. Programs that use **StrType** must **#include** this header file:

```
#include "strtype.h"
```

In the **#include** directive, you cannot place the file name in angle brackets (< >), which tell the preprocessor to look for the file in the standard include directory. Instead, you enclose the file name in double quotes (" "). The double quotes tell the preprocessor to look for the file in the programmer's current directory. Therefore, to use **StrType** in your program, you must (a) verify that the file **strtype.h** is in the directory in which you are currently working on your program, and (b) make sure your program uses the directive

```
#include "strtype.h"
```

Additional directions are given in **README.TXT**.

Throughout this book you can use **StrType** instead of the **string** data type as follows. First, in your variable declarations, substitute the word **StrType** for **string** as the name of the data type. Second, change the directive **#include <string>** to **#include "strtype.h"**. For example, instead of

```
#include <string>
  ⋮
string lastName;
```

you would write

```
#include "strtype.h"
  ⋮
StrType lastName;
```

Finally, there is a restriction on performing input into **StrType** variables. Chapter 4 discusses the use of the >> operator and the **getline** function to input characters into a **string** variable. Using >> with **StrType** variables, at most 1023 characters can be read and stored into one variable. In practice, however, this isn't much of a restriction. It would be extremely rare for an input string to consist of that many characters. Input using **getline** is also restricted to 1023 characters. In the function call

```
getline(cin, myString);
```

in which **myString** is a **StrType** variable, the **getline** function does not skip leading whitespace characters and continues until it either has read 1023 characters or it reaches the newline character '\n', whichever comes first. That is, **getline** reads and stores an entire input line (to a maximum of 1023 characters), embedded blanks and all. Note that for an input line of 1023 characters or less, the newline character *is* consumed (but is not stored into **myString**).

71560_APPX_DaleNEW.indd 996 6/12/09 2:22:13 PM

D.2 Standard Header Files and Namespaces

Historically, the standard header files in both C and C++ had file names ending in `.h` (meaning "header file"). Certain header files—for example, `iostream.h`, `iomanip.h`, and `fstream.h`—related specifically to C++. Others, such as `math.h`, `stddef.h`, `stdlib.h`, and `string.h`, were carried over from the C standard library and were available to both C and C++ programs. When you used an `#include` directive such as

```
#include <math.h>
```

near the beginning of your program, all identifiers declared in `math.h` were introduced into your program in global scope (as discussed in Chapter 8). With the advent of the *namespace* mechanism in ISO/ANSI standard C++ (see Chapter 2 and, in more detail, Chapter 8), all of the standard header files were modified so that identifiers are declared within a namespace called `std`. In standard C++, when you `#include` a standard header file, the identifiers therein are not automatically placed into global scope.

To preserve compatibility with older versions of C++ that still need the original files `iostream.h`, `math.h`, and so forth, the new standard header files are renamed as follows: The C++-related header files have the `.h` removed, and the header files from the C library have the `.h` removed *and* the letter *c* inserted at the beginning. Here is a list of the old and new names for some of the most commonly used header files.

Old Name	New Name
iostream.h	iostream
iomanip.h	iomanip
fstream.h	fstream
assert.h	cassert
ctype.h	cctype
float.h	cfloat
limits.h	climits
math.h	cmath
stddef.h	cstddef
stdlib.h	cstdlib
string.h	cstring

Be careful: The last entry in the list above refers to the C language concept of a string and is unrelated to the `string` type defined in the C++ standard library.

If you are working with a prestandard compiler that does not recognize the new header file names or namespaces, simply substitute the old header file names for the new ones as you encounter them in the book. For example, where we have written

```
#include <iostream>
using namespace std;
```

you would write

```
#include <iostream.h>
```

For compatibility, C++ systems are likely to retain both versions of the header files for some time to come.

D.3 The **fixed** and **showpoint** Manipulators

Chapter 3 introduces five manipulators for formatting the output: **endl**, **setw**, **fixed**, **showpoint**, and **setprecision**. If you are using a prestandard compiler with the header file **iostream.h**, the **fixed** and **showpoint** manipulators may not be available.

In place of the following code shown in Chapter 3,

```
#include <iostream>
using namespace std;
  ⋮
cout << fixed << showpoint;        // Set up floating-pt.
                                   //   output format
```

you can substitute the following code:

```
#include <iostream.h>
  ⋮
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::showpoint);
```

These two statements employ some advanced C++ notation. Our advice is simply to use the statements just as you see them and not worry about the details. Here's the general idea. **setf** is a void function associated with the **cout** stream. (Note that the dot, or period, between **cout** and **setf** is required.) The first function call ensures that floating-point numbers are always printed in decimal form rather than scientific notation. The second function call specifies that the decimal point should always be printed, even for whole numbers. In other words, these two function calls accomplish the same effect as the **fixed** and **showpoint** manipulators.

Note: If your compiler complains about the syntax **ios::fixed**, **ios::floatfield**, or **ios::showpoint**, you may have to replace **ios** with **ios_base** as follows:

```
cout.setf(ios_base::fixed, ios_base::floatfield);
cout.setf(ios_base::showpoint);
```

D.4 The **bool** Type

Before the ISO/ANSI C++ language standard, C++ did not have a **bool** data type. Some prestandard compilers implemented the **bool** type before the standard was approved, but others did not. If your compiler does not recognize the **bool** type, the following discussion will assist you in writing programs that are compatible with those in this book.

In versions of C++ without the **bool** type, the value 0 represents *false*, and any nonzero value represents *true*. It is customary in pre–standard C++ to use the **int** type to represent Boolean data:

```
int dataOK;
  ⋮
dataOK = 1;    // Store "true" into dataOK
  ⋮
dataOK = 0;    // Store "false" into dataOK
```

To make the code more self-documenting, many pre–standard C++ programmers define their own Boolean data type by using a *Typedef statement*. This statement allows you to introduce a new name for an existing data type:

```
typedef int bool;
```

All this statement does is tell the compiler to substitute the word `int` for every occurrence of the word `bool` in the rest of the program. Thus, when the compiler encounters a statement such as

```
bool dataOK;
```

it translates the statement into

```
int dataOK;
```

With the Typedef statement and declarations of two named constants, `true` and `false`, the code at the beginning of this discussion becomes the following:

```
typedef int bool;
const int true = 1;
const int false = 0;
  ⋮
bool dataOK;
  ⋮
dataOK = true;
  ⋮
dataOK = false;
```

Throughout the book, our programs use the words `bool`, `true`, and `false` when manipulating Boolean data. If your compiler recognizes `bool` as a built-in type, there is nothing you need to do. Otherwise, here are three steps you can take.

1. Use your system's editor to create a file containing the following lines:

   ```
   #ifndef BOOL_H
   #define BOOL_H
   typedef int bool;
   const int true = 1;
   const int false = 0;
   #endif
   ```

   Don't worry about the meaning of the first, second, and last lines. They are explained in Chapter 14. Simply type the lines as you see them above.

2. Save the file you created in step 1, giving it the name `bool.h`. Save this file into the same directory in which you work on your C++ programs.

3. Near the top of every program in which you need `bool` variables, type the line

   ```
   #include "bool.h"
   ```

   Be sure to surround the file name with double quotes, not angle brackets (`< >`). The quotes tell the preprocessor to look for `bool.h` in your current directory rather than the C++ system directory.

With `bool`, `true`, and `false` defined in this fashion, the programs in this book run correctly, and you can use `bool` in your own programs, even if it is not a built-in type.

Appendix E Character Sets

The following charts show the ordering of characters in two widely used character sets: ASCII (American Standard Code for Information Interchange) and EBCDIC (Extended Binary Coded Decimal Interchange Code). The internal representation for each character is shown in decimal. For example, the letter *A* is represented internally as the integer 65 in ASCII and as 193 in EBCDIC. The space (blank) character is denoted by a "□".

Left Digit(s) / *Right Digit*	*ASCII*									
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	□	!	“	#	$	%	&	’
4	(	)	*	+	,	−	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Codes 00–31 and 127 are the following nonprintable control characters:

NUL	Null character	VT	Vertical tab	SYN	Synchronous idle
SOH	Start of header	FF	Form feed	ETB	End of transmitted block
STX	Start of text	CR	Carriage return	CAN	Cancel
ETX	End of text	SO	Shift out	EM	End of medium
EOT	End of transmission	SI	Shift in	SUB	Substitute
ENQ	Enquiry	DLE	Data link escape	ESC	Escape
ACK	Acknowledge	DC1	Device control one	FS	File separator
BEL	Bell character (beep)	DC2	Device control two	GS	Group separator
BS	Back space	DC3	Device control three	RS	Record separator
HT	Horizontal tab	DC4	Device control four	US	Unit separator
LF	Line feed	NAK	Negative acknowledge	DEL	Delete

71560_APPX_DaleNEW.indd 1000 6/12/09 2:22:13 PM

Left Digit(s) / *Right Digit*	*EBCDIC*									
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
6					□					
7					¢	.	<	(	+	|
8	&									
9	!	$	*	)	;	¬	–	/		
10							ˆ	,	%	_
11	>	?								
12		`	:	#	@	′	=	“		a
13	b	c	d	e	f	g	h	i		
14						j	k	l	m	n
15	o	p	q	r						
16		~	s	t	u	v	w	x	y	z
17								\	{	}
18	[	]								
19				A	B	C	D	E	F	G
20	H	I								J
21	K	L	M	N	O	P	Q	R		
22							S	T	U	V
23	W	X	Y	Z						
24	0	1	2	3	4	5	6	7	8	9

In the EBCDIC table, nonprintable control characters—codes 00–63, 250–255, and those for which empty spaces appear in the chart—are not shown.

Appendix F Program Style, Formatting, and Documentation

Throughout this text, we encourage the use of good programming style and documentation. Although the programs you write for class assignments may not be looked at by anyone except the person grading your work, outside of class you will write programs that will be used by others.

Useful programs have very long lifetimes, during which they must be modified and updated. When maintenance work must be done, either you or another programmer will have to do it. Good style and documentation are essential if another programmer is to understand and work with your program. You will also discover that after not working with your own program for a few months, you'll be amazed at how many of the details you've forgotten.

F.1 General Guidelines

The style used in the programs and fragments throughout this text provides a good starting point for developing your own style. Our goals in creating this style were to make it simple, consistent, and easy to read.

Style is of benefit only for a human reader of your program—differences in style make no difference to the computer. Good style includes the use of meaningful variable names, comments, and indentation of control structures, all of which help others to understand and

work with your program. Perhaps the most important aspect of program style is consistency. If the style within a program is not consistent, then it becomes misleading and confusing.

Sometimes, a particular style is specified for you by your instructor or by the company you work for. When you are modifying someone else's code, you should use his or her style in order to maintain consistency within the program. However, you will also develop your own, personal programming style based on what you've been taught, your own experience, and your personal taste.

F.2 Comments

Comments are extra information included to make a program easier to understand. You should include a comment anywhere the code is difficult to understand. However, don't overcomment. Too many comments in a program can obscure the code and be a source of distraction.

In our style, there are four basic types of comments: headers, declarations, in-line, and sidebar.

Header comments appear at the top of the program and should include your name, the date that the program was written, and its purpose. It is also useful to include sections describing input, output, and assumptions. Think of the header comments as the reader's introduction to your program. Here is an example:

```
// This program computes the sidereal time for a given date and
// solar time.
//
// Written By: Your Name
//
// Date Completed: 4/8/10
//
// Input: A date and time in the form MM DD YYYY HH MM SS
//
// Output: Sidereal time in the form HH MM SS
//
// Assumptions: Solar time is specified for a longitude of 0
//     degrees (GMT, UT, or Z time zone)
```

Header comments should also be included for all user-defined functions (see Chapters 7 and 8).

Declaration comments accompany the constant and variable declarations in the program. Anywhere that an identifier is declared, it is helpful to include a comment that explains its purpose. In programs in the text, declaration comments appear to the right of the identifier being declared. For example:

```
const float E = 2.71828;     // The base of the natural logarithms

float deltaX;                // The difference in the x direction
float deltaY;                // The difference in the y direction
```

Notice that aligning the comments gives the code a neater appearance and is less distracting.

In-line comments are used to break long sections of code into shorter, more comprehensible fragments. These are often the names of modules in your algorithm design, although

you may occasionally choose to include other information. It is generally a good idea to surround in-line comments with blank lines to make them stand out. For example:

```
// Prepare file for reading

scoreFile.open("scores.dat");

// Get data

scoreFile >> test1 >> weight1;
scoreFile >> test2 >> weight2;
scoreFile >> test3 >> weight3;

// Print heading

cout << "Test Score   Weight" << endl;
```

Even if comments are not used, blank lines can be inserted wherever there is a logical break in the code that you would like to emphasize.

Sidebar comments appear to the right of executable statements and are used to shed light on the purpose of the statement. Sidebar comments are often just pseudocode statements from the lowest levels of your design. If a complicated C++ statement requires some explanation, the pseudocode statement should be written to the right of the C++ statement. For example:

```
while (file1 && file2)    // While neither file is empty...
{
    ⋮
```

In addition to the four main types of comments that we have discussed, there are some miscellaneous comments that we should mention. After the `main` function, we recommend using a row of asterisks (or dashes or equal signs or ...) in a comment before and after each function to help it to stand out. For example:

```
//******************************************************************

void PrintSecondHeading()
{
    ⋮
}

//******************************************************************
```

For example, one can use C++'s alternative comment form

```
/* Some comment */
```

to document the flow of information for each parameter of a function:

```
void GetData( /* out */ int age,        // Patient's age
              /* out */ int weight )    // Patient's weight
{
    ⋮
}
```

```
void Print( /* in */     float val,      // Value to be printed
            /* in/out */ int&  count )   // Number of lines printed
                                          //   so far
{
    ⋮
}
```

(Chapter 8 describes the purpose of labeling each parameter as `/* in */`, `/* out */`, or `/* in/out */`.)

Programmers sometimes place a comment after the right brace of a block (compound statement) to indicate which control structure the block belongs to:

```
while (num >= 0)
{
     ⋮

    if (num == 25)
    {
         ⋮
    } // if
} // while
```

Attaching comments in this fashion can help to clarify the code and aid in debugging mismatched braces.

F.3 Identifiers

The most important consideration in choosing a name for a data item or function in a program is that the name convey as much information as possible about what the data item is or what the function does. The name should also be readable in the context in which it is used. For example, the following names convey the same information but one is more readable than the other:

```
datOfInvc        invoiceDate
```

Identifiers for types, constants, and variables should be nouns, whereas names of void functions (non-value-returning functions) should be imperative verbs or phrases containing imperative verbs. Because of the way that value-returning functions are invoked, their names should be nouns or occasionally adjectives. Here are some examples:

Variables	`address`, `price`, `radius`, `monthNumber`
Constants	`PI`, `TAX_RATE`, `STRING_LENGTH`, `ARRAY_SIZE`
Data types	`NameType`, `CarMakes`, `RoomLists`, `Hours`
Void functions	`GetData`, `ClearTable`, `PrintBarChart`
Value-returning functions	`CubeRoot`, `Greatest`, `Color`, `AreaOf`, `IsEmpty`

Although an identifier may be a series of words, very long identifiers can become quite tedious and can make the program difficult to read.

The best approach to designing an identifier is to try writing out different names until you reach an acceptable compromise—and then write an especially informative declaration comment next to the declaration.

Capitalization is another consideration when choosing an identifier. C++ is a case-sensitive language; that is, uppercase and lowercase letters are distinct. Different programmers

use different conventions for capitalizing identifiers. In this text, we begin each variable name with a lowercase letter and capitalize the beginning of each successive English word. We begin each function name and data type name with a capital letter and, again, capitalize the beginning of each successive English word. For named constants, we capitalize the entire identifier, separating successive English words with underscore (_) characters. Keep in mind, however, that C++ reserved words such as `main`, `if`, and `while` are always lowercase letters, and the compiler will not recognize them if you capitalize them differently.

F.4 Formatting Lines and Expressions

C++ allows you to break a long statement in the middle and continue onto the next line. (However, you cannot split a line in the middle of an identifier, a literal constant, or a string.) When you must split a line, it's important to choose a breaking point that is logical and readable. Compare the readability of the following code fragments.

```
cout << "For a radius of " << radius << " the diameter of the cir"
     << "cle is " << diameter << endl;
cout << "For a radius of " << radius
     << " the diameter of the circle is " << diameter << endl;
```

When you must split an expression across multiple lines, try to end each line with an operator. Also, try to take advantage of any repeating patterns in the expression. For example,

```
meanOfMaxima = (Maximum(set1Value1, set1Value2, set1Value3) +
                Maximum(set2Value1, set2Value2, set2Value3) +
                Maximum(set3Value1, set3Value2, set3Value3)) / 3.0;
```

When writing expressions, also keep in mind that spaces improve readability. Usually you should include one space on either side of the = operator and most other operators. Occasionally, spaces are left out to emphasize the order in which operations are performed. Here are some examples:

```
if (x+y > y+z)
    maximum = x + y;
else
    maximum = y + z;
hypotenuse = sqrt(a*a + b*b);
```

F.5 Indentation

The purpose of indenting statements in a program is to provide visual cues to the reader and to make the program easier to debug. When a program is properly indented, the way the statements are grouped is immediately obvious. Compare the following two program fragments:

```
while (count <= 10)
{
cin >> num;
if (num == 0)
{
count++;
num = 1;
}
cout << num << endl;
cout << count << endl;
}
```

```
while (count <= 10)
{
    cin >> num;
    if (num == 0)
    {
        count++;
        num = 1;
    }
    cout << num << endl;
    cout << count << endl;
}
```

As a basic rule in this text, each nested or lower-level item is indented by four spaces. Exceptions to this rule are parameter declarations and statements that are split across two or more lines. Indenting by four spaces is a matter of personal preference. Some people prefer to indent by three, five, or even more than five spaces.

In this book, we indent the entire body of a function. Also, in general, any statement that is part of another statement is indented. For example, the If-Then-Else contains two parts, the then-clause and the else-clause. The statements within both clauses are indented four spaces beyond the beginning of the If-Then-Else statement. The If-Then statement is indented like the If-Then-Else, except that there is no else-clause. Here are examples of the If-Then-Else and the If-Then:

```
if (sex == MALE)
{
    maleSalary = maleSalary + salary;
    maleCount++;
}
else
    femaleSalary = femaleSalary + salary;

if (count > 0)
    average = total / count;
```

For nested If-Then-Else statements that form a generalized multiway branch (the If-Then-Else-If, described in Chapter 5), a special style of indentation is used in the text. Here is an example:

```
if (month == JANUARY)
    monthNumber = 1;
else if (month == FEBRUARY)
    monthNumber = 2;
else if (month == MARCH)
    monthNumber = 3;
else if (month == APRIL)
    ⋮
else
    monthNumber = 12;
```

The remaining C++ statements all follow the basic indentation guideline mentioned previously. For reference purposes, here are examples of each:

```
while (count <= 10)
{
    cin >> value;
    sum = sum + value;
    count++;
}

do
{
    GetAnswer(letter);
    PutAnswer(letter);
} while (letter != 'N');

for (count = 1; count <= numSales; count++)
    cout << '*';
```

```
for (count = 10; count >= 1; count--)
{
    inFile >> dataItem;
    outFile << dataItem << ' ' << count << endl;
}

switch (color)
{
    RED    : cout << "Red";
             break;
    ORANGE : cout << "Orange";
             break;
    YELLOW : cout << "Yellow";
             break;
    GREEN  :
    BLUE   :
    INDIGO :
    VIOLET : cout << "Short visible wavelengths";
             break;
    WHITE  :
    BLACK  : cout << "Not valid colors";
             color = NONE;
}
```

Appendix G More on Floating-Point Numbers

We have used floating-point numbers off and on since we introduced them in Chapter 2, but we have not examined them in depth. Floating-point numbers have special properties when used on the computer. In this appendix we consider them in detail.

G.1 Representation of Floating-Point Numbers

Let's assume we have a computer in which each memory location is the same size and is divided into a sign plus five decimal digits. When a variable or constant is defined, the location assigned to it consists of five digits and a sign. When an `int` variable or constant is defined, the interpretation of the number stored in that place is straightforward. When a `float` variable or constant is defined, the number stored there has both a whole number part and a fractional part, so it must be coded to represent both parts.

Let's see what such coded numbers might look like. The range of whole numbers we can represent with five digits is −99,999 through +99,999:

–99999 through +99999

+	9	9	9	9	9	Largest positive number
+	0	0	0	0	0	Zero
–	9	9	9	9	9	Largest negative number

Precision The maximum number of significant digits.

Our precision (the number of digits we can represent) is five digits, and each number within that range can be represented exactly.

What happens if we allow one of those digits (the leftmost one, for example) to represent an exponent?

Then +82345 represents the number $+2345 \times 10^8$. The range of numbers we now can represent is much larger:

$$-9999 \times 10^9 \text{ through } 9999 \times 10^9$$

or

$$-9{,}999{,}000{,}000{,}000 \text{ through } +9{,}999{,}000{,}000{,}000$$

Significant digits The digits from the first nonzero digit on the left to the last nonzero digit on the right (plus any 0 digits that are exact).

However, our precision is now only four digits; that is, only four-digit numbers can be represented exactly in our system. What happens to numbers with more digits? The four leftmost digits are represented correctly, and the rightmost digits, or least significant digits, are lost (assumed to be 0). **FIGURE G.1** shows what happens. Note that 1,000,000 can be represented exactly but −4,932,416 cannot, because our coding scheme limits us to four significant digits.

To extend our coding scheme to represent floating-point numbers, we must be able to represent negative exponents. Examples are

$$7394 \times 10^{-2} = 73.94$$

and

$$22 \times 10^{-4} = .0022$$

NUMBER	POWER OF TEN NOTATION	CODED REPRESENTATION (Sign, Exp, digits)	VALUE
+99,999	$+9999 \times 10^1$	+ 1 9 9 9 9	+99,990
−999,999	-9999×10^2	− 2 9 9 9 9	−999,900
+1,000,000	-1000×10^3	+ 3 1 0 0 0	+1,000,000
−4,932,416	-4932×10^3	− 3 4 9 3 2	−4,932,000

FIGURE G.1 Coding Using Positive Exponents

71560_APPX_DaleNEW.indd 1008 6/12/09 2:22:15 PM

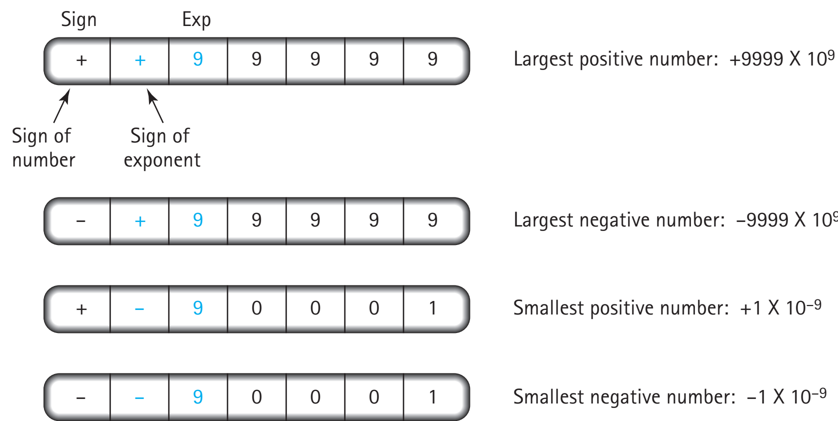


FIGURE G.2 Coding Using Positive and Negative Exponents

Because our scheme does not include a sign for the exponent, let's change it slightly. The existing sign becomes the sign of the exponent, and we add a sign to the far left to represent the sign of the number itself (see **FIGURE G.2**).

All the numbers between -9999×10^9 and 9999×10^9 can now be represented accurately to four digits. Adding negative exponents to our scheme allows us to represent fractional numbers as small as 1×10^{-9}.

FIGURE G.3 shows how we would encode some floating-point numbers. Note that our precision is still only four digits. The numbers 0.1032, −5.406, and 1,000,000 can be represented exactly. The number 476.0321, however, with seven significant digits, is represented as 476.0; the 321 cannot be represented. (We should point out that some computers perform *rounding* rather than simple truncation when discarding excess digits. Using our assumption of four significant digits, such a machine would store 476.0321 as 476.0 but

NUMBER	POWER OF TEN NOTATION	CODED REPRESENTATION							VALUE
		Sign		Exp					
0.1032	$+1032 \times 10^{-4}$	+	–	4	1	0	3	2	0.1032
–5.4060	-5406×10^{-3}	–	–	3	5	4	0	6	–5.406
–0.003	-3000×10^{-6}	–	–	6	3	0	0	0	–0.0030
476.0321	$+4760 \times 10^{-1}$	+	–	1	4	7	6	0	476.0
1,000,000	$+1000 \times 10^{3}$	+	+	3	1	0	0	0	1,000,000

FIGURE G.3 Coding of Some Floating Point Numbers

71560_APPX_DaleNEW.indd 1009 6/12/09 2:22:15 PM

would store 476.0823 as 476.1. We continue our discussion assuming simple truncation rather than rounding.)

G.2 Arithmetic with Floating-Point Numbers

When we use integer arithmetic, our results are exact. Floating-point arithmetic, however, is seldom exact. To understand why, let's add the three floating-point numbers *x*, *y*, and *z* using our coding scheme.

First, we add *x* to *y* and then we add *z* to the result. Next, we perform the operations in a different order, adding *y* to *z*, and then adding *x* to that result. The associative law of arithmetic says that the two answers should be the same—but are they? Let's use the following values for *x*, *y*, and *z*:

$x = -1324 \times 10^3 \quad y = 1325 \times 10^3 \quad z = 5424 \times 10^0$

Here is the result of adding *z* to the sum of *x* and *y*:

```
(x)     -1324 × 10^3
(y)      1325 × 10^3
--------------------------------------
            1 × 10^3    =  1000 × 10^0

(x+y)    1000 × 10^0
(z)      5424 × 10^0
--------------------------------------
         6424 × 10^0    ←  (x+y)+z
```

Now here is the result of adding *x* to the sum of *y* and *z*:

```
(y)   1325000 × 10^0
(z)      5424 × 10^0
--------------------------------------
      1330424 × 10^0    =  1330 × 10^3 (truncated to four digits)

(y+z)    1330 × 10^3
(x)     -1324 × 10^3
--------------------------------------
            6 × 10^3    =  6000 × 10^0 ← x + (y + z)
```

Representational Error Arithmetic error that occurs when the precision of the true result of an arithmetic operation is greater than the precision of the machine.

These two answers are the same in the thousands place but are different thereafter. The error behind this discrepancy is called **representational error**.

Because of representational errors, it is unwise to use a floating-point variable as a loop control variable. Because precision may be lost in calculations involving floating-point numbers, it is difficult to predict when (or even *if*) a loop control variable of type `float` (or `double` or `long double`) will equal the termination value. A count-controlled loop with a floating-point control variable can behave unpredictably.

Also because of representational errors, you should never compare floating-point numbers for exact equality. Rarely are two floating-point numbers exactly equal, and thus you should compare them only for near equality. If the difference between the two numbers is less than some acceptable small value, you can consider them equal for the purposes of the given problem.

G.3 Implementation of Floating-Point Numbers in the Computer

All computers limit the precision of floating-point numbers, although modern machines use binary rather than decimal arithmetic. In our representation, we used only 5 digits to simplify the examples, and some computers really are limited to only 4 or 5 digits of precision. A more typical system might provide 6 significant digits for `float` values, 15 digits for `double` values, and 19 for the `long double` type. We have shown only a single-digit exponent, but most systems allow 2 digits for the `float` type and up to 4-digit exponents for type `long double`.

When you declare a floating-point variable, part of the memory location is assumed to contain the exponent, and the number itself (called the *mantissa*) is assumed to be in the balance of the location. The system is called floating-point representation because the number of significant digits is fixed, and the decimal point conceptually is allowed to float (move to different positions as necessary). In our coding scheme, every number is stored as four digits, with the leftmost digit being nonzero and the exponent adjusted accordingly. Numbers in this form are said to be *normalized*. The number 1,000,000 is stored as

+	+	3	1	0	0	0

and 0.1032 is stored as

+	−	4	1	0	3	2

Normalization provides the maximum precision possible.

Model Numbers Any real number that can be represented exactly as a floating-point number in the computer is called a *model number.* A real number whose value cannot be represented exactly is approximated by the model number closest to it. In our system with four digits of precision, 0.3021 is a model number. The values 0.3021409, 0.3021222, and 0.30209999999 are examples of real numbers that are represented in the computer by the same model number. The following table shows all of the model numbers for an even simpler floating-point system that has one digit in the mantissa and an exponent that can be −1, 0, or 1. Zero is a special case, because it has the same value regardless of the exponent.

0.1×10^{-1}	0.1×10^{0}	$0.1 \times 10^{+1}$
0.2×10^{-1}	0.2×10^{0}	$0.2 \times 10^{+1}$
0.3×10^{-1}	0.3×10^{0}	$0.3 \times 10^{+1}$
0.4×10^{-1}	0.4×10^{0}	$0.4 \times 10^{+1}$
0.5×10^{-1}	0.5×10^{0}	$0.5 \times 10^{+1}$
0.6×10^{-1}	0.6×10^{0}	$0.6 \times 10^{+1}$
0.7×10^{-1}	0.7×10^{0}	$0.7 \times 10^{+1}$
0.8×10^{-1}	0.8×10^{0}	$0.8 \times 10^{+1}$
0.9×10^{-1}	0.9×10^{0}	$0.9 \times 10^{+1}$

The difference between a real number and the model number that represents it is a form of representational error called *rounding error.* We can measure rounding error in

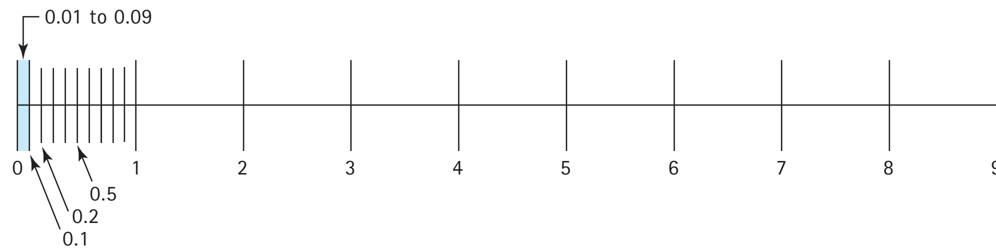


FIGURE G.4 A Graphical Representation of Model Numbers

two ways. The *absolute error* is the difference between the real number and the model number. For example, the absolute error in representing 0.3021409 by the model number 0.3021 is 0.0000409. The *relative error* is the absolute error divided by the real number and sometimes is stated as a percentage. For example, 0.0000409 divided by 0.3021409 is 0.000135, or 0.0135%.

The maximum absolute error depends on the *model interval*—the difference between two adjacent model numbers. In our example, the interval between 0.3021 and 0.3022 is 0.0001. The maximum absolute error in this system, for this interval, is less than 0.0001. Adding digits of precision makes the model interval (and thus the maximum absolute error) smaller.

The model interval is not a fixed number; it varies with the exponent. To see why the interval varies, consider that the interval between 3021.0 and 3022.0 is 1.0, which is 10^4 times larger than the interval between 0.3021 and 0.3022. This makes sense, because 3021.0 is simply 0.3021 times 10^4. Thus, a change in the exponent of the model numbers adjacent to the interval has an equivalent effect on the size of the interval. In practical terms, this means that we give up significant digits in the fractional part in order to represent numbers with large integer parts. **FIGURE G.4** illustrates this by graphing all of the model numbers listed in the preceding table.

We also can use relative and absolute error to measure the rounding error resulting from calculations. For example, suppose we multiply 1.0005 by 1000. The correct result is 1000.5, but because of rounding error, our four-digit computer produces 1000.0 as its result. The absolute error of the computed result is 0.5, and the relative error is 0.05%. Now suppose we multiply 100,050.0 by 1000. The correct result is 100,050,000, but the computer produces 100,000,000 as its result. If we look at the relative error, it is still a modest 0.05%, but the absolute error has grown to 50,000. Notice that this example is another case of changing the size of the model interval.

Whether it is more important to consider the absolute error or the relative error depends on the situation. It is unacceptable for an audit of a company to discover a $50,000 accounting error; the fact that the relative error is only 0.05% is not important. On the other hand, a 0.05% relative error is acceptable in representing prehistoric dates because the error in measurement techniques increases with age. That is, if we are talking about a date roughly 10,000 years ago, an absolute error of 5 years is acceptable; if the date is 100,000,000 years ago, then an absolute error of 50,000 years is equally acceptable.

Comparing Floating-Point Numbers We have cautioned against comparing floating-point numbers for exact equality. Our exploration of representational errors in this chapter reveals why calculations may not produce the expected results even though it appears that they

should. In Chapter 5, we wrote an expression that compares two floating-point variables `r` and `s` for near equality using the floating-point absolute value function `fabs`:

```
fabs(r - s) < 0.00001
```

From our discussion of model numbers, you now can recognize that the constant 0.00001 in this expression represents a maximum absolute error. We can generalize this expression as

```
fabs(r - s) < ERROR_TERM
```

where `ERROR_TERM` is a value that must be determined for each programming problem.

What if we want to compare floating-point numbers with a relative error measure? We must multiply the error term by the value in the problem to which the error is relative. For example, if we want to test whether `r` and `s` are "equal" within 0.05% of `s`, we write the following expression:

```
fabs(r - s) < 0.0005 * s
```

Keep in mind that the choice of the acceptable error and whether it should be absolute or relative depends on the problem being solved. The error terms we have shown in our example expressions are completely arbitrary and may not be appropriate for most problems. In solving a problem that involves the comparison of floating-point numbers, you typically want an error term that is as small as possible. Sometimes the choice is specified in the problem description or is reasonably obvious. Some cases require careful analysis of both the mathematics of the problem and the representational limits of the particular computer. Such analyses are the domain of a branch of mathematics called *numerical analysis* and are beyond the scope of this text.

Underflow and Overflow In addition to representational errors, there are two other problems to watch out for in floating-point arithmetic: *underflow* and *overflow.*

Underflow is the condition that arises when the value of a calculation is too small to be represented. Going back to our decimal representation, let's look at a calculation involving small numbers:

$$\begin{array}{r} 4210 \times 10^{-8} \\ \times\ 2000 \times 10^{-8} \\ \hline 8420000 \times 10^{-16} \end{array} = 8420 \times 10^{-13}$$

This value cannot be represented in our scheme because the exponent -13 is too small. Our minimum is -9. One way to resolve the problem is to set the result of the calculation to 0.0. Obviously, any answer depending on this calculation will not be exact.

Overflow is a more serious problem because there is no logical recourse when it occurs. For example, the result of the calculation

$$\begin{array}{r} 9999 \times 10^{9} \\ \times\ 1000 \times 10^{9} \\ \hline 9999000 \times 10^{18} \end{array} = 9999 \times 10^{21}$$

cannot be stored, so what should we do? To be consistent with our response to underflow, we could set the result to 9999×10^9 (the maximum representable value in this case). Yet this seems intuitively wrong. The alternative is to stop with an error message.

C++ does not define what should happen in the case of overflow or underflow. Different implementations of C++ solve the problem in different ways. You might try to cause an overflow with your system and see what happens. Some systems may print a run-time error message such as "FLOATING POINT OVERFLOW." On other systems, you may get the largest number that can be represented.

Although we are discussing problems with floating-point numbers, integer numbers also can overflow both negatively and positively. Most implementations of C++ ignore integer overflow. To see how your system handles the situation, you should try adding 1 to `INT_MAX` and −1 to `INT_MIN`. On most systems, adding 1 to `INT_MAX` sets the result to `INT_MIN`, a negative number.

Sometimes you can avoid overflow by arranging computations carefully. Suppose you want to know how many different five-card poker hands can be dealt from a deck of cards. What we are looking for is the number of *combinations* of 52 cards taken 5 at a time. The standard mathematical formula for the number of combinations of n things taken r at a time is

$$\frac{n!}{r!(n-r)!}$$

We could use the `Factorial` function we wrote in the Recursion Chapter and write this formula in an assignment statement:

```
hands = Factorial(52) / (Factorial(5) * Factorial(47));
```

The only problem is that 52! is a very large number (approximately $8.065X \times 10^{67}$). And 47! is also large (approximately 2.5862×10^{59}). Both of these numbers are well beyond the capacity of most systems to represent exactly as integers (52! requires 68 digits of precision). Even though they can be represented on many machines as floating-point numbers, most of the precision is still lost. By rearranging the calculations, however, we can achieve an exact result on any system with 9 or more digits of precision. How? Consider that most of the multiplications in computing 52! are canceled when the product is divided by 47!

$$\frac{52!}{5! \times 47!} = \frac{52 \times 51 \times 50 \times 49 \times 48 \times 47 \times 46 \times 45 \times 44 \times \ldots}{(5 \times 4 \times 3 \times 2 \times 1) \times (47 \times 46 \times 45 \times 44 \times \ldots)}$$

So, we really only have to compute

```
hands = 52 * 51 * 50 * 49 * 48 / Factorial(5);
```

which means the numerator is 311,875,200 and the denominator is 120. On a system with 9 or more digits of precision, we get an exact answer: 2,598,960 poker hands.

Cancellation Error Another type of error that can happen with floating-point numbers is called cancellation error, a form of representational error that occurs when numbers of widely differing magnitudes are added or subtracted. Let's look at an example:

$$(1 + 0.00001234 - 1) = 0.00001234$$

The laws of arithmetic say this equation should be true. But is it true if the computer does the arithmetic?

$$\begin{array}{r} 100000000 \times 10^{-8} \\ +\quad 1234 \times 10^{-8} \\ \hline 100001234 \times 10^{-8} \end{array}$$

To four digits, the sum is 1000×10^{-3}. Now the computer subtracts 1:

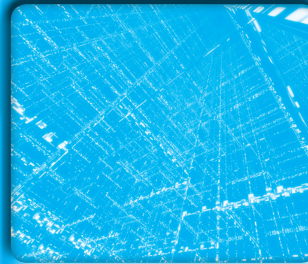
$$\begin{array}{r} 1000 \times 10^{-3} \\ -1000 \times 10^{-3} \\ \hline 0 \end{array}$$

The result is 0, not .00001234.

Sometimes you can avoid adding two floating-point numbers that are drastically different in size by carefully arranging the calculations. Suppose a problem requires many small floating-point numbers to be added to a large floating-point number. The result is more accurate if the program first sums the smaller numbers to obtain a larger number and then adds the sum to the large number.

BACKGROUND INFORMATION

Practical Implications of Limited Precision



A discussion of representational, overflow, underflow, and cancellation errors may seem purely academic. In fact, these errors have serious practical implications in many problems. We close this section with three examples illustrating how limited precision can have costly or even disastrous effects.

During the Mercury space program, several of the spacecraft splashed down a considerable distance from their computed landing points. This delayed the recovery of the spacecraft and the astronaut, putting both in some danger. Eventually, the problem was traced to an imprecise representation of the Earth's rotation period in the program that calculated the landing point.

As part of the construction of a hydroelectric dam, a long set of high-tension cables had to be constructed to link the dam to the nearest power distribution point. The cables were to be several miles long, and each one was to be a continuous unit. (Because of the high power output from the dam, shorter cables couldn't be spliced together.) The cables were constructed at great expense and strung between the two points. It turned out that they were too short, however, so another set had to be manufactured. The problem was traced to errors of precision in calculating the length of the catenary curve (the curve that a cable forms when hanging between two points).

An audit of a bank turned up a mysterious account with a large amount of money in it. The account was traced to an unscrupulous programmer who had used limited precision to his advantage. The bank computed interest on its accounts to a precision of a tenth of a cent. The tenths of cents were not added to the customers' accounts, so the programmer had the extra tenths for all the ac-

BACKGROUND INFORMATION Practical Implications of Limited Precision, continued

counts summed and deposited into an account in his name. Because the bank had thousands of accounts, these tiny amounts added up to a large amount of money. And because the rest of the bank's programs did not use as much precision in their calculations, the scheme went undetected for many months.

The moral of this discussion is twofold: (1) The results of floating-point calculations are often imprecise, and these errors can have serious consequences; and (2) if you are working with extremely large numbers or extremely small numbers, you need more information than this book provides and should consult a numerical analysis text.

Appendix H Using C Strings[1]

Starting with Chapter 2, we use the `string` class to store and manipulate character strings.

```
string name;

name = "James Smith";
len = name.length();
  ⋮
```

In some contexts, we think of a string as a single unit of data. In other contexts, we treat it as a group of individually accessible characters. In particular, we think of a string as a variable-length, linear collection of homogeneous components (of type `char`).

C string In C and C++, a null-terminated sequence of characters stored in an array.

The `string` class from the standard library is one approach to implementing a data type with these properties. However, before this class was developed, the C language, upon which C++ is built, had its own approach: the **C string**.

We are using a C string whenever we write a string constant. In C and C++, a string constant is a sequence of characters enclosed by double quotes:

```
"Hi"
```

A string constant is not a member of the `string` class, rather it is stored as a `char` array with enough components to hold each specified character plus one more—the *null character*. The null character, which is the first character in both the ASCII and EBCDIC character sets, is internally represented by the value 0. In C++, the escape sequence `\0` stands for the null character. When the compiler encounters the string `"Hi"` in a program, it stores the

1. Note: This appendix assumes that you have already read about arrays in Chapter 11.

three characters 'H', 'i', and '\0' into a three-element, anonymous (unnamed) **char** array as follows:

	Unnamed array
[0]	'H'
[1]	'i'
[2]	'\0'

The C string is the only kind of C++ array for which there exists an aggregate constant—the string constant. Notice that in a C++ program, the symbols **'A'** denote a single character, whereas the symbols **"A"** denote two: the character 'A' and the null character.[2]

In addition to C string constants, we can create C string *variables*. To do so, we explicitly declare a **char** array and store into it whatever characters we want to, finishing with the null character. Here's an example:

```
char myStr[8];   // Room for 7 significant characters plus '\0'

myStr[0] = 'H';
myStr[1] = 'i';
myStr[2] = '\0';
```

In C++, all C strings (constants or variables) are assumed to be null-terminated. This convention is agreed upon by all C++ programmers and standard library functions. The null character serves as a sentinel value; it allows algorithms to locate the end of the string. For example, here is a function that determines the length of any C string, not counting the terminating null character:

```
int StrLength(const char str[] )

// Precondition:
//   str holds a null-terminated string
// Postcondition:
//   Function value == number of characters in str (excluding '\0')
{
  int i = 0;              // Index variable

  while (str[i] != '\0')
    i++;
  return i;
}
```

2. C *string* is not an official term used in C++ language manuals. Such manuals typically use the term string. However, we use C *string* to distinguish between the general concept of a string, the built-in array representation defined by the C and C++ languages, and the `string` class in the standard library.

The value of `i` at loop exit is returned by the function. If the array being examined is

[0]	'B'
[1]	'y'
[2]	'\0'
[3]	
	⋮

then `i` equals 2 at loop exit. The string length is therefore 2.

The argument to the **`StrLength`** function can be a C string variable, as in the function call

```
cout << StrLength(myStr);
```

or it can be a string constant:

```
cout << StrLength("Hello");
```

In the first case, the base address of the **`myStr`** array is sent to the function, as we discussed in Chapter 11. In the second case, a base address is also sent to the function—the base address of the unnamed array that the compiler has set aside for the string constant.

There is one more thing we should say about our **`StrLength`** function. A C++ programmer would not actually write this function. The standard library supplies several string-processing functions, one of which is named **`strlen`** and does exactly what our **`StrLength`** function does. Later in this appendix, we look at **`strlen`** and other library functions.

H.1 Initializing C Strings

In Chapter 11, we showed how to initialize an array in its declaration by specifying a list of initial values within braces, like this:

```
int delta[5] = {25, -3, 7, 13, 4};
```

To initialize a C string variable in its declaration, you could use the same technique:

```
char message[8] = {'W', 'h', 'o', 'o', 'p', 's', '!', '\0'};
```

However, C++ allows a more convenient way to initialize a C string. You can simply initialize the array by using a string constant:

```
char message[8] = "Whoops!";
```

This shorthand notation is unique to C strings because there is no other kind of array for which there are aggregate constants.

71560_APPX_DaleNEW.indd 1018 6/12/09 2:22:18 PM

We said in Chapter 12 that you can omit the size of an array when you initialize it in its declaration (in which case, the compiler determines its size). This feature is often used with C strings because it keeps you from having to count the number of characters. For example,

```
char promptMsg[] = "Enter a positive number:";  // Size is 25
char errMsg[] = "Value must be positive.";      // Size is 24
```

Be very careful about one thing: C++ treats initialization (in a declaration) and assignment (in an assignment statement) as two distinct operations. Different rules apply. Remember that array initialization is legal, but aggregate array assignment is not.

```
char myStr[20] = "Hello";   // OK
   ⋮
myStr = "Howdy";            // Not allowed
```

H.2 C String Input and Output

In Chapter 11, we emphasized that C++ does not provide aggregate operations on arrays. There is no aggregate assignment, aggregate comparison, or aggregate arithmetic on arrays. We also said that aggregate input/output of arrays is not possible, with one exception. C strings are that exception. Let's look first at output.

To output the contents of an array that is *not* a C string, you aren't allowed to do this:

```
int alpha[100];
   ⋮
cout << alpha;   // Not allowed
```

Instead, you must write a loop and print the array elements one at a time. However, aggregate output of a null-terminated **char** array (that is, a C string) is valid. The C string can be a constant:

```
cout << "Results are:";
```

or it can be a variable:

```
char msg[8] = "Welcome";
   ⋮
cout << msg;
```

In both cases, the insertion operator (<<) outputs each character in the array until the null character is found. It is up to you to double-check that the terminating null character is present in the array. If not, the << operator will march through the array and into the rest of memory, printing out bytes until—just by chance—it encounters a byte whose integer value is 0.

To input C strings, we have several options. The first is to use the extraction operator (>>), which behaves exactly the same as with **string** class objects. When reading input characters into a C string variable, the >> operator skips leading whitespace characters and then reads successive characters into the array, stopping at the first trailing whitespace character (which is not consumed, but remains as the first character waiting in the input

stream). The >> operator also takes care of adding the null character to the end of the string. For example, assume we have the following code:

```
char firstName[31];   // Room for 30 characters plus '\0'
char lastName[31];

cin >> firstName >> lastName;
```

If the input stream initially looks like this (where ◊ denotes a blank):

```
◊◊John◊Smith◊◊◊25
```

then our input statement stores 'J', 'o', 'h', 'n', and '\0' into **firstName[0]** through **firstName[4]**; stores 'S', 'm', 'i', 't', 'h', and '\0' into **lastName[0]** through **lastName[5]**; and leaves the input stream as

```
◊◊◊25
```

The >> operator, however, has two potential drawbacks.

1. If the array isn't large enough to hold the sequence of input characters (and the '\0'), the >> operator will continue to store characters into memory past the end of the array.
2. The >> operator cannot be used to input a string that has blanks within it. (It stops reading as soon as it encounters the first whitespace character.)

To cope with these limitations, we can use a variation of the **get** function, a member of the **istream** class. We have used the **get** function to input a single character, even if it is a whitespace character:

```
cin.get(inputChar);
```

The **get** function also can be used to input C strings, in which case the function call requires two arguments. The first is the array name and the second is an **int** expression.

```
cin.get(myStr, charCount + 1);
```

The **get** function does not skip leading whitespace characters and continues until it either has read **charCount** characters or it reaches the newline character **'\n'**, whichever comes first. It then appends the null character to the end of the string. With the statements

```
char oneLine[81];   // Room for 80 characters plus '\0'
  ⋮
cin.get(oneLine, 81);
```

the **get** function reads and stores an entire input line (to a maximum of 80 characters), embedded blanks and all. If the line has fewer than 80 characters, reading stops at **'\n'** but does not consume it. The newline character is now the first one waiting in the input stream. To read two consecutive lines worth of strings, it is necessary to consume the newline character:

```
char dummy;
  ⋮
cin.get(string1, 81);
cin.get(dummy);          // Consume newline before next "get"
cin.get(string2, 81);
```

The first function call reads characters up to, but not including, the `'\n'`. If the input of **dummy** were omitted, then the input of **string2** would read *no* characters because `'\n'` would immediately be the first character waiting in the stream.

Finally, the **ignore** function can be useful in conjunction with the **get** function. The statement

```
cin.ignore(200, '\n');
```

says to skip at most 200 input characters but stop if a newline was read. (The newline character *is* consumed by this function.) If a program inputs a long string from the user but only wants to retain the first four characters of the response, here is a way to do it:

```
char response[5];         // Room for 4 characters plus '\0'

cin.get(response, 5);     // Input at most 4 characters
cin.ignore(100, '\n');    // Skip remaining chars up to and
                          //   including '\n'
```

The value 100 in the last statement is arbitrary. Any "large enough" number will do.

Here is a table that summarizes the differences between the >> operator and the **get** function when reading C strings:

Statement	Skips Leading Whitespace?	Stops Reading When?
`cin >> inputStr;`	Yes	At the first trailing whitespace character (which is not consumed)
`cin.get(inputStr, 21);`	No	When either 20 characters are read or '\n' is encountered (which is not consumed)

Certain library functions and member functions of system-supplied classes require C strings as arguments. An example is the **ifstream** class member function named **open**. To open a file, we pass the name of the file as a C string, either a constant or a variable:

```
ifstream file1;
ifstream file2;
char     fileName[51];            // Max. 50 characters plus '\0'

file1.open("students.dat");
cin.get(fileName, 51);            // Read at most 50 characters
cin.ignore(100, '\n');            // Skip rest of input line
file2.open(fileName);
```

If our file name is contained in a **string** class object, we still can use the **open** function, *provided* we use the **string** class member function named **c_str** to convert the string to a C string:

```
ifstream inFile;
string   fileName;

cin >> fileName;
inFile.open(fileName.c_str());
```

Comparing these two code segments, you can observe a major advantage of the **string** class over C strings: A string in a **string** class object has unbounded length, whereas the length of a C string is bounded by the array size, which is fixed at compile time.

H.3 C String Library Routines

Through the header file `cstring`, the C++ standard library provides a large assortment of C string operations. In this section, we discuss three of these library functions: `strlen`, which returns the length of a string; `strcmp`, which compares two strings using the relations less-than, equal, and greater-than; and `strcpy`, which copies one string to another. Here is a summary of `strlen`, `strcmp`, and `strcpy`:

Header File	Function	Function Value	Effect
`<cstring>`	`strlen(str)`	Integer length of `str` (excluding '\0')	Computes length of `str`
`<cstring>`	`strcmp(str1, str2)`	An integer < 0, if `str1 < str2`	Compares `str1` and `str2`
		The integer 0, if `str1 = str2`	
		An integer > 0, if `str1 > str2`	
`<cstring>`	`strcpy(toStr, fromStr)`	Base address of `toStr` (usually ignored)	Copies `fromStr` (including '\0') to `toStr`, overwriting what was there; `toStr` must be large enough to hold the result

The `strlen` function is similar to the `StrLength` function we wrote earlier. It returns the number of characters in a C string prior to the terminating '\0'. Here's an example of a call to the function:

```
#include <cstring>
  ⋮
char subject[] = "Computer Science";

cout << strlen(subject);   // Prints 16
```

The `strcpy` routine is important because aggregate assignment with the = operator is not allowed on C strings. In the following code fragment, we show the wrong way and the right way to perform a string copy.

```
#include <cstring>
  ⋮
char myStr[100];
  ⋮
myStr = "Abracadabra";           // No
strcpy(myStr, "Abracadabra");    // Yes
```

In `strcpy`'s argument list, the destination string is the one on the left, just as an assignment operation transfers data from right to left. It is the caller's responsibility to make sure that the destination array is large enough to hold the result.

The `strcpy` function is technically a value-returning function; it not only copies one C string to another, but also returns as a function value the base address of the destination array. The reason why the caller would want to use this function value is not at all obvious, and we don't discuss it here. Programmers nearly always ignore the function value and simply

invoke `strcpy` as if it were a void function (as we did above). You may wish to review the Background Information box in Chapter 9 entitled "Ignoring a Function Value."

The `strcmp` function is used for comparing two strings. The function receives two C strings as parameters and compares them in *lexicographic* order (the order in which they would appear in a dictionary)—the same ordering used in comparing `string` class objects. Given the function call `strcmp(str1, str2)`, the function returns one of the following `int` values: a negative integer, if `str1 < str2` lexicographically; the value 0, if `str1 = str2`; or a positive integer, if `str1 > str2`. The precise values of the negative integer and the positive integer are unspecified. You simply test to see if the result is less than 0, 0, or greater than 0. Here is an example:

```
if (strcmp(str1, str2) < 0)    // If str1 is less than str2 . . .
    ⋮
```

We have described only three of the string-handling routines provided by the standard library. These three are the most commonly needed, but there are many more. If you are designing or maintaining programs that use C strings extensively, you should read the documentation on strings for your C++ system.

H.4 String Class or C Strings?

When working with string data, should you use a class like `string`, or should you use C strings? From the standpoints of clarity, versatility, and ease of use, there is no contest. Use a `string` class. The standard library `string` class provides strings of unbounded length, aggregate assignment, aggregate comparison, concatenation with the + operator, and so forth.

However, it is still useful to be familiar with C strings. Among the thousands of software products currently in use that are written in C and C++, most (but a declining percentage) use C strings to represent string data. In your next place of employment, if you are asked to modify or upgrade such software, understanding C strings is essential.

Appendix I C++ `char` Constants

In C++, `char` constants come in two different forms. The first form is a single printable character enclosed by apostrophes (single quotes):

```
'A'   '8'   ')'   '+'
```

Notice that we said *printable* character. Character sets include both printable characters and *control* characters (or *nonprintable* characters). Control characters are not meant to be printed. In the early days of computing, including the period when the C language was first developed, the primary interface to the computer was often a very simple printer or display screen. Control characters were used to direct the actions of the screen, printer, and other hardware devices. With modern graphical user interfaces, the control mechanisms are more complex, and their coverage is beyond the scope of this text. However, C++ (and C) still preserve the simpler interface in the form input and output to a console window, or terminal program, that simulates the old-style hardware.

If you look at the listing of the ASCII character set in Appendix E, you will see that the printable characters are those with integer values 32–126. The remaining characters (with values 0–31 and 127) are nonprinting control characters.

To accommodate control characters, C++ provides a second form of `char` constant: the *escape sequence*. An escape sequence is one or more characters preceded by a backslash (\), for example, the escape sequence `\n` represents the newline character. Here is the complete description of the two forms of `char` constant in C++:

1. A single printable character—except an apostrophe (') or backslash (\)—enclosed by apostrophes.
2. One of the following escape sequences, enclosed by apostrophes:

`\n`	Newline
`\t`	Horizontal tab
`\v`	Vertical tab
`\b`	Backspace
`\r`	Carriage return
`\f`	Form feed
`\a`	Alert (a bell or beep sound)
`\\`	Backslash
`\'`	Single quote (apostrophe)
`\"`	`Double` quote (quotation mark)
`\0`	Null character (all 0 bits)
`\ddd`	Octal equivalent (one, two, or three octal digits specifying the integer value of the desired character)
`\xddd`	Hexadecimal equivalent (one or more hexadecimal digits specifying the integer value of the desired character)

Even though an escape sequence is written as two or more characters, each escape sequence represents a single character in the character set. The alert character (`\a`) is the same as what is called the BEL character in ASCII. On old mechanical printers, it actually rang a bell. These days, it causes the terminal program to make a beeping sound. You can output the alert character like this:

```
cout << '\a';
```

In the list of escape sequences above, the entries labeled *Octal equivalent* and *Hexadecimal equivalent* let you refer to any character in your machine's character set by specifying its integer value in either octal (base-8) or hexadecimal (base-16). The reason for using these seemingly strange number bases is that, being powers of two, they translate more directly into binary.

Note that you can use an escape sequence within a string just as you can use any printable character within a string. The statement

```
cout << "\aWhoops!\n";
```

beeps the beeper, displays `Whoops!`, and terminates the output line. The statement

```
cout << "She said \"Hi\"";
```

outputs

```
She said "Hi"
```

and does not terminate the output line.