Contents

Preface

This is the third edition of *C++: The Complete Reference*. In the years that have transpired since the second edition, C++ has undergone many changes. Perhaps the most important is that it is now a standardized language. In November of 1997, the ANSI/ISO committee charged with the task of standardizing C++, passed out of committee an International Standard for C++. This event marked the end of a very long, and at times contentious, process. As a member of the ANSI/ISO C++ committee, I watched the progress of the emerging standard, following each debate and argument. Near the end, there was a world-wide, daily dialogue, conducted via e-mail, in which the pros and cons of this or that issue were put forth, and finally resolved. While the process was longer and more exhausting than anyone at first envisioned, the result was worth the trouble. We now have a standard for what is, without question, the most important programming language in the world.

During standardization, several new features were added to C++. Some are relatively small. Others, like the STL (Standard Template Library) have ramifications that will affect the course of programming for years to come. The net effect of the additions was that the scope and range of the language were greatly expanded. For example, because of the addition of the numerics library, C++ can be more conveniently used for numeric processing. Of course, the information contained in this edition

support object-oriented programming (OOP). However, the C-like aspects of C++ were never abandoned, and the ANSI/ISO C standard is a *base document* for the International Standard for C++. Thus, an understanding of C++ implies an understanding of C.

In a book such as this *Complete Reference*, dividing the C++ language into two pieces—the C foundation and the C++-specific features—achieves three major benefits:

1. The dividing line between C and C++ is clearly delineated.
2. Readers already familiar with C can easily find the C++-specific information.
3. It provides a convenient place in which to discuss those features of C++ that relate mostly to the C subset.

Understanding the dividing line between C and C++ is important because both are widely used languages and it is very likely that you will be called upon to write or maintain both C and C++ code. When working on C code, you need to know where C ends and C++ begins. Many C++ programmers will, from time to time, be required to write code that is limited to the "C subset." This will be especially true for embedded systems programming and the maintenance of existing applications. Knowing the difference between C and C++ is simply part of being a top-notch professional C++ programmer.

A clear understanding of C is also valuable when converting C code into C++. To do this in a professional manner, a solid knowledge of C is required. For example, without a thorough understanding of the C I/O system, it is not possible to efficiently convert an I/O-intensive C program into C++.

Many readers already know C. Covering the C-like features of C++ in their own section makes it easier for the experienced C programmer to quickly and easily find information about C++ without having to wade through reams of information that he or she already knows. Of course, throughout Part One, any minor differences between C and C++ are noted. Also, separating the C foundation from the more advanced, object-oriented features of C++ makes it possible to tightly focus on those advanced features because all of the basics will have already been discussed.

Although C++ contains the entire C language, not all of the features provided by the C language are commonly used when writing "C++-style" programs. For example, the C I/O system is still available to the C++ programmer even though C++ defines its own, object-oriented version. The preprocessor is another example. The preprocessor is very important to C, but less so to C++. Discussing several of the "C-only" features in Part One prevents them from cluttering up the remainder of the book.

Remember *The C subset described in Part One constitutes the core of C++ and the foundation upon which C++'s object-oriented features are built. All the features described here are part of C++ and available for your use.*

Note *Part One of this book is adapted from my book* C: The Complete Reference *(Osborne/McGraw-Hill). If you are particularly interested in C, you will find this book helpful.*

- Compilers
- File utilities
- Performance enhancers
- Real-time executives

As C grew in popularity, many programmers began to use it to program all tasks because of its portability and efficiency—and because they liked it! At the time of its creation, C was a much longed-for, dramatic improvement in programming languages. Of course, C++ has carried on this tradition.

With the advent of C++, some thought that C as a distinct language would die out. Such has not been the case. First, not all programs require the application of the object-oriented programming features provided by C++. For example, applications such as embedded systems are still typically programmed in C. Second, much of the world still runs on C code, and those programs will continue to be enhanced and maintained. While C's greatest legacy is as the foundation for C++, it will continue to be a vibrant, widely used language for many years to come.

The Form of a C Program

Table 1-2 lists the 32 keywords that, combined with the formal C syntax, form the C programming language. Of these, 27 were defined by the original version of C. These five were added by the ANSI C committee: **enum**, **const**, **signed**, **void**, and **volatile**. All are, of course, part of the C++ language.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table 1-2. *The 32 Keywords Defined by Standard C*

You can apply the modifiers **signed**, **short**, **long**, and **unsigned** to integer base types. You can apply **unsigned** and **signed** to characters. You may also apply **long** to **double**. Table 2-1 shows all valid data type combinations, along with their minimal ranges and approximate bit widths. (These values also apply to a typical C++ implementation.) Remember, the table shows the *minimum range* that these types will have as specified by Standard C/C++, not their typical range. For example, on computers that use two's complement arithmetic (which is nearly all), an integer will have a range of at least 32,767 to –32,768.

The use of **signed** on integers is allowed, but redundant because the default integer declaration assumes a signed number. The most important use of **signed** is to modify **char** in implementations in which **char** is unsigned by default.

The difference between signed and unsigned integers is in the way that the high-order bit of the integer is interpreted. If you specify a signed integer, the compiler generates code that assumes that the high-order bit of an integer is to be used as a *sign flag*. If the sign flag is 0, the number is positive; if it is 1, the number is negative.

In general, negative numbers are represented using the *two's complement* approach, which reverses all bits in the number (except the sign flag), adds 1 to this number, and sets the sign flag to 1.

Signed integers are important for a great many algorithms, but they only have half the absolute magnitude of their unsigned relatives. For example, here is 32,767:

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

If the high-order bit were set to 1, the number would be interpreted as –1. However, if you declare this to be an **unsigned int**, the number becomes 65,535 when the high-order bit is set to 1.

Identifier Names

In C/C++, the names of variables, functions, labels, and various other user-defined objects are called *identifiers*. These identifiers can vary from one to several characters. The first character must be a letter or an underscore, and subsequent characters must be either letters, digits, or underscores. Here are some correct and incorrect identifier names:

Correct	Incorrect
Count	1count
test23	hi!there
high_balance	high...balance

In C, identifiers may be of any length. However, not all characters will necessarily be significant. If the identifier will be involved in an external link process, then at least the first six characters will be significant. These identifiers, called *external names*, include function names and global variables that are shared between files. If the identifier is not used in an external link process, then at least the first 31 characters will be significant. This type of identifier is called an *internal name* and includes the names of local variables, for example. In C++, there is no limit to the length of an identifier, and at least the first 1,024 characters are significant. This difference may be important if you are converting a program from C to C++.

In an identifier, upper- and lowercase are treated as distinct. Hence, **count**, **Count**, and **COUNT** are three separate identifiers.

An identifier cannot be the same as a C or C++ keyword, and should not have the same name as functions that are in the C or C++ library.

Variables

As you probably know, a *variable* is a named location in memory that is used to hold a value that may be modified by the program. All variables must be declared before they can be used. The general form of a declaration is

type variable_list;

Here, *type* must be a valid data type plus any modifiers, and *variable_list* may consist of one or more identifier names separated by commas. Here are some declarations:

```
int i,j,l;
short int si;
unsigned int ui;
double balance, profit, loss;
```

Remember, in C/C++ the name of a variable has nothing to do with its type.

Where Variables Are Declared

Variables will be declared in three basic places: inside functions, in the definition of function parameters, and outside of all functions. These are local variables, formal parameters, and global variables.

Local Variables

Variables that are declared inside a function are called *local variables*. In some C/C++ literature, these variables are referred to as *automatic* variables. This book uses the more

```
void sp_to_dash(const char *str)
{
  while(*str) {
    if(*str== ' ') printf("%c", '-');
    else printf("%c", *str);
    str++;
  }
}
```

If you had written **sp_to_dash( )** in such a way that the string would be modified, it would not compile. For example, if you had coded **sp_to_dash( )** as follows, you would receive a compile-time error:

```
/* This is wrong. */
void sp_to_dash(const char *str)
{
  while(*str) {
    if(*str==' ' ) *str = '-'; /* can't do this, str is const */
    printf("%c", *str);
    str++;
  }
}
```


Many functions in the standard library use **const** in their parameter declarations. For example, the **strlen( )** function has this prototype:

size_t strlen(const char *str);

Specifying *str* as **const** ensures that **strlen( )** will not modify the string pointed to by *str*. In general, when a standard library function has no need to modify an object pointed to by a calling argument, it is declared as **const.**

You can also use **const** to verify that your program does not modify a variable. Remember, a variable of type **const** can be modified by something outside your program. For example, a hardware device may set its value. However, by declaring a variable as **const**, you can prove that any changes to that variable occur because of external events.

volatile

The modifier **volatile** tells the compiler that a variable's value may be changed in ways not explicitly specified by the program. For example, a global variable's address may be passed to the operating system's clock routine and used to hold the real time of the

the outcome of a relational or logical operation is **true** or **false**. But since this automatically converts into 1 or 0, the distinction between C and C++ on this issue is mostly academic.

Table 2-5 shows the relational and logical operators. The truth table for the logical operators is shown here using 1's and 0's.

p	q	p && q	p || q	!p
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Both the relational and logical operators are lower in precedence than the arithmetic operators. That is, an expression like 10 > 1+12 is evaluated as if it were written 10 > (1+12). Of course, the result is false.

You can combine several operations together into one expression, as shown here:

```
10>5 && !(10<9) || 3<=4
```


Relational Operators

Operator	Action
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
= =	Equal
!=	Not equal

Logical Operators

Operator	Action
&&	AND
||	OR
!	NOT

Table 2-5. *Relational and Logical Operators*

```
!0 && 0 || 0
```

is false. However, when you add parentheses to the same expression, as shown here, the result is true:

```
!(0 && 0) || 0
```

Remember, all relational and logical expressions produce either a true or false result. Therefore, the following program fragment is not only correct, but will print the number 1.

```
int x;

x = 100;
printf("%d", x>10);
```

Bitwise Operators

Unlike many other languages, C/C++ supports a full complement of bitwise operators. Since C was designed to take the place of assembly language for most programming tasks, it needed to be able to support many operations that can be done in assembler, including operations on bits. *Bitwise operation* refers to testing, setting, or shifting the individual bits in a byte or word, which correspond to the **char** and **int** data types and variants. You cannot use bitwise operations on **float**, **double**, **long double**, **void**, **bool**, or other more complex types. Table 2-6 lists the operators that apply to bitwise operations. These operations are applied to the individual bits of the operands.

Operator	Action
&	AND
|	OR
^	Exclusive OR (XOR)
~	One's complement (NOT)

Table 2-6. *Bitwise Operators*

C/C++ defines (using **typedef**) a special type called **size_t**, which corresponds loosely to an unsigned integer. Technically, the value returned by **sizeof** is of type **size_t**. For all practical purposes, however, you can think of it (and use it) as if it were an unsigned integer value.

sizeof primarily helps to generate portable code that depends upon the size of the built-in data types. For example, imagine a database program that needs to store six integer values per record. If you want to port the database program to a variety of computers, you must not assume the size of an integer, but must determine its actual length using **sizeof**. This being the case, you could use the following routine to write a record to a disk file:

```
/* Write 6 integers to a disk file. */
void put_rec(int rec[6], FILE *fp)
{
  int len;

  len = fwrite(rec, sizeof(int)*6, 1, fp);
  if(len != 1) printf("Write Error");
}
```

Coded as shown, **put_rec( )** compiles and runs correctly in any environment, including those that use 16- and 32-bit integers.

One final point: **sizeof** is evaluated at compile time, and the value it produces is treated as a constant within your program.

The Comma Operator

The comma operator strings together several expressions. The left side of the comma operator is always evaluated as **void**. This means that the expression on the right side becomes the value of the total comma-separated expression. For example,

```
x = (y=3, y+1);
```

first assigns **y** the value 3 and then assigns **x** the value 4. The parentheses are necessary because the comma operator has a lower precedence than the assignment operator.

Essentially, the comma causes a sequence of operations. When you use it on the right side of an assignment statement, the value assigned is the value of the last expression of the comma-separated list.

The comma operator has somewhat the same meaning as the word "and" in normal English as used in the phrase "do this and this and this."

to **true** and any zero value converts to **false**, there is no practical difference between C and C++ on this point.

Selection Statements

C/C++ supports two types of selection statements: **if** and **switch**. In addition, the **?** operator is an alternative to **if** in certain circumstances.

if

The general form of the **if** statement is

```
if (expression) statement;
else statement;
```

where a *statement* may consist of a single statement, a block of statements, or nothing (in the case of empty statements). The **else** clause is optional.

If *expression* evaluates to true (anything other than 0), the statement or block that forms the target of **if** is executed; otherwise, the statement or block that is the target of **else** will be executed, if it exists. Remember, only the code associated with **if** or the code associated with **else** executes, never both.

In C, the conditional statement controlling **if** must produce a scalar result. A *scalar* is either an integer, character, pointer, or floating-point type. In C++, it may also be of type **bool**. It is rare to use a floating-point number to control a conditional statement because this slows execution time considerably. (It takes several instructions to perform a floating-point operation. It takes relatively few instructions to perform an integer or character operation.)

The following program contains an example of **if**. The program plays a very simple version of the "guess the magic number" game. It prints the message ** **Right** ** when the player guesses the magic number. It generates the magic number using the standard random number generator **rand( )**, which returns an arbitrary number between 0 and **RAND_MAX** (which defines an integer value that is 32,767 or larger). **rand( )** requires the header file **stdlib.h**. (A C++ program may also use the new-style header **<cstdlib>**.)

```
/* Magic number program #1. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* magic number */
```

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* magic number */
  int guess; /* user's guess */

  magic = rand(); /* generate the magic number */

  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("** Right ** ");
    printf("%d is the magic number", magic);
  }
  else if(guess > magic)
    printf("Wrong, too high");
  else printf("Wrong, too low");

  return 0;
}
```


The ? Alternative

You can use the **?** operator to replace **if-else** statements of the general form:

if(*condition*) *expression*;
else *expression*;

However, the target of both **if** and **else** must be a single expression—not another statement.

The **?** is called a *ternary operator* because it requires three operands. It takes the general form

Exp1 ? Exp2 : Exp3

where *Exp1*, *Exp2*, and *Exp3* are expressions. Notice the use and placement of the colon.

The value of a **?** expression is determined as follows: *Exp1* is evaluated. If it is true, *Exp2* is evaluated and becomes the value of the entire **?** expression. If *Exp1* is false, then *Exp3* is evaluated and its value becomes the value of the expression. For example, consider

```
  magic = rand(); /* generate the magic number */

  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("** Right ** ");
    printf("%d is the magic number", magic);
  }
  else
    guess > magic ? printf("High") : printf("Low");

  return 0;
}
```

Here, the **?** operator displays the proper message based on the outcome of the test **guess > magic**.

The Conditional Expression

Sometimes newcomers to C/C++ are confused by the fact that you can use any valid expression to control the **if** or the **?** operator. That is, you are not restricted to expressions involving the relational and logical operators (as is the case in languages like BASIC or Pascal). The expression must simply evaluate to either a true or false (zero or nonzero) value. For example, the following program reads two integers from the keyboard and displays the quotient. It uses an **if** statement, controlled by the second number, to avoid a divide-by-zero error.


```
/* Divide the first number by the second. */

#include <stdio.h>

int main(void)
{
  int a, b;

  printf("Enter two numbers: ");
  scanf("%d%d", &a, &b);

  if(b) printf("%d\n", a/b);
  else printf("Cannot divide by zero.\n");

  return 0;
}
```

First, **ch** is initialized to null. As a local variable, its value is not known when **wait_for_char( )** is executed. The **while** loop then checks to see if **ch** is not equal to **A**. Because **ch** was initialized to null, the test is true and the loop begins. Each time you press a key, the condition is tested again. Once you enter an **A**, the condition becomes false because **ch** equals **A**, and the loop terminates.

Like **for** loops, **while** loops check the test condition at the top of the loop, which means that the body of the loop will not execute if the condition is false to begin with. This feature may eliminate the need to perform a separate conditional test before the loop. The **pad( )** function provides a good illustration of this. It adds spaces to the end of a string to fill the string to a predefined length. If the string is already at the desired length, no spaces are added.

```
#include <stdio.h>
#include <string.h>

void pad(char *s, int length);

int main(void)
{
  char str[80];

  strcpy(str, "this is a test");
  pad(str, 40);
  printf("%d", strlen(str));
  return 0;
}

/* Add spaces to the end of a string. */
void pad(char *s, int length)
{
  int l;

  l = strlen(s); /* find out how long it is */

  while(l<length) {
    s[l] = ' '; /* insert a space */
    l++;
  }
  s[l]= '\0'; /* strings need to be
                 terminated in a null */
}
```

```
        matrix[0][i]==matrix[2][i]) return matrix[0][i];

  /* test diagonals */
  if(matrix[0][0]==matrix[1][1] &&
     matrix[1][1]==matrix[2][2])
       return matrix[0][0];

  if(matrix[0][2]==matrix[1][1] &&
     matrix[1][1]==matrix[2][0])
       return matrix[0][2];

  return ' ';
}
```

Arrays of Pointers

Pointers may be arrayed like any other data type. The declaration for an **int** pointer array of size 10 is

```
int *x[10];
```

To assign the address of an integer variable called **var** to the third element of the pointer array, write

```
x[2] = &var;
```

To find the value of **var**, write

```
*x[2]
```

If you want to pass an array of pointers into a function, you can use the same method that you use to pass other arrays—simply call the function with the array name without any indexes. For example, a function that can receive array x looks like this:

```
void display_array(int *q[])
{
  int t;

  for(t=0; t<10; t++)
    printf("%d ", *q[t]);
}
```


Remember, **q** is not a pointer to integers, but rather a pointer to an array of pointers to integers. Therefore you need to declare the parameter **q** as an array of integer pointers, as just shown. You cannot declare **q** simply as an integer pointer because that is not what it is.

Pointer arrays are often used to hold pointers to strings. You can create a function that outputs an error message given its code number, as shown here:

```
void syntax_error(int num)
{
  static char *err[] = {
    "Cannot Open File\n",
    "Read Error\n",
```

by other parts of the program. Stated another way, the code and data that are defined within one function cannot interact with the code or data defined in another function because the two functions have a different scope.

Variables that are defined within a function are called *local* variables. A local variable comes into existence when the function is entered and is destroyed upon exit. That is, local variables cannot hold their value between function calls. The only exception to this rule is when the variable is declared with the **static** storage class specifier. This causes the compiler to treat the variable as if it were a global variable for storage purposes, but limits its scope to within the function. (Chapter 2 covers global and local variables in depth.)

In C (and C++) you cannot define a function within a function. This is why neither C nor C++ are technically block-structured languages.

Function Arguments

If a function is to use arguments, it must declare variables that accept the value of the arguments. These variables are called the *formal parameters* of the function. They behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. As shown in the following function, the parameter declarations occur after the function name:

```
/* Return 1 if c is part of string s; 0 otherwise. */
int is_in(char *s,  char c)
{
  while(*s)
    if(*s==c) return 1;
    else s++;
  return 0;
}
```


The function **is_in( )** has two parameters: **s** and **c**. This function returns 1 if the character **c** is part of the string **s**; otherwise, it returns 0.

As with local variables, you may make assignments to a function's formal parameters or use them in an expression. Even though these variables perform the special task of receiving the value of the arguments passed to the function, you can use them as you do any other local variable.

Call by Value, Call by Reference

In a computer language, there are two ways that arguments can be passed to a subroutine. The first is known as *call by value*. This method copies the *value of* an

```
#include <stdio.h>

char *match(char c, char *s);  /* prototype */

int main(void)
{
  char s[80], *p, ch;

  gets(s);
  ch = getchar();
  p = match(ch, s);

  if(*p)  /* there is a match */
    printf("%s ", p);
  else
    printf("No match found.");

  return 0;
}
```

This program reads a string and then a character. If the character is in the string, the program prints the string from the point of match. Otherwise, it prints **No match found**.

Functions of Type void

One of **void**'s uses is to explicitly declare functions that do not return values. This prevents their use in any expression and helps avert accidental misuse. For example, the function **print_vertical( )** prints its string argument vertically down the side of the screen. Since it returns no value, it is declared as **void**.

```
void print_vertical(char *str)
{
  while(*str)
    printf("%c\n", *str++);
}
```

Here is an example that uses **print_vertical( )**.

```
#include <stdio.h>

void print_vertical(char *str);  /* prototype */
```

```
int main(int argc, char *argv[])
{
  if(argc > 1) print_vertical(argv[1]);

  return 0;
}

void print_vertical(char *str)
{
  while(*str)
    printf("%c\n", *str++);
}
```

One last point: Early versions of C did not define the **void** keyword. Thus, in early C programs, functions that did not return values simply defaulted to type **int**. Therefore, don't be surprised to see many examples of this in older code.

What Does main( ) Return?

The **main( )** function returns an integer to the calling process, which is generally the operating system. Returning a value from **main( )** is the equivalent of calling **exit( )** with the same value. If **main( )** does not explicitly return a value, the value passed to the calling process is technically undefined. In practice, most C/C++ compilers automatically return 0, but do not rely on this if portability is a concern.

Recursion

In C/C++, a function can call itself. A function is said to be *recursive* if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself, and is sometimes called *circular definition*.

A simple example of a recursive function is **factr( )**, which computes the factorial of an integer. The factorial of a number **n** is the product of all the whole numbers between 1 and **n**. For example, 3 factorial is 1 x 2 x 3, or 6. Both **factr( )** and its iterative equivalent are shown here:

```
/* recursive */
int factr(int n) {
  int answer;

  if(n==1) return(1);
  answer = factr(n-1)*n; /* recursive call */
  return(answer);
```

function, but without the overhead associated with a function call. For this reason, inline code is often used instead of function calls when execution time is critical.

Inline code is faster than a function call for two reasons. First, a CALL instruction takes time to execute. Second, if there are arguments to pass, these have to be placed on the stack, which also takes time. For most applications, this very slight increase in execution time is of no significance. But if it is, remember that each function call uses time that would be saved if the function's code were placed in line. For example, the following are two versions of a program that prints the square of the numbers from 1 to 10. The inline version runs faster than the other because the function call adds time.

in line

```
#include <stdio.h>


int main(void)
{
  int x;

  for(x=1; x<11; ++x)
    printf("%d", x*x);

  return 0;
}
```

function call

```
#include <stdio.h>
int sqr(int a);

int main(void)
{
  int x;

  for(x=1; x<11; ++x)
    printf("%d", sqr(x));

  return 0;
}

int sqr(int a)
{
  return a*a;
}
```


Note *In C++, the concept of inline functions is expanded and formalized. In fact, inline functions are an important component of the C++ language.*

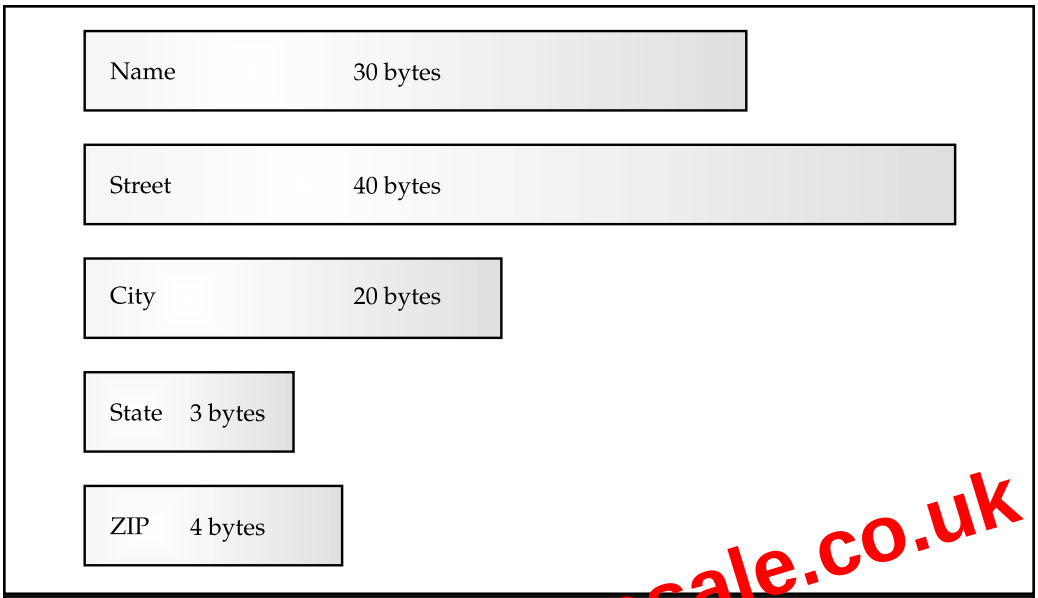


Figure 7-1. *The **addr_info** structure in memory*

If you only need one structure variable, the structure type name is not needed. That means that

```
struct {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info;
```

declares one variable named **addr_info** as defined by the structure preceding it.

The general form of a structure declaration is

struct *struct-type-name* {
  type member-name;
  type member-name;
  type member-name;
  .
  .
  .
} *structure-variables*;

where either *struct-type-name* or *structure-variables* may be omitted, but not both.

```
{
  struct {
    int a;
    int b;
  } x, y;

  x.a = 10;

  y = x;  /* assign one structure to another */

  printf("%d", y.a);

  return 0;
}
```

After the assignment, **y.a** will contain the value 10.

Arrays of Structures

Perhaps the most common usage of structures is in arrays of structures. To declare an array of structures, you must first define a structure and then declare an array variable of that type. For example, to declare a 100-element array of structures of type **addr**, defined earlier, write

```
struct addr addr_info[100];
```

This creates 100 sets of variables that are organized as defined in the structure **addr**.

To access a specific structure, index the structure name. For example, to print the ZIP code of structure 3, write

```
printf("%d", addr_info[2].zip);
```

Like all array variables, arrays of structures begin indexing at 0.

Passing Structures to Functions

This section discusses passing structures and their members to functions.

Passing Structure Members to Functions

When you pass a member of a structure to a function, you are actually passing the value of that member to the function. Therefore, you are passing a simple variable (unless, of course, that element is compound, such as an array). For example, consider this structure:

```
struct fred
{
  char x;
  int y;
  float z;
  char s[10];
} mike;
```

Here are examples of each member being passed to a function:

```
func(mike.x);    /* passes character value of x */
func2(mike.y);   /* passes integer value of y */
func3(mike.z);   /* passes float value of z */
func4(mike.s);   /* passes address of string s */
func(mike.s[2]); /* passes character value of s[2] */
```


If you wish to pass the *address* of an individual structure member, put the **&** operator before the structure name. For example, to pass the address of the members of the structure **mike**, write

```
func(&mike.x);    /* passes address of character x */
func2(&mike.y);   /* passes address of integer y */
func3(&mike.z);   /* passes address of float z */
func4(mike.s);    /* passes address of string s */
func(&mike.s[2]); /* passes address of character s[2] */
```

Remember that the **&** operator precedes the structure name, not the individual member name. Note also that **s** already signifies an address, so no **&** is required.

Passing Entire Structures to Functions

When a structure is used as an argument to a function, the entire structure is passed using the standard call-by-value method. Of course, this means that any changes

places the address of the structure **person** into the pointer **p**.

To access the members of a structure using a pointer to that structure, you must use the –> operator. For example, this references the **balance** field:

```
p->balance
```

The –> is usually called the *arrow operator*, and consists of the minus sign followed by a greater-than sign. The arrow is used in place of the dot operator when you are accessing a structure member through a pointer to the structure.

To see how a structure pointer can be used, examine this simple program, which prints the hours, minutes, and seconds on your screen using a software timer.

```
/* Display a software timer. */
#include <stdio.h>

#define DELAY 128000

struct my_time {
  int hours;
  int minutes;
  int seconds;
} ;

void display(struct my_time *t);
void update(struct my_time *t);
void delay(void);

int main(void)
{
  struct my_time systime;

  systime.hours = 0;
  systime.minutes = 0;
  systime.seconds = 0;

  for(;;) {
    update(&systime);
    display(&systime);
  }

  return 0;
}
```

```
void update(struct my_time *t)
{
  t->seconds++;
  if(t->seconds==60) {
    t->seconds = 0;
    t->minutes++;
  }

  if(t->minutes==60) {
    t->minutes = 0;
    t->hours++;
  }

  if(t->hours==24) t->hours = 0;
  delay();
}

void display(struct my_time *t)
{
  printf("%02d:", t->hours);
  printf("%02d:", t->minutes);
  printf("%02d\n", t->seconds);
}

void delay(void)
{
  long int t;

  /* change this as needed */
  for(t=1; t<DELAY; ++t) ;
}
```


The timing of this program is adjusted by changing the definition of **DELAY**.

As you can see, a global structure called **my_time** is defined but no variable is declared. Inside **main( )**, the structure **systime** is declared and initialized to 00:00:00. This means that **systime** is known directly only to the **main( )** function.

The functions **update( )** (which changes the time) and **display( )** (which prints the time) are passed the address of **systime**. In both functions, their arguments are declared as a pointer to a **my_time** structure.

Inside **update( )** and **display( )**, each member of **systime** is accessed via a pointer. Because **update( )** receives a pointer to the **systime** structure, it can update its value.

For example, to set the hours back to 0 when 24:00:00 is reached, **update( )** contains this line of code:

```
if(t->hours==24) t->hours = 0;
```

This tells the compiler to take the address of **t** (which points to **systime** in **main( )**) and to reset **hours** to zero.

Remember, use the dot operator to access structure elements when operating on the structure itself. When you have a pointer to a structure, use the arrow operator.

Arrays and Structures Within Structures

A member of a structure may be either a simple or compound type. A simple member is one that is of any of the built-in data types, such as integer or character. You have already seen one type of compound element: the character arrays used in **addr**. Other compound data types include one-dimensional and multidimensional arrays of the other data types and structures.

A member of a structure that is an array is treated as you might expect from the earlier examples. For example, consider this structure:

```
struct x {
  int a[10][10]; /* 10 x 10 array of ints */
  float b;
} y;
```


To reference integer 3,7 in **a** of structure **y**, write

```
y.a[3][7]
```

When a structure is a member of another structure, it is called a *nested structure*. For example, the structure **address** is nested inside **emp** in this example:

```
struct emp {
  struct addr address; /* nested structure */
  float wage;
} worker;
```

Here, structure **emp** has been defined as having two members. The first is a structure of type **addr**, which contains an employee's address. The other is **wage**, which holds the employee's wage. The following code fragment assigns 93456 to the **zip** element of **address**.

Bit	Meaning When Set
0	Change in clear-to-send line
1	Change in data-set-ready
2	Trailing edge detected
3	Change in receive line
4	Clear-to-send
5	Data-set-ready
6	Telephone ringing
7	Received signal

You can represent the information in a status byte using the following bit-field:

```
struct status_type {
  unsigned delta_cts: 1;
  unsigned delta_dsr: 1;
  unsigned tr_edge:   1;
  unsigned delta_rec: 1;
  unsigned cts:       1;
  unsigned dsr:       1;
  unsigned ring:      1;
  unsigned rec_line:  1;
} status;
```


You might use a routine similar to that shown here to enable a program to determine when it can send or receive data.

```
status = get_port_status();
if(status.cts) printf("clear to send");
if(status.dsr) printf("data ready");
```

To assign a value to a bit-field, simply use the form you would use for any other type of structure element. For example, this code fragment clears the **ring** field:

```
status.ring = 0;
```

As you can see from this example, each bit-field is accessed with the dot operator. However, if the structure is referenced through a pointer, you must use the –> operator.

```
struct s {
  char ch;
  int i;
  double f;
} s_var;
```

Here, **sizeof(s_var)** is at least 13 (8 + 4 + 1). However, the size of **s_var** might be greater because the compiler is allowed to pad a structure in order to achieve word or paragraph alignment. (A paragraph is 16 bytes.) Since the size of a structure may be greater than the sum of the sizes of its members, you should always use **sizeof** when you need to know the size of a structure.

Since **sizeof** is a compile-time operator, all the information necessary to compute the size of any variable is known at compile time. This is especially meaningful for **union**s, because the size of a **union** is always equal to the size of its largest member. For example, consider

```
union u {
  char ch;
  int i;
  double f;
} u_var;
```


Here, the **sizeof(u_var)** is 8. At run time, it does not matter what **u_var** is actually holding. All that matters is the size of its largest member, because any **union** must be as large as its largest element.

typedef

You can define new data type names by using the keyword **typedef**. You are not actually *creating* a new data type, but rather defining a new name for an existing type. This process can help make machine-dependent programs more portable. If you define your own type name for each machine-dependent data type used by your program, then only the **typedef** statements have to be changed when compiling for a new environment. **typedef** also can aid in self-documenting your code by allowing descriptive names for the standard data types. The general form of the **typedef** statement is

Code	Format
%i	Signed decimal integers
%e	Scientific notation (lowercase e)
%E	Scientific notation (uppercase E)
%f	Decimal floating point
%g	Uses %e or %f, whichever is shorter
%G	Uses %E or %F, whichever is shorter
%o	Unsigned octal
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%X	Unsigned hexadecimal (uppercase letters)
%p	Displays a pointer
%n	The associated argument must be a pointer to an integer. This specifier causes the number of characters written so far to be put into that integer.
%%	Prints a % sign

Table 8-2. ***printf( )*** *Format Specifiers* (continued)

Printing Characters

To print an individual character, use **%c**. This causes its matching argument to be output, unmodified, to the screen.

To print a string, use **%s**.

Printing Numbers

You may use either **%d** or **%i** to indicate a signed decimal number. These format specifiers are equivalent; both are supported for historical reasons.

To output an unsigned value, use **%u**.

The **%f** format specifier displays numbers in floating point.

```
#include <stdio.h>

int main(void)
{
  int i;

  /* display a table of squares and cubes */
  for(i=1; i<20; i++)
    printf("%8d %8d %8d\n", i, i*i, i*i*i);

  return 0;
}
```

A sample of its output is shown here:

```
       1        1        1
       2        4        8
       3        9       27
       4       16       64
       5       25      125
       6       36      216
       7       49      343
       8       64      512
       9       81      729
      10      100     1000
      11      121     1331
      12      144     1728
      13      169     2197
      14      196     2744
      15      225     3375
      16      256     4096
      17      289     4913
      18      324     5832
      19      361     6859
```


The Precision Specifier

The *precision specifier* follows the minimum field width specifier (if there is one). It consists of a period followed by an integer. Its exact meaning depends upon the type of data it is applied to.

When you apply the precision specifier to floating-point data using the **%f**, **%e**, or **%E** specifiers, it determines the number of decimal places displayed. For example,

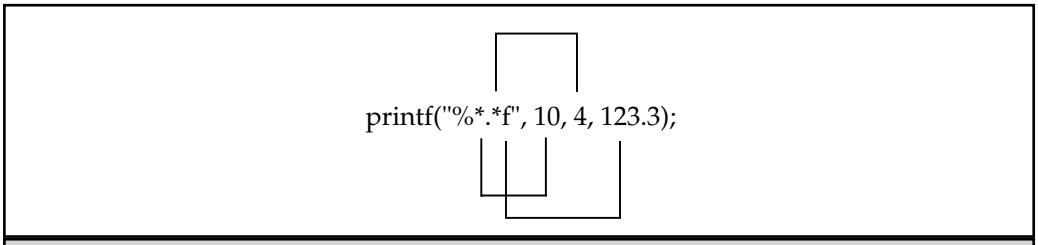


Figure 8-1. *How the * is matched to its value*

scanf( )

scanf( ) is the general-purpose console input routine. It can read all the built-in data types and automatically convert numbers into the proper internal format. It is much like the reverse of **printf( )**. The prototype for **scanf( )** is

```
int scanf(const char *control_string, ...);
```

The **scanf( )** function returns the number of data items successfully assigned a value. If an error occurs, **scanf( )** returns **EOF**. The *control_string* determines how values are read into the variables pointed to in the argument list.

The control string consists of three classifications of characters:

- Format specifiers
- White-space characters
- Non-white-space characters

Let's take a look at each of these now.

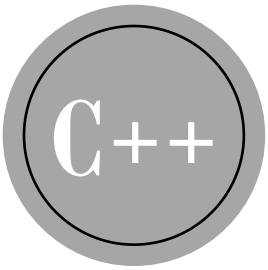
Format Specifiers

The input format specifiers are preceded by a % sign and tell **scanf( )** what type of data is to be read next. These codes are listed in Table 8-3. The format specifiers are matched, in order from left to right, with the arguments in the argument list. Let's look at some examples.

Inputting Numbers

To read an integer, use either the **%d** or **%i** specifier. To read a floating-point number represented in either standard or scientific notation, use **%e**, **%f**, or **%g**.

You can use **scanf( )** to read integers in either octal or hexadecimal form by using the **%o** and **%x** format commands, respectively. The **%x** may be in either upper- or



Chapter 9

File I/O

This chapter describes the C file system. As explained in Chapter 8, C++ supports two complete I/O systems: the one inherited from C and the object-oriented system defined by C++. This chapter covers the C file system. (The C++ file system is discussed in Part Two.) While most new code will use the C++ file system, knowledge of the C file system is still important for the reasons given in the preceding chapter.

C Versus C++ File I/O

There is sometimes confusion over how C's file system relates to C++. First, C++ supports the entire Standard C file system. Thus, if you will be porting older C code to C++, you will not have to change all of your I/O routines right away. Second, C++ defines its own, object-oriented I/O system, which includes both I/O functions and I/O operators. The C++ I/O system completely duplicates the functionality of the C I/O system and renders the C file system redundant. While you will usually want to use the C++ I/O system, you are free to use the C file system if you like. Of course, most C++ programmers elect to use the C++ I/O system for reasons that are made clear in Part Two of this book.

Streams and Files

Before beginning a discussion of the C file system, it is necessary to know the difference between the terms *streams* and *files*. The C I/O system supplies a consistent interface to the programmer independent of the actual device being accessed. That is, the C I/O system provides a level of abstraction between the programmer and the device. This abstraction is called a *stream* and the actual device is called a *file*. It is important to understand how streams and files interact.

Note *The concept of streams and files is also important to the C++ I/O system discussed in Part Two.*

Streams

The C file system is designed to work with a wide variety of devices, including terminals, disk drives, and tape drives. Even though each device is very different, the buffered file system transforms each into a logical device called a stream. All streams behave similarly. Because streams are largely device independent, the same function that can write to a disk file can also be used to write to another type of device, such as the console. There are two types of streams: text and binary.

If you are new to programming, the separation of streams and files may seem unnecessary or contrived. Just remember that its main purpose is to provide a consistent interface. You need only think in terms of streams and use only one file system to accomplish all I/O operations. The I/O system automatically converts the raw input or output from each device into an easily managed stream.

File System Basics

The C file system is composed of several interrelated functions. The most common of these are shown in Table 9-1. They require the header **stdio.h**. C++ programs may also use the new-style header **<cstdio>**.

Name	Function
fopen( )	Opens a file.
fclose( )	Closes a file.
putc( )	Writes a character to a file.
fputc( )	Same as **putc( )**.
getc( )	Reads a character from a file.
fgetc( )	Same as **getc( )**.
fgets( )	Reads a string from a file.
fputs( )	Writes a string to a file.
fseek( )	Seeks to a specified byte in a file.
ftell( )	Returns the current file position.
fprintf( )	Is to a file what **printf( )** is to the console.
fscanf( )	Is to a file what **scanf( )** is to the console.
feof( )	Returns true if end-of-file is reached.
ferror( )	Returns true if an error has occurred.
rewind( )	Resets the file position indicator to the beginning of the file.
remove( )	Erases a file.
fflush( )	Flushes a file.

Table 9-1. *Commonly Used C File-System Functions*

and **fclose( )**. It reads characters from the keyboard and writes them to a disk file until the user types a dollar sign. The filename is specified from the command line. For example, if you call this program KTOD, typing **KTOD TEST** allows you to enter lines of text into the file called TEST.

```
/* KTOD: A key to disk program. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char ch;

  if(argc!=2) {
    printf("You forgot to enter the filename.\n");
    exit(1);
  }

  if((fp=fopen(argv[1], "w"))==NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  do {
    ch = getchar();
    putc(ch, fp);
  } while (ch != '$');

  fclose(fp);

  return 0;
}
```


The complementary program DTOS reads any text file and displays the contents on the screen.

```
/* DTOS: A program that reads files and displays them
         on the screen. */
#include <stdio.h>
#include <stdlib.h>
```

problems, the C file system includes the function **feof( )**, which determines when the end of the file has been encountered. The **feof( )** function has this prototype:

```
int feof(FILE *fp);
```

feof( ) returns true if the end of the file has been reached; otherwise, it returns 0. Therefore, the following routine reads a binary file until the end of the file is encountered:

```
while(!feof(fp)) ch = getc(fp);
```

Of course, you can apply this method to text files as well as binary files.

The following program, which copies text or binary files, contains an example of **feof( )**. The files are opened in binary mode and **feof( )** checks for the end of the file.

```
/* Copy a file. */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  FILE *in, *out;
  char ch;

  if(argc!=3) {
    printf("You forgot to enter a filename.\n");
    exit(1);
  }

  if((in=fopen(argv[1], "rb"))==NULL) {
    printf("Cannot open source file.\n");
    exit(1);
  }
  if((out=fopen(argv[2], "wb")) == NULL) {
    printf("Cannot open destination file.\n");
    exit(1);
  }

  /* This code actually copies the file. */
  while(!feof(in)) {
    ch = getc(in);
```

The Console I/O Connection

Recall from Chapter 8 that there is little distinction between console I/O and file I/O. The console I/O functions described in Chapter 8 actually direct their I/O operations to either **stdin** or **stdout**. In essence, the console I/O functions are simply special versions of their parallel file functions. The reason they exist is as a convenience to you, the programmer.

As described in the previous section, you can perform console I/O using any of the file system functions. However, what might surprise you is that you can perform disk file I/O using console I/O functions, such as **printf( )**! This is because all of the console I/O functions operate on **stdin** and **stdout**. In environments that allow redirection of I/O, this means that **stdin** and **stdout** could refer to a device other than the keyboard and screen. For example, consider this program:

```
#include <stdio.h>

int main(void)
{
  char str[80];

  printf("Enter a string: ");
  gets(str);
  printf(str);

  return 0;
}
```


Assume that this program is called TEST. If you execute TEST normally, it displays its prompt on the screen, reads a string from the keyboard, and displays that string on the display. However, in an environment that supports I/O redirection, either **stdin**, **stdout**, or both could be redirected to a file. For example, in a DOS or Windows environment, executing TEST like this:

```
TEST > OUTPUT
```

causes the output of TEST to be written to a file called OUTPUT. Executing TEST like this:

```
TEST < INPUT > OUTPUT
```

directs **stdin** to the file called INPUT and sends output to the file called OUTPUT.

When a program terminates, any redirected streams are reset to their default status.

You can include various instructions to the compiler in the source code of a C/C++ program. These are called *preprocessor directives*, and although not actually part of the C or C++ language per se, they expand the scope of the programming environment. This chapter also examines comments.

The Preprocessor

Before beginning, it is important to put the preprocessor in historical perspective. As it relates to C++, the preprocessor is largely a holdover from C. Moreover, the C++ preprocessor is virtually identical to the one defined by C. The main difference between C and C++ in this regard is the degree to which each relies upon the preprocessor. In C, each preprocessor directive is necessary. In C++, some features have been rendered redundant by newer and better C++ language elements. In fact, one of the long-term design goals of C++ is the elimination of the preprocessor altogether. But for now and well into the foreseeable future, the preprocessor will still be widely used.

The preprocessor contains the following directives:

#define	#elif	#else	#endif
#error	#if	#ifdef	#ifndef
#include	#line	#pragma	#undef

As you can see, all preprocessor directives begin with a # sign. In addition, each preprocessing directive must be on its own line. For example,

```
#include <stdio.h>  #include <stdlib.h>
```

will not work.

#define

The **#define** directive defines an identifier and a character sequence (i.e., a set of characters) that will be substituted for the identifier each time it is encountered in the source file. The identifier is referred to as a *macro name* and the replacement process as *macro replacement*. The general form of the directive is

#define *macro-name char-sequence*

#include

The **#include** directive instructs the compiler to read another source file in addition to the one that contains the **#include** directive. The name of the additional source file must be enclosed between double quotes or angle brackets. For example,

```
#include "stdio.h"
#include <stdio.h>
```

both instruct the compiler to read and compile the header for the C I/O system library functions.

Include files can have **#include** directives in them. This is referred to as *nested includes*. The number of levels of nesting allowed varies between compilers. However, Standard C stipulates that at least eight nested inclusions will be available. Standard C++ recommends that at least 256 levels of nesting be supported.

Whether the filename is enclosed by quotes or by angle brackets determines how the search for the specified file is conducted. If the filename is enclosed in angle brackets, the file is searched for in a manner defined by the creator of the compiler. Often, this means searching some special directory reserved for include files. If the filename is enclosed in quotes, the file is searched for in another implementation-defined manner. For many compilers, this means searching the current working directory. If the file is not found, the search is repeated as if the filename had been enclosed in angle brackets.

Typically, most programmers use angle brackets to include the standard header files. The use of quotes is generally reserved for including files specifically related to the program at hand. However, there is no hard and fast rule that demands this usage.

In addition to *files*, a C++ program can use the **#include** directive to include a C++ *header*. C++ defines a set of standard headers that provide the information necessary to the various C++ libraries. A header is a standard identifier that might, but need not, map to a filename. Thus, a header is simply an abstraction that guarantees that the appropriate information required by your program is included. Various issues associated with headers are described in Part Two.

Conditional Compilation Directives

There are several directives that allow you to selectively compile portions of your program's source code. This process is called *conditional compilation* and is used widely by commercial software houses that provide and maintain many customized versions of one program.

One reason for using **defined** is that it allows the existence of a macro name to be determined by a **#elif** statement.

#line

The **#line** directive changes the contents of __LINE__ and __FILE__, which are predefined identifiers in the compiler. The __LINE__ identifier contains the line number of the currently compiled line of code. The __FILE__ identifier is a string that contains the name of the source file being compiled. The general form for **#line** is

#line *number* "*filename*"

where *number* is any positive integer and becomes the new value of __LINE__, and the optional *filename* is any valid file identifier, which becomes the new value of __FILE__. **#line** is primarily used for debugging and special applications.

For example, the following code specifies that the line count will begin with 100. The **printf( )** statement displays the number 102 because it is the third line in the program after the **#line 100** statement.

```
#include <stdio.h>

#line 100                  /* reset the line counter */
int main(void)             /* line 100 */
{                          /* line 101 */
  printf("%d\n",__LINE__); /* line 102 */

  return 0;
}
```


#pragma

#pragma is an implementation-defined directive that allows various instructions to be given to the compiler. For example, a compiler may have an option that supports program execution tracing. A trace option would then be specified by a **#pragma** statement. You must check the compiler's documentation for details and options.

The # and ## Preprocessor Operators

There are two preprocessor operators: # and **##.** These operators are used with the **#define** statement.

worked. Object-oriented methods were created to help programmers break through these barriers.

Object-oriented programming took the best ideas of structured programming and combined them with several new concepts. The result was a different way of organizing a program. In the most general sense, a program can be organized in one of two ways: around its code (what is happening) or around its data (who is being affected). Using only structured programming techniques, programs are typically organized around code. This approach can be thought of as "code acting on data." For example, a program written in a structured language such as C is defined by its functions, any of which may operate on any type of data used by the program.

Object-oriented programs work the other way around. They are organized around data, with the key principle being "data controlling access to code." In an object-oriented language, you define the data and the routines that are permitted to act on that data. Thus, a data type defines precisely what sort of operations can be applied to that data.

To support the principles of object-oriented programming, all OOP languages have three traits in common: encapsulation, polymorphism, and inheritance. Let's examine each.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an *object* is created. In other words, an object is the device that supports encapsulation.

Within an object, code, data, or both may be *private* to that object or *public*. Private code or data is known to and accessible only by another part of the object. That is, private code or data may not be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program may access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object.

For all intents and purposes, an object is a variable of a user-defined type. It may seem strange that an object that links both code and data can be thought of as a variable. However, in object-oriented programming, this is precisely the case. Each time you define a new type of object, you are creating a new data type. Each specific instance of this data type is a compound variable.

Polymorphism

Object-oriented programming languages support *polymorphism*, which is characterized by the phrase "one interface, multiple methods." In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The

name **iostream**. The reason is that **<iostream>** is one of the new-style headers defined by Standard C++. New-style headers do not use the **.h** extension.

The next line in the program is

```
using namespace std;
```

This tells the compiler to use the **std** namespace. Namespaces are a recent addition to C++. A namespace creates a declarative region in which various program elements can be placed. Namespaces help in the organization of large programs. The **using** statement informs the compiler that you want to use the **std** namespace. This is the namespace in which the entire Standard C++ library is declared. By using the **std** namespace you simplify access to the standard library. The programs in Part One, which use only the C subset, don't need a namespace statement because the C library functions are also available in the default, global namespace.

Note *Since both new-style headers and namespaces are recent additions to C++, you may encounter older code that does not use them. Also, if you are using an older compiler, it may not support them. Instructions for using an older compiler are found later in this chapter.*

Now examine the following line:

```
int main()
```

Notice that the parameter list in **main( )** is empty. In C++, this indicates that **main( )** has no parameters. This differs from C. In C, a function that has no parameters must use **void** in its parameter list, as shown here:

```
int main(void)
```

This was the way **main( )** was declared in the programs in Part One. However, in C++, the use of **void** is redundant and unnecessary. As a general rule, in C++ when a function takes no parameters, its parameter list is simply empty; the use of **void** is not required.

The next line contains two C++ features.

```
cout << "This is output.\n";  // this is a single line comment
```

First, the statement

```
cout << "This is output.\n";
```

```
int main()
{
  return 0;
}
```

This version uses the new-style header and specifies a namespace. Both of these features were mentioned in passing earlier. Let's look closely at them now.

The New C++ Headers

As you know, when you use a library function in a program, you must include its header file. This is done using the **#include** statement. For example, in C, to include the header file for the I/O functions, you include **stdio.h** with a statement like this:

```
#include <stdio.h>
```

Here, **stdio.h** is the name of the file used by the I/O functions, and the preceding statement causes that file to be included in your program. The key point is that this **#include** statement *includes a file*.

When C++ was first invented and for several years after that it used the same style of headers as did C. That is, it used *header files*. In fact, Standard C++ still supports C-style headers for header files that you create and for backward compatibility. However, Standard C++ created a new kind of header that is used by the Standard C++ library. The new-style headers *do not* specify filenames. Instead, they simply specify standard identifiers that may be mapped to files by the compiler, although they need not be. The new-style C++ headers are an abstraction that simply guarantee that the appropriate prototypes and definitions required by the C++ library have been declared.

Since the new-style headers are not filenames, they do not have a **.h** extension. They consist solely of the header name contained between angle brackets. For example, here are some of the new-style headers supported by Standard C++.

<iostream> <fstream> <vector> <string>

The new-style headers are included using the **#include** statement. The only difference is that the new-style headers do not necessarily represent filenames.

Because C++ includes the entire C function library, it still supports the standard C-style header files associated with that library. That is, header files such as **stdio.h** or **ctype.h** are still available. However, Standard C++ also defines new-style headers that you can use in place of these header files. The C++ versions of the C standard headers simply add a "c" prefix to the filename and drop the **.h**. For example, the C++ new-style header for **math.h** is **<cmath>**. The one for **string.h** is **<cstring>**. Although it

Working with an Old Compiler

As explained, both namespaces and the new-style headers are fairly recent additions to the C++ language, added during standardization. While all new C++ compilers support these features, older compilers may not. When this is the case, your compiler will report one or more errors when it tries to compile the first two lines of the sample programs in this book. If this is the case, there is an easy work-around: simply use an old-style header and delete the **namespace** statement. That is, just replace

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

with

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

This change transforms a modern program into an old-style one. Since the old-style header reads all of its contents into the global namespace, there is no need for a **namespace** statement.

One other point: for now and for the next few years, you will see many C++ programs that use the old-style headers and do not include a **using** statement. Your C++ compiler will be able to compile them just fine. However, for new programs, you should use the modern style because it is the only style of program that complies with the C++ standard. While old-style programs will continue to be supported for many years, they are technically noncompliant.

Introducing C++ Classes

This section introduces C++'s most important feature: the class. In C++, to create an object, you first must define its general form by using the keyword **class**. A **class** is similar syntactically to a structure. Here is an example. The following class defines a type called **stack**, which will be used to create a stack:

```
#define SIZE 100

// This creates the class stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  void init();
```

```
  void push(int i);
  int pop();
};
```

A **class** may contain private as well as public parts. By default, all items defined in a **class** are private. For example, the variables **stck** and **tos** are private. This means that they cannot be accessed by any function that is not a member of the **class**. This is one way that encapsulation is achieved—access to certain items of data may be tightly controlled by keeping them private. Although it is not shown in this example, you can also define private functions, which then may be called only by other members of the **class**.

To make parts of a **class** public (that is, accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after **public** can be accessed by all other functions in the program. Essentially, the rest of your program accesses an object through its public functions. Although you can have public variables, good practice dictates that you should try to limit their use. Instead, you should make all data private and control access to it through public functions. One other point: Notice that the **public** keyword is followed by a colon.

The functions **init( ), push( ),** and **pop( )** are called *member functions* because they are part of the class **stack**. The variables **stck** and **tos** are called *member variables* (or *data members*). Remember, an object forms a bond between code and data. Only member functions have access to the private members of their class. Thus, only **init( ), push( )**, and **pop( )** may access **stck** and **tos.**

Once you have defined a **class**, you can create an object of that type by using the class name. In essence, the class name becomes a new data type specifier. For example, this creates an object called **mystack** of type **stack**:

```
stack mystack;
```

When you declare an object of a class, you are creating an *instance* of that class. In this case, **mystack** is an instance of **stack**. You may also create objects when the **class** is defined by putting their names after the closing curly brace, in exactly the same way as you would with a structure.

To review: In C++, **class** creates a new data type that may be used to create objects of that type. Therefore, an object is an instance of a class in just the same way that some other variable is an instance of the **int** data type, for example. Put differently, a class is a logical abstraction, while an object is real. (That is, an object exists inside the memory of the computer.)

The general form of a simple **class** declaration is

class *class-name* {
  private data and functions

Operator Overloading

Polymorphism is also achieved in C++ through operator overloading. As you know, in C++, it is possible to use the << and >> operators to perform console I/O operations. They can perform these extra operations because in the **<iostream>** header, these operators are overloaded. When an operator is overloaded, it takes on an additional meaning relative to a certain class. However, it still retains all of its old meanings.

In general, you can overload most of C++'s operators by defining what they mean relative to a specific class. For example, think back to the **stack** class developed earlier in this chapter. It is possible to overload the **+** operator relative to objects of type **stack** so that it appends the contents of one stack to the contents of another. However, the **+** still retains its original meaning relative to other types of data.

Because operator overloading is, in practice, somewhat more complex than function overloading, examples are deferred until Chapter 14.

Inheritance

As stated earlier in this chapter, inheritance is one of the major traits of an object-oriented programming language. In C++, inheritance is supported by allowing one class to incorporate another class into its declaration. Inheritance allows a hierarchy of classes to be built, moving from most general to most specific. The process involves first defining a *base class*, which defines those qualities common to all objects to be derived from the base. The base class represents the most general description. The classes that are derived from the base are usually referred to as *derived classes*. A derived class includes all features of the generic base class and then adds qualities specific to the derived class. To demonstrate how this works, the next example creates classes that categorize different types of buildings.

To begin, the **building** class is declared, as shown here. It will serve as the base for two derived classes.

```
class building {
  int rooms;
  int floors;
  int area;
public:
  void set_rooms(int num);
  int get_rooms();
  void set_floors(int num);
  int get_floors();
  void set_area(int num);
  int get_area();
};
```

Because (for the sake of this example) all buildings have three common features—one or more rooms, one or more floors, and a total area—the **building** class embodies these components into its declaration. The member functions beginning with **set** set the values of the private data. The functions starting with **get** return those values.

You can now use this broad definition of a building to create derived classes that describe specific types of buildings. For example, here is a derived class called **house:**

```
// house is derived from building
class house : public building {
  int bedrooms;
  int baths;
public:
  void set_bedrooms(int num);
  int get_bedrooms();
  void set_baths(int num);
  int get_baths();
};
```

Notice how **building** is inherited. The general form for inheritance is

class *derived-class access base-class* {
  // body of derived class
}

Here, *access* is optional. However, if present, it must be **public, private,** or **protected**. (These options are further examined in Chapter 12.) For now, all inherited classes will use **public**. Using **public** means that all of the public members of the base class will become public members of the derived class. Therefore, the public members of the class **building** become public members of the derived class **house** and are available to the member functions of **house** just as if they had been declared inside **house**. However, **house**'s member functions *do not* have access to the private elements of **building**. This is an important point. Even though **house** inherits **building**, it has access only to the public members of **building**. In this way, inheritance does not circumvent the principles of encapsulation necessary to OOP.

A derived class has direct access to both its own members and the public members of the base class.

Here is a program illustrating inheritance. It creates two derived classes of **building** using inheritance; one is **house**, the other, **school**.

```
public:
  stack();  // constructor
  ~stack(); // destructor
  void push(int i);
  int pop();
};

// stack's constructor function
stack::stack()
{
  tos = 0;
  cout << "Stack Initialized\n";
}

// stack's destructor function
stack::~stack()
{
  cout << "Stack Destroyed\n";
}
```

Notice that, like constructor functions, destructor functions do not have return values.

To see how constructors and destructors work, here is a new version of the **stack** program examined earlier in this chapter. Observe that **init( )** is no longer needed.


```
// Using a constructor and destructor.
#include <iostream>
using namespace std;

#define SIZE 100

// This creates the class stack.
class stack {
  int stck[SIZE];
  int tos;
public:
  stack();  // constructor
  ~stack(); // destructor
  void push(int i);
  int pop();
};

// stack's constructor function
stack::stack()
```

```
{
  tos = 0;
  cout << "Stack Initialized\n";
}

// stack's destructor function
stack::~stack()
{
  cout << "Stack Destroyed\n";
}

void stack::push(int i)
{
  if(tos==SIZE) {
    cout << "Stack is full.\n";
    return;
  }
  stck[tos] = i;
  tos++;
}

int stack::pop()
{
  if(tos==0) {
    cout << "Stack underflow.\n";
    return 0;
  }
  tos--;
  return stck[tos];
}

int main()
{
  stack a, b;  // create two stack objects

  a.push(1);
  b.push(2);

  a.push(3);
  b.push(4);

  cout << a.pop() << " ";
```

public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	wchar_t	while	

Table 11-1. *The C++ keywords* (continued)

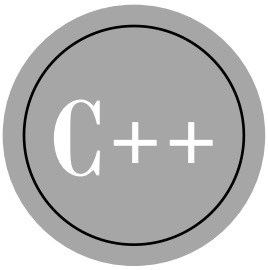
The General Form of a C++ Program

Although individual styles will differ, most C++ programs will have this general form:

```
#includes
base-class declarations
derived class declarations
nonmember function prototypes
int main( )
{
    //...
}
nonmember function definitions
```

In most large projects, all **class** declarations will be put into a header file and included with each module. But the general organization of a program remains the same.

The remaining chapters in this section examine in greater detail the features discussed in this chapter, as well as all other aspects of C++.



Chapter 12

Classes and Objects

```
  unsigned short u;
  unsigned char c[2];
};

void swap_byte::swap()
{
  unsigned char t;

  t = c[0];
  c[0] = c[1];
  c[1] = t;
}

void swap_byte::show_word()
{
  cout << u;
}

void swap_byte::set_byte(unsigned short i)
{
  u = i;
}

int main()
{
  swap_byte b;

  b.set_byte(49034);
  b.swap();
  b.show_word();

  return 0;
}
```


Like a structure, a **union** declaration in C++ defines a special type of class. This means that the principle of encapsulation is preserved.

There are several restrictions that must be observed when you use C++ unions. First, a **union** cannot inherit any other classes of any type. Further, a **union** cannot be a base class. A **union** cannot have virtual member functions. (Virtual functions are discussed in Chapter 17.) No **static** variables can be members of a **union**. A reference member cannot be used. A **union** cannot have as a member any object that overloads the **=** operator. Finally, no object can be a member of a **union** if the object has an explicit constructor or destructor function.

Friend Functions

It is possible to grant a nonmember function access to the private members of a class by using a **friend**. A **friend** function has access to all **private** and **protected** members of the class for which it is a **friend**. To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend**. Consider this program:

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

class myclass {
  int a, b;
public:
  friend int sum(myclass x);
  void set_ab(int i, int j);
};

void myclass::set_ab(int i, int j)
{
  a = i;
  b = j;
}

// Note: sum() is not a member function of any class.
int sum(myclass x)
{
  /* Because sum() is a friend of myclass, it can
     directly access a and b. */

  return x.a + x.b;
}

int main()
{
  myclass n;

  n.set_ab(3, 4);

  cout << sum(n);

  return 0;
}
```

precede its definition with the **inline** keyword. For example, in this program, the function **max( )** is expanded in line instead of called:

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

inline int max(int a, int b)
{
  return a>b ? a : b;
}

int main()
{
  cout << max(10, 20);
  cout << " " << max(99, 88);

  return 0;
}
```

As far as the compiler is concerned, the preceding program is equivalent to this one:

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

int main()
{

  cout << (10>20 ? 10 : 20);
  cout << " " << (99>88 ? 99 : 88);

  return 0;
}
```


The reason that **inline** functions are an important addition to C++ is that they allow you to create very efficient code. Since classes typically require several frequently executed interface functions (which provide access to private data), the efficiency of these functions is of critical concern. As you probably know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time. However, when a function is expanded in line, none of those operations occur. Although expanding function calls in line can

Notice that in the definition of **myclass( ),** the parameters **i** and **j** are used to give initial values to **a** and **b**.

The program illustrates the most common way to specify arguments when you declare an object that uses a parameterized constructor function. Specifically, this statement

```
myclass ob(3, 4);
```

causes an object called **ob** to be created and passes the arguments **3** and **4** to the **i** and **j** parameters of **myclass( ).** You may also pass arguments using this type of declaration statement:

```
myclass ob = myclass(3, 4);
```

However, the first method is the one generally used, and this is the approach taken by most of the examples in this book. Actually, there is a small technical difference between the two types of declarations that relates to copy constructors. (Copy constructors are discussed in Chapter 14.)

Here is another example that uses a parameterized constructor function. It creates a class that stores information about library books.


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

const int IN = 1;
const int CHECKED_OUT = 0;

class book {
  char author[40];
  char title[40];
  int status;
public:
  book(char *n, char *t, int s);
  int get_status() {return status;}
  void set_status(int s) {status = s;}
  void show();
};

book::book(char *n, char *t, int s)
{
```

```
  p = ob; // get start of array
  for(i=0; i<3; i++) {
    cout << p->get_i() << "\n";
    p++; // point to next object
  }

  return 0;
}
```

You can assign the address of a public member of an object to a pointer and then access that member by using the pointer. For example, this is a valid C++ program that displays the number **1** on the screen:

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

class cl {
public:
  int i;
  cl(int j) { i=j; }
};

int main()
{
  cl ob(1);
  int *p;

  p = &ob.i; // get address of ob.i

  cout << *p; // access ob.i via p

  return 0;
}
```


Because **p** is pointing to an integer, it is declared as an integer pointer. It is irrelevant that **i** is a member of **ob** in this situation.

Type Checking C++ Pointers

There is one important thing to understand about pointers in C++: You may assign one pointer to another only if the two pointer types are compatible. For example, given:

Of course, the type of the initializer must be compatible with the type of data for which memory is being allocated.

This program gives the allocated integer an initial value of 87:

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

int main()
{
  int *p;

  try {
    p = new int (87); // initialize to 87
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  cout << "At " << p << " ";
  cout << "is the value " << *p << "\n";

  delete p;

  return 0;
}
```


Allocating Arrays

You can allocate arrays using **new** by using this general form:

p_var = new *array_type [size];*

Here, *size* specifies the number of elements in the array.

To free an array, use this form of **delete:**

delete [ ] *p_var;*

Here, the **[ ]** informs **delete** that an array is being released.

For example, the next program allocates a 10-element integer array.

```
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  p->get_bal(n, s);

  cout << s << "'s balance is: " << n;
  cout << "\n";

  delete p;

  return 0;
}
```

The parameters to the object's constructor function are specified after the type name, just as in other sorts of initializations.

You can allocate arrays of objects, but there is one catch. Since no array allocated by **new** can have an initializer, you must make sure that if the class contains constructor functions, one will be parameterless. If you don't, the C++ compiler will not find a matching constructor when you attempt to allocate the array and will not compile your program.

In this version of the preceding program, an array of **balance** objects is allocated, and the parameterless constructor is called.


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

class balance {
  double cur_bal;
  char name[80];
public:
  balance(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  balance() {} // parameterless constructor
  ~balance() {
    cout << "Destructing ";
```

The output from this program is shown here.

```
Ralph Wilson's balance is: 12387.9
A. C. Conners's balance is: 144
I. M. Overdrawn's balance is: -11.23
Destructing I. M. Overdrawn
Destructing A. C. Conners
Destructing Ralph Wilson
```

One reason that you need to use the **delete [ ]** form when deleting an array of dynamically allocated objects is so that the destructor function can be called for each object in the array.

The nothrow Alternative

In Standard C++ it is possible to have **new** return **null** instead of throwing an exception when an allocation failure occurs. This form of **new** is most useful when you are compiling older code with a modern C++ compiler. It is also valuable when you are replacing calls to **malloc( )** with **new**. (This is common when updating C code to C++.) This form of **new** is shown here:

p_var = new(nothrow) *type*;

Here, *p_var* is a pointer variable of *type*. The **nothrow** form of **new** works like the original version of **new** from years ago. Since it returns null on failure, it can be "dropped into" older code without having to add exception handling. However, for new code, exceptions provide a better alternative. To use the **nothrow** option, you must include the header **<new>**.

The following program shows how to use the **new(nothrow)** alternative.

```
// Demonstrate nothrow version of new.
#include <iostream>
#include <new>
using namespace std;

int main()
{
  int *p, i;

  p = new(nothrow) int[32]; // use nothrow option
  if(!p) {
    cout << "Allocation failure.\n";
```

The next program overloads **myfunc( )** using a different number of parameters:

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

int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);

int main()
{
  cout << myfunc(10) << " "; // calls myfunc(int i)
  cout << myfunc(4, 5); // calls myfunc(int i, int j)

  return 0;
}

int myfunc(int i)
{
  return i;
}

int myfunc(int i, int j)
{
  return i*j;
}
```


As mentioned, the key point about function overloading is that the functions must differ in regard to the types and/or number of parameters. Two functions differing only in their return types cannot be overloaded. For example, this is an invalid attempt to overload **myfunc( ):**

```
int myfunc(int i);   // Error: differing return types are
float myfunc(int i); // insufficient when overloading.
```

Sometimes, two function declarations will appear to differ, when in fact they do not. For example, consider the following declarations.

```
void f(int *p);
void f(int p[]); // error, *p is same as p[]
```

Remember, to the compiler ***p** is the same as **p[ ].** Therefore, although the two prototypes appear to differ in the types of their parameter, in actuality they do not.

Default Function Arguments

C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function. The default value is specified in a manner syntactically similar to a variable initialization. For example, this declares **myfunc( )** as taking one **double** argument with a default value of 0.0:

```
void myfunc(double d = 0.0)
{
  // ...
}
```

Now, **myfunc( )** can be called one of two ways, as the following examples show:

```
myfunc(198.234); // pass an explicit value
myfunc();        // let function use default
```

The first call passes the value 198.234 to **d**. The second call automatically gives **d** the default value zero.

One reason that default arguments are included in C++ is because they provide another method for the programmer to manage greater complexity. To handle the widest variety of situations, quite frequently a function contains more parameters than are required for its most common usage. Thus, when the default arguments apply, you need specify only the arguments that are meaningful to the exact situation, not all those needed by the most general case. For example, many of the C++ I/O functions make use of default arguments for just this reason.

A simple illustration of how useful a default function argument can be is shown by the **clrscr( )** function in the following program. The **clrscr( )** function clears the screen by outputting a series of linefeeds (not the most efficient way, but sufficient for this example). Because a very common video mode displays 25 lines of text, the default argument of 25 is provided. However, because some terminals can display more or less than 25 lines (often depending upon what type of video mode is used), you can override the default argument by specifying one explicitly.

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

void clrscr(int size=25);

int main()
{
  register int i;
```

```
  for(i=0; i<30; i++ ) cout << i << endl;
  cin.get();
  clrscr(); // clears 25 lines

  for(i=0; i<30; i++ ) cout << i << endl;
  cin.get();
  clrscr(10); // clears 10 lines

  return 0;
}

void clrscr(int size)
{
   for(; size; size--) cout << endl;
}
```

As this program illustrates, when the default value is appropriate to the situation, no argument need be specified when **clrscr( )** is called. However, it is still possible to override the default and give **size** a different value when needed.

A default argument can also be used as a flag telling the function to reuse a previous argument. To illustrate this usage, a function called **iputs( )** is developed here that automatically indents a string by a specified amount. To begin, here is a version of this function that does not use a default argument:

```
void iputs(char *str, int indent)
{
  if(indent < 0) indent = 0;

  for( ; indent; indent--) cout << " ";

  cout << str << "\n";
}
```

This version of **iputs( )** is called with the string to output as the first argument and the amount to indent as the second. Although there is nothing wrong with writing **iputs( )** this way, you can improve its usability by providing a default argument for the **indent** parameter that tells **iputs( )** to indent to the previously specified level. It is quite common to display a block of text with each line indented the same amount. In this situation, instead of having to supply the same **indent** argument over and over, you can give

Closely related to function overloading is operator overloading. In C++, you can overload most operators so that they perform special operations relative to classes that you create. For example, a class that maintains a stack might overload **+** to perform a push operation and **– –** to perform a pop. When an operator is overloaded, none of its original meanings are lost. Instead, the type of objects it can be applied to is expanded.

The ability to overload operators is one of C++'s most powerful features. It allows the full integration of new class types into the programming environment. After overloading the appropriate operators, you can use objects in expressions in just the same way that you use C++'s built-in data types. Operator overloading also forms the basis of C++'s approach to I/O.

You overload operators by creating operator functions. An *operator function* defines the operations that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword **operator**. Operator functions can be either members or nonmembers of a class. Nonmember operator functions are almost always friend functions of the class, however. The way operator functions are written differs between member and nonmember functions. Therefore, each will be examined separately, beginning with member operator functions.

Creating a Member Operator Function

A member operator function takes this general form:

```
ret-type class-name::operator#(arg-list)
{
  // operations
}
```

Often, operator functions return an object of the class they operate on, but *ret-type* can be any valid type. The **#** is a placeholder. When you create an operator function, substitute the operator for the **#**. For example, if you are overloading the **/** operator, use **operator/.** When you are overloading a unary operator, *arg-list* will be empty. When you are overloading binary operators, *arg-list* will contain one parameter. (The reasons for this seemingly unusual situation will be made clear in a moment.)

Here is a simple first example of operator overloading. This program creates a class called **loc**, which stores longitude and latitude values. It overloads the **+** operator relative to this class. Examine this program carefully, paying special attention to the definition of **operator+( )**:

```
// Postfix increment
type operator++(int x) {
 // body of postfix operator
}

// Prefix decrement
type operator– –( ) {
 // body of prefix operator
}

// Postfix decrement
type operator– –(int x) {
 // body of postfix operator
}
```

Note *You should be careful when working with older C++ programs where the increment and decrement operators are concerned. In older versions of C++ it was not possible to specify separate prefix and postfix versions of an overloaded* ++ *or* – –. *The prefix form was used for both.*

Overloading the Shorthand Operators

You can overload any of C++'s "shorthand" operators, such as **+=, –=,** and the like. For example, this function overloads **+=** relative to **loc**:

```
loc loc::operator+=(loc op2)
{
  longitude = op2.longitude + longitude;
  latitude = op2.latitude + latitude;

  return *this;
}
```

When overloading one of these operators, keep in mind that you are simply combining an assignment with another type of operation.

Operator Overloading Restrictions

There are some restrictions that apply to operator overloading. You cannot alter the precedence of an operator. You cannot change the number of operands that an operator takes. (You can choose to ignore an operand, however.) Except for the function call

```
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }

  void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
  }

  friend loc operator+(loc op1, int op2);
  friend loc operator+(int op1, loc op2);
};

// + is overloaded for loc + int.
loc operator+(loc op1, int op2)
{
  loc temp;

  temp.longitude = op1.longitude + op2;
  temp.latitude = op1.latitude + op2;

  return temp;
}
// + is overloaded for int + loc.
loc operator+(int op1, loc op2)
{
  loc temp;

  temp.longitude = op1 + op2.longitude;
  temp.latitude = op1 + op2.latitude;

  return temp;
}

int main()
{
  loc ob1(10, 20), ob2( 5, 30), ob3(7, 14);

  ob1.show();
```

```
    cout << "Allocation error for p1.\n";
    return 1;;
  }

  try {
    p2 = new loc (-10, -20);
  } catch (bad_alloc xa) {
    cout << "Allocation error for p2.\n";
    return 1;;
  }

  try {
    f = new float; // uses overloaded new, too
  } catch (bad_alloc xa) {
    cout << "Allocation error for f.\n";
    return 1;;
  }

  *f = 10.10F;
  cout << *f << "\n";

  p1->show();
  p2->show();

  delete p1;
  delete p2;
  delete f;

  return 0;
}
```


Run this program to prove to yourself that the built-in **new** and **delete** operators have indeed been overloaded.

Overloading new and delete for Arrays

If you want to be able to allocate arrays of objects using your own allocation system, you will need to overload **new** and **delete** a second time. To allocate and free arrays, you must use these forms of **new** and **delete**.

```
  cout << ob[1]; // displays 2
  cout << " ";

  ob[1] = 25; // [] appears on left
  cout << ob[1]; // displays 25

  ob[3] = 44; // generates runtime error, 3 out-of-range

  return 0;
}
```

In this program, when the statement

```
ob[3] = 44;
```

executes, the boundary error is intercepted by **operator[]( )**, and the program is terminated before any damage can be done. (In actual practice, some sort of error-handling function would be called to deal with the out-of-range condition; the program would not have to terminate.)

Overloading ( )

When you overload the ( ) function call operator, you are not, per se, creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters. Let's begin with an example. Given the overloaded operator function declaration

```
double operator()(int a, float f, char *s);
```

and an object **O** of its class, then the statement

```
O(10, 23.34, "hi");
```

translates into this call to the **operator( )** function.

```
O.operator()(10, 23.34, "hi");
```

In general, when you overload the **( )** operator, you define the parameters that you want to pass to that function. When you use the **( )** operator in your program, the

```
public:
  derived(int x) { k=x; }
  void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob(3);

  ob.set(1, 2); // error, can't access set()
  ob.show(); // error, can't access show()

  return 0;
}
```

Remember *When a base class' access specifier is **private**, public and protected members of the base become private members of the derived class. This means that they are still accessible by members of the derived class but cannot be accessed by parts of your program that are not members of either the base or derived class.*

Inheritance and protected Members

The **protected** keyword is included in C++ to provide greater flexibility in the inheritance mechanism. When a member of a class is declared as **protected**, that member is not accessible by other, nonmember elements of the program. With one important exception, access to a protected member is the same as access to a private member—it can be accessed only by other members of its class. The sole exception to this is when a protected member is inherited. In this case, a protected member differs substantially from a private one.

As explained in the preceding section, a private member of a base class is not accessible by other parts of your program, including any derived class. However, protected members behave differently. If the base class is inherited as **public**, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class. By using **protected**, you can create class members that are private to their class but that can still be inherited and accessed by a derived class. Here is an example:

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

class base {
```

```
protected:
  int i, j; // private to base, but accessible by derived
public:
  void set(int a, int b) { i=a; j=b; }
  void show() { cout << i << " " << j << "\n"; }
};

class derived : public base {
  int k;
public:
  // derived may access base's i and j
  void setk() { k=i*j; }

  void showk() { cout << k << "\n"; }
};

int main()
{
  derived ob;

  ob.set(2, 3); // OK, known to derived
  ob.show(); // OK, known to derived

  ob.setk();
  ob.showk();

  return 0;
}
```


In this example, because **base** is inherited by **derived** as **public** and because **i** and **j** are declared as **protected, derived**'s function **setk( )** may access them. If **i** and **j** had been declared as **private** by **base**, then **derived** would not have access to them, and the program would not compile.

When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited (as public) by the first derived class may also be inherited as protected again by a second derived class. For example, this program is correct, and **derived2** does indeed have access to **i** and **j**.

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

class base {
```

```
public:
  int j, k;
  void seti(int x) { i = x; }
  int geti() { return i; }
};

// Inherit base as private.
class derived: private base {
public:
  /* The next three statements override
     base's inheritance as private and restore j,
     seti(), and geti() to public access. */
  base::j; // make j public again - but not k
  base::seti; // make seti() public
  base::geti; // make geti() public

// base::i; // illegal, you cannot elevate access

  int a; // public
};

int main()
{
  derived ob;

//ob.i = 10; // illegal because i is private in derived

  ob.j = 20; // legal because j is made public in derived
//ob.k = 30; // illegal because k is private in derived

  ob.a = 40; // legal because a is public in derived
  ob.seti(10);

  cout << ob.geti() << " " << ob.j << " " << ob.a;

  return 0;
}
```


Access declarations are supported in C++ to accommodate those situations in which most of an inherited class is intended to be made private, but a few members are to retain their public or protected status.

```
  derived3 ob;

  ob.i = 10; // this is ambiguous, which i???
  ob.j = 20;
  ob.k = 30;

  // i ambiguous here, too
  ob.sum = ob.i + ob.j + ob.k;

  // also ambiguous, which i?
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

As the comments in the program indicate, both **derived1** and **derived2** inherit **base**. However, **derived3** inherits both **derived1** and **derived2**. This means that there are two copies of **base** present in an object of type **derived3**. Therefore, in an expression like

```
ob.i = 10;
```

which **i** is being referred to, the one in **derived1** or the one in **derived2?** Because there are two copies of **base** present in object **ob**, there are two **ob.i**s! As you can see, the statement is inherently ambiguous.

There are two ways to remedy the preceding program. The first is to apply the scope resolution operator to **i** and manually select one **i.** For example, this version of the program does compile and run as expected:

```
// This program uses explicit scope resolution to select i.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derived1 inherits base.
```

being included in **derived3**? The answer, as you probably have guessed, is yes. This solution is achieved using **virtual** base classes.

When two or more objects are derived from a common base class, you can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as **virtual** when it is inherited. You accomplish this by preceding the base class' name with the keyword **virtual** when it is inherited. For example, here is another version of the example program in which **derived3** contains only one copy of **base:**

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;

class base {
public:
  int i;
};

// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
  int j;
};

// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
  int k;
};

/* derived3 inherits both derived1 and derived2.
   This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
  int sum;
};

int main()
{
  derived3 ob;

  ob.i = 10; // now unambiguous
```

```
  ob.j = 20;
  ob.k = 30;

  // unambiguous
  ob.sum = ob.i + ob.j + ob.k;

  // unambiguous
  cout << ob.i << " ";

  cout << ob.j << " " << ob.k << " ";
  cout << ob.sum;

  return 0;
}
```

As you can see, the keyword **virtual** precedes the rest of the inherited class' specification. Now that both **derived1** and **derived2** have inherited **base** as **virtual**, any multiple inheritance involving them will cause only one copy of **base** to be present. Therefore, in **derived3**, there is only one copy of **base** and **ob.i = 10** is perfectly valid and unambiguous.

One further point to keep in mind: Even though both **derived1** and **derived2** specify **base** as **virtual**, **base** is still present in objects of either type. For example, the following sequence is perfectly valid:

```
// define a class of type derived1
derived1 myclass;

myclass.i = 88;
```

The only difference between a normal base class and a **virtual** one is what occurs when an object inherits the base more than once. If **virtual** base classes are used, then only one base class is present in the object. Otherwise, multiple copies will be found.

This page intentionally left blank.

Polymorphism is supported by C++ both at compile time and at run time. As discussed in earlier chapters, compile-time polymorphism is achieved by overloading functions and operators. Run-time polymorphism is accomplished by using inheritance and virtual functions, and these are the topics of this chapter.

Virtual Functions

A *virtual function* is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the *form* of the *interface* to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a *specific method*.

When accessed "normally," virtual functions behave just like any other type of class member function. However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when accessed via a pointer. As discussed in Chapter 13, a base-class pointer can be used to point to an object of any class derived from that base. When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon *the type of object pointed to* by the pointer. And this determination is made *at run time*. Thus, when different objects are pointed to, different versions of the virtual function are executed. The same effect applies to base-class references.

To begin, examine this short example:


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

class base {
public:
  virtual void vfunc() {
    cout << "This is base's vfunc().\n";
  }
};

class derived1 : public base {
public:
  void vfunc() {
    cout << "This is derived1's vfunc().\n";
  }
```

```
public:
// vfunc() not overridden by derived2, base's is used
};

int main()
{
  base *p, b;
  derived1 d1;
  derived2 d2;

  // point to base
  p = &b;
  p->vfunc(); // access base's vfunc()

  // point to derived1
  p = &d1;
  p->vfunc(); // access derived1's vfunc()

  // point to derived2
  p = &d2;
  p->vfunc(); // use base's vfunc()

  return 0;
}
```


The program produces this output:

```
This is base's vfunc().
This is derived1's vfunc().
This is base's vfunc().
```

Because **derived2** does not override **vfunc( )**, the function defined by **base** is used when **vfunc( )** is referenced relative to objects of type **derived2**.

The preceding program illustrates a special case of a more general rule. Because inheritance is hierarchical in C++, it makes sense that virtual functions are also hierarchical. This means that when a derived class fails to override a virtual function, the first redefinition found in reverse order of derivation is used. For example, in the following program, **derived2** is derived from **derived1**, which is derived from **base**. However, **derived2** does not override **vfunc( )**. This means that, relative to **derived2**,

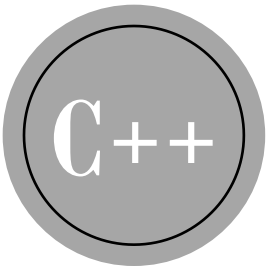
```
protected:
  int val;
public:
  void setval(int i) { val = i; }

  // show() is a pure virtual function
  virtual void show() = 0;
};

class hextype : public number {
public:
  void show() {
    cout << hex << val << "\n";
  }
};

class dectype : public number {
public:
  void show() {
    cout << val << "\n";
  }
};

class octtype : public number {
public:
  void show() {
    cout << oct << val << "\n";
  }
};

int main()
{
  dectype d;
  hextype h;
  octtype o;

  d.setval(20);
  d.show(); // displays 20 - decimal

  h.setval(20);
  h.show(); // displays 14 - hexadecimal
```

```
public:
  f_to_c(double i) : convert(i) { }
  void compute() {
    val2 = (val1-32) / 1.8;
  }
};

int main()
{
  convert *p;  // pointer to base class

  l_to_g lgob(4);
  f_to_c fcob(70);

  // use virtual function mechanism to convert
  p = &lgob;
  cout << p->getinit() << " liters is ";
  p->compute();
  cout << p->getconv() << " gallons\n"; // l_to_g

  p = &fcob;
  cout << p->getinit() << " in Fahrenheit is ";
  p->compute();
  cout << p->getconv() << " Celsius\n";  // f_to_c

  return 0;
}
```


The preceding program creates two derived classes from **convert**, called **l_to_g** and **f_to_c**. These classes perform the conversions of liters to gallons and Fahrenheit to Celsius, respectively. Each derived class overrides **compute( )** in its own way to perform the desired conversion. However, even though the actual conversion (that is, method) differs between **l_to_g** and **f_to_c**, the interface remains constant.

One of the benefits of derived classes and virtual functions is that handling a new case is a very easy matter. For example, assuming the preceding program, you can add a conversion from feet to meters by including this class:

```
// Feet to meters
class f_to_m : public convert {
public:
  f_to_m(double i) : convert(i) { }
```



Chapter 18

Templates

A Generic Sort

Sorting is exactly the type of operation for which generic functions were designed. Within wide latitude, a sorting algorithm is the same no matter what type of data is being sorted. The following program illustrates this by creating a generic bubble sort. While the bubble sort is a rather poor sorting algorithm, its operation is clear and uncluttered and it makes an easy-to-understand example. The **bubble( )** function will sort any type of array. It is called with a pointer to the first element in the array and the number of elements in the array.

```
// A Generic bubble sort.
#include <iostream>
using namespace std;

template <class X> void bubble(
  X *items,  // pointer to array to be sorted
  int count) // number of items in array
{
  register int a, b;
  X t;

  for(a=1; a<count; a++)
    for(b=count-1; b>=a; b--)
      if(items[b-1] > items[b]) {
        // exchange elements
        t = items[b-1];
        items[b-1] = items[b];
        items[b] = t;
      }
}

int main()
{
  int iarray[7] = {7, 5, 4, 3, 9, 8, 6};
  double darray[5] = {4.3, 2.5, -0.9, 100.2, 3.0};

  int i;

  cout << "Here is unsorted integer array: ";
  for(i=0;  i<7; i++)
```

```
    register int i;
    for(i=0; i<SIZE; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType> AType &atype<AType>::operator[](int i)
{
  if(i<0 || i> SIZE-1) {
    cout << "\nIndex value of ";
    cout << i << " is out-of-bounds.\n";
    exit(1);
  }
  return a[i];
}

int main()
{
  atype<int> intob; // integer array
  atype<double> doubleob; // double array

  int i;

  cout << "Integer array: ";
  for(i=0; i<SIZE; i++) intob[i] = i;
  for(i=0; i<SIZE; i++) cout << intob[i] << "  ";
  cout << '\n';

  cout << "Double array: ";
  for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;
  for(i=0; i<SIZE; i++) cout << doubleob[i] << "  ";
  cout << '\n';

  intob[12] = 100; // generates runtime error

  return 0;
}
```


This program implements a generic safe-array type and then demonstrates its use by creating an array of **int**s and an array of **double**s. You should try creating other types of arrays. As this example shows, part of the power of generic classes is that they

```
  atype<int, 10> intob;       // integer array of size 10
  atype<double, 15> doubleob; // double array of size 15

  int i;

  cout << "Integer array: ";
  for(i=0; i<10; i++) intob[i] = i;
  for(i=0; i<10; i++) cout << intob[i] << "  ";
  cout << '\n';

  cout << "Double array: ";
  for(i=0; i<15; i++) doubleob[i] = (double) i/3;
  for(i=0; i<15; i++) cout << doubleob[i] << "  ";
  cout << '\n';

  intob[12] = 100; // generates runtime error

  return 0;
}
```


Look carefully at the template specification for **atype**. Note that **size** is declared as an **int**. This parameter is then used within **atype** to declare the size of the array **a**. Even though **size** is depicted as a "variable" in the source code, its value is known at compile time. This allows it to be used to set the size of the array. **size** is also used in the bounds checking within the **operator][( )** function. Within **main( )**, notice how the integer and floating-point arrays are created. The second parameter specifies the size of each array.

Non-type parameters are restricted to integers, pointers, or references. Other types, such as **float**, are not allowed. The arguments that you pass to a non-type parameter must consist of either an integer constant, or a pointer or reference to a global function or object. Thus, non-type parameters should themselves be thought of as constants, since their values cannot be changed. For example, inside **operator[ ]( )**, the following statement is not allowed.

```
size = 10; // Error
```

Since non-type parameters are treated as constants, they can be used to set the dimension of an array, which is a significant, practical benefit.

As the safe-array example illustrates, the use of non-type parameters greatly expands the utility of template classes. Although the information contained in the non-type argument must be known at compile-time, this restriction is mild compared with the power offered by non-type parameters.

The following program shows how to restrict the types of exceptions that can be thrown from a function.

```
// Restricting function throw types.
#include <iostream>
using namespace std;

// This function can only throw ints, chars, and doubles.
void Xhandler(int test) throw(int, char, double)
{
  if(test==0) throw test; // throw int
  if(test==1) throw 'a'; // throw char
  if(test==2) throw 123.23; // throw double
}

int main()
{
  cout << "start\n";

  try{
    Xhandler(0); // also try passing 1 and 2 to Xhandler()
  }
  catch(int i) {
    cout << "Caught an integer\n";
  }
  catch(char c) {
    cout << "Caught char\n";
  }
  catch(double d) {
    cout << "Caught double\n";
  }

  cout << "end";

  return 0;
}
```


In this program, the function **Xhandler( )** may only throw integer, character, and **double** exceptions. If it attempts to throw any other type of exception, an abnormal program termination will occur. (That is, **unexpected( )** will be called.) To see an example of this, remove **int** from the list and retry the program.

It is important to understand that a function can be restricted only in what types of exceptions it throws back to the **try** block that called it. That is, a **try** block *within* a

function may throw any type of exception so long as it is caught *within* that function. The restriction applies only when throwing an exception outside of the function.

The following change to **Xhandler( )** prevents it from throwing any exceptions.

```
// This function can throw NO exceptions!
void Xhandler(int test) throw()
{
  /* The following statements no longer work.  Instead,
     they will cause an abnormal program termination. */
  if(test==0) throw test;
  if(test==1) throw 'a';
  if(test==2) throw 123.23;
}
```

Note *At the time of this writing, Microsoft's Visual C++ does not support the* ***throw( )*** *clause for functions.*

Rethrowing an Exception

If you wish to rethrow an expression from within an exception handler, you may do so by calling **throw**, by itself, with no exception. This causes the current exception to be passed on to an outer **try/catch** sequence. The most likely reason for doing so is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler copes with another. An exception can only be rethrown from within a **catch** block (or from any function called from within that block). When you rethrow an exception, it will not be recaught by the same **catch** statement. It will propagate outward to the next **catch** statement. The following program illustrates rethrowing an exception, in this case a **char *** exception.


```
// Example of "rethrowing" an exception.
#include <iostream>
using namespace std;

void Xhandler()
{
  try {
    throw "hello"; // throw a char *
  }
  catch(const char *) { // catch a char *
    cout << "Caught char * inside Xhandler\n";
    throw ; // rethrow char * out of function
  }
}
```

C++ Streams

Like the C-based I/O system, the C++ I/O system operates through streams. Streams were discussed in detail in Chapter 9; that discussion will not be repeated here. However, to summarize: A *stream* is a logical device that either produces or consumes information. A stream is linked to a physical device by the I/O system. All streams behave in the same way even though the actual physical devices they are connected to may differ substantially. Because all streams behave the same, the same I/O functions can operate on virtually any type of physical device. For example, you can use the same function that writes to a file to write to the printer or to the screen. The advantage to this approach is that you need learn only one I/O system.

The C++ Stream Classes

As mentioned, Standard C++ provides support for its I/O system in **<iostream>**. In this header, a rather complicated set of class hierarchies is defined that support I/O operations. The I/O classes begin with a system of template classes. As explained in Chapter 18, a template class defines the form of a class without fully specifying the data upon which it will operate. Once a template class has been defined, specific instances of it can be created. As it relates to the I/O library, Standard C++ creates two specializations of the I/O template classes: one for 8-bit characters and another for wide characters. This book will use only the 8-bit character classes since they are by far the most common, but the same techniques apply to both.

The C++ I/O system is built upon two related but different template class hierarchies. The first is derived from the low-level I/O class called **basic_streambuf**. This class supplies the basic, low-level input and output operations, and provides the underlying support for the entire C++ I/O system. Unless you are doing advanced I/O programming, you will not need to use **basic_streambuf** directly. The class hierarchy that you will most commonly be working with is derived from **basic_ios**. This is a high-level I/O class that provides formatting, error checking, and status information related to stream I/O. (A base class for **basic_ios** is called **ios_base**, which defines several nontemplate traits used by **basic_ios**.) **basic_ios** is used as a base for several derived classes, including **basic_istream**, **basic_ostream**, and **basic_iostream**. These classes are used to create streams capable of input, output, and input/output, respectively.

As explained, the I/O library creates two specializations of the template class hierarchies just described: one for 8-bit characters and one for wide characters. Here is a list of the mapping of template class names to their character and wide-character versions.

Standard C++ also defines these four additional streams: **win**, **wout**, **werr**, and **wlog**. These are wide-character versions of the standard streams. Wide characters are of type **wchar_t** and are generally 16-bit quantities. Wide characters are used to hold the large character sets associated with some human languages.

Formatted I/O

The C++ I/O system allows you to format I/O operations. For example, you can set a field width, specify a number base, or determine how many digits after the decimal point will be displayed. There are two related but conceptually different ways that you can format data. First, you can directly access members of the **ios** class. Specifically, you can set various format status flags defined inside the **ios** class or call various **ios** member functions. Second, you can use special functions called *manipulators* that can be included as part of an I/O expression.

We will begin the discussion of formatted I/O by using the **ios** member functions and flags.

Formatting Using the ios Members

Each stream has associated with it a set of format flags that control the way information is formatted. The **ios** class declares a bitmask enumeration called **fmtflags** in which the following values are defined. (Technically, these values are defined within **ios_base**, which, as explained earlier, is a base class for **ios**.)

adjustfield	basefield	boolalpha	dec
fixed	floatfield	hex	internal
left	oct	right	scientific
showbase	showpoint	showpos	skipws
unitbuf	uppercase		

These values are used to set or clear the format flags. If you are using an older compiler, it may not define the **fmtflags** enumeration type. In this case, the format flags will be encoded into a long integer.

When the **skipws** flag is set, leading white-space characters (spaces, tabs, and newlines) are discarded when performing input on a stream. When **skipws** is cleared, white-space characters are not discarded.

When the **left** flag is set, output is left justified. When **right** is set, output is right justified. When the **internal** flag is set, a numeric value is padded to fill a field by inserting spaces between any sign or base character. If none of these flags are set, output is right justified by default.

By default, numeric values are output in decimal. However, it is possible to change the number base. Setting the **oct** flag causes output to be displayed in octal. Setting the **hex** flag causes output to be displayed in hexadecimal. To return output to decimal, set the **dec** flag.

Setting **showbase** causes the base of numeric values to be shown. For example, if the conversion base is hexadecimal, the value 1F will be displayed as 0x1F.

By default, when scientific notation is displayed, the **e** is in lowercase. Also, when a hexadecimal value is displayed, the **x** is in lowercase. When **uppercase** is set, these characters are displayed in uppercase.

Setting **showpos** causes a leading plus sign to be displayed before positive values.

Setting **showpoint** causes a decimal point and trailing zeros to be displayed for all floating-point output—whether needed or not.

By setting the **scientific** flag, floating-point numeric values are displayed using scientific notation. When **fixed** is set, floating-point values are displayed using normal notation. When neither flag is set, the compiler chooses an appropriate method.

When **unitbuf** is set, the buffer is flushed after each insertion operation.

When **boolalpha** is set, Booleans can be input or output using the keywords **true** and **false**.

Since it is common to refer to the **oct**, **dec**, and **hex** fields, they can be collectively referred to as **basefield**. Similarly, the **left**, **right**, and **internal** fields can be referred to as **adjustfield**. Finally, the **scientific** and **fixed** fields can be referenced as **floatfield**.

Setting the Format Flags

To set a flag, use the **setf( )** function. This function is a member of **ios**. Its most common form is shown here:

fmtflags setf(fmtflags *flags*);

This function returns the previous settings of the format flags and turns on those flags specified by *flags*. For example, to turn on the **showpos** flag, you can use this statement:

```
stream.setf(ios::showpos);
```

Here, *stream* is the stream you wish to affect. Notice the use of **ios::** to qualify **showpos**. Since **showpos** is an enumerated constant defined by the **ios** class, it must be qualified by **ios** when it is used.

The following program displays the value 100 with the **showpos** and **showpoint** flags turned on.

There are overloaded forms of **width( )**, **precision( )**, and **fill( )** that obtain but do not change the current setting. These forms are shown here:

```
char fill( );
streamsize width( );
streamsize precision( );
```

Using Manipulators to Format I/O

The second way you can alter the format parameters of a stream is through the use of special functions called *manipulators* that can be included in an I/O expression. The standard manipulators are shown in Table 20-1. As you can see by examining the table, many of the I/O manipulators parallel member functions of the **ios** class. Many of the manipulators were added recently to C++ and will not be supported by older compilers.

Manipulator	Purpose	Input/Output
boolalpha	Turns on **boolalpha** flag.	Input/Output
dec	Turns on **dec** flag.	Input/Output
endl	Output a newline character and flush the stream.	Output
ends	Output a null.	Output
fixed	Turns on **fixed** flag.	Output
flush	Flush a stream.	Output
hex	Turns on **hex** flag.	Input/Output
internal	Turns on **internal** flag.	Output
left	Turns on **left** flag.	Output
nobooalpha	Turns off **boolalpha** flag.	Input/Output
noshowbase	Turns off **showbase** flag.	Output
noshowpoint	Turns off **showpoint** flag.	Output
noshowpos	Turns off **showpos** flag.	Output

Table 20-1. *The C++ Manipulators*

```
int main()
{
  phonebook a("Ted", 111, 555, 1234);
  phonebook b("Alice", 312, 555, 5768);
  phonebook c("Tom", 212, 555, 9991);

  cout << a << b << c;

  return 0;
}
```

When you define the body of an inserter function, remember to keep it as general as possible. For example, the inserter shown in the preceding example can be used with any stream because the body of the function directs its output to **stream**, which is the stream that invoked the inserter. While it would not be wrong to have written

```
stream << o.name << " ";
```

as

```
cout << o.name << " ";
```

this would have the effect of hard-coding **cout** as the output stream. The original version will work with any stream, including those linked to disk files. Although in some situations, especially where special output devices are involved, you will want to hard-code the output stream, in most cases you will not. In general, the more flexible your inserters are, the more valuable they are.

Note *The inserter for the **phonebook** class works fine unless the value of **num** is something like 0034, in which case the preceding zeroes will not be displayed. To fix this, you can either make **num** into a string or you can set the fill character to zero and use the **width( )** format function to generate the leading zeroes. The solution is left to the reader as an exercise.*

Before moving on to extractors, let's look at one more example of an inserter function. An inserter need not be limited to handling only text. An inserter can be used to output data in any form that makes sense. For example, an inserter for some class that is part of a CAD system may output plotter instructions. Another inserter might generate graphics images. An inserter for a Windows-based program could display a dialog box. To sample the flavor of outputting things other than text, examine the following program, which draws boxes on the screen. (Because C++ does not define a

graphics library, the program uses characters to draw a box, but feel free to substitute graphics if your system supports them.)

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

class box {
  int x, y;
public:
  box(int i, int j) { x=i; y=j; }
  friend ostream &operator<<(ostream &stream, box o);
};

// Output a box.
ostream &operator<<(ostream &stream, box o)
{
  register int i, j;

  for(i=0; i<o.x; i++)
    stream << "*";

  stream << "\n";

  for(j=1; j<o.y-1; j++) {
    for(i=0; i<o.x; i++)
      if(i==0 || i==o.x-1) stream << "*";
      else stream << " ";
    stream << "\n";
  }

  for(i=0; i<o.x; i++)
     stream << "*";
  stream << "\n";

  return stream;
}

int main()
{
  box a(14, 6), b(30, 7), c(40, 5);
```

```
using namespace std;

// A simple input manipulator.
istream &getpass(istream &stream)
{
  cout << '\a';  // sound bell
  cout << "Enter password: ";

  return stream;
}

int main()
{
  char pw[80];

  do {
    cin >> getpass >> pw;
  } while (strcmp(pw, "password"));

  cout << "Logon complete\n";

  return 0;
}
```


Remember that it is required that your manipulator return **stream**. If it does not, your manipulator cannot be used in a series of input or output operations.

If **open( )** fails, the stream will evaluate to false when used in a Boolean expression. Therefore, before using a file, you should test to make sure that the open operation succeeded. You can do so by using a statement like this:

```
if(!mystream) {
  cout << "Cannot open file.\n";
  // handle error
}
```

Although it is entirely proper to open a file by using the **open( )** function, most of the time you will not do so because the **ifstream**, **ofstream**, and **fstream** classes have constructor functions that automatically open the file. The constructor functions have the same parameters and defaults as the **open( )** function. Therefore, you will most commonly see a file opened as shown here:

```
ifstream mystream("myfile"); // open file for input
```

As stated, if for some reason the file cannot be opened, the value of the associated stream variable will evaluate to false. Therefore, whether you use a constructor function to open the file or an explicit call to **open( )**, you will want to confirm that the file has actually been opened by testing the value of the stream.

You can also check to see if you have successfully opened a file by using the **is_open( )** function, which is a member of **fstream**, **ifstream**, and **ofstream**. It has this prototype:

```
bool is_open( );
```

It returns true if the stream is linked to an open file and false otherwise. For example, the following checks if **mystream** is currently open:

```
if(!mystream.is_open()) {
  cout << "File is not open.\n";
  // ...
```

To close a file, use the member function **close( )**. For example, to close the file linked to a stream called **mystream**, use this statement:

```
mystream.close();
```

The **close( )** function takes no parameters and returns no value.

```
  }

  char item[20];
  float cost;

  in >> item >>  cost;
  cout << item << " " << cost << "\n";
  in >> item >> cost;
  cout << item << " " << cost << "\n";
  in >> item >> cost;
  cout << item << " " << cost << "\n";

  in.close();
  return 0;
}
```

In a way, reading and writing files by using >> and << is like using the C-based functions **fprintf( )** and **fscanf( )** functions. All information is stored in the file in the same format as it would be displayed on the screen.

Following is another example of disk I/O. This program reads strings entered at the keyboard and writes them to disk. The program stops when the user enters an exclamation point. To use the program, specify the name of the output file on the command line.

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

int main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: output <filename>\n";
    return 1;
  }

  ofstream out(argv[1]); // output, normal file

  if(!out) {
    cout << "Cannot open output file.\n";
    return 1;
  }
```

```
  ifstream in(argv[1], ios::in | ios::binary);

  if(!in) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  register int i, j;
  int count = 0;
  char c[16];

  cout.setf(ios::uppercase);
  while(!in.eof()) {
    for(i=0; i<16 && !in.eof(); i++) {
      in.get(c[i]);
    }
    if(i<16) i--; // get rid of eof

    for(j=0; j<i; j++)
      cout << setw(3) << hex << (int) c[j];
    for(; j<16; j++) cout << "  ";

    cout << " ";
    for(j=0; j<i; j++)
      if(isprint(c[j])) cout << c[j];
      else cout << ".";

    cout << endl;

    count++;
    if(count==16) {
      count = 0;
      cout << "Press ENTER to continue: ";
      cin.get();
      cout << endl;
    }
  }

  in.close();

  return 0;
}
```

```
  fstream inout(argv[1], ios::in | ios::out | ios::binary);

  if(!inout) {
    cout << "Cannot open input file.\n";
    return 1;
  }

  long e, i, j;
  char c1, c2;
  e = atol(argv[2]);

  for(i=0, j=e; i<j; i++, j--) {
    inout.seekg(i, ios::beg);
    inout.get(c1);
    inout.seekg(j, ios::beg);
    inout.get(c2);

    inout.seekp(i, ios::beg);
    inout.put(c2);
    inout.seekp(j, ios::beg);
    inout.put(c1);
  }

  inout.close();
  return 0;
}
```


To use the program, specify the name of the file that you want to reverse, followed by the number of characters to reverse. For example, to reverse the first 10 characters of a file called TEST, use this command line:

```
reverse test 10
```

If the file had contained this:

```
This is a test.
```

it will contain the following after the program executes:

```
a si sihTtest.
```

```
  else if(i & ios::failbit)
    cout << "Non-Fatal I/O error\n";
  else if(i & ios::badbit)
    cout << "Fatal I/O error\n";
}
```

This program will always report one "error." After the **while** loop ends, the final call to **checkstatus( )** reports, as expected, that an **EOF** has been encountered. You might find the **checkstatus( )** function useful in programs that you write.

The other way that you can determine if an error has occurred is by using one or more of these functions:

```
bool bad( );
bool eof( );
bool fail( );
bool good( );
```

The **bad( )** function returns true if **badbit** is set. The **eof( )** function was discussed earlier. The **fail( )** returns true if **failbit** is set. The **good( )** function returns true if there are no errors. Otherwise, it returns false.

Once an error has occurred, it may need to be cleared before your program continues. To do this, use the **clear( )** function, which has this prototype:

```
void clear(iostate flags=ios::goodbit);
```

If *flags* is **goodbit** (as it is by default), all error flags are cleared. Otherwise, set *flags* as you desire.

Customized I/O and Files

In Chapter 20 you learned how to overload the insertion and extraction operators relative to your own classes. In that chapter, only console I/O was performed, but because all C++ streams are the same, you can use the same overloaded inserter or extractor function to perform I/O on the console or a file with no changes whatsoever. As an example, the following program reworks the phone book example in Chapter 20 so that it stores a list on disk. The program is very simple: It allows you to add names to the list or to display the list on the screen. It uses custom inserters and extractors to input and output the telephone numbers. You might find it interesting to enhance the program so that it will find a specific number or delete unwanted numbers.

```
        break;
      case '3':
        pb.close();
        return 0;
    }
  }
}
```

Notice that the overloaded << operator can be used to write to a disk file or to the screen without any changes. This is one of the most important and useful features of C++'s approach to I/O.

nothing to do with inheritance or class hierarchies.) The **name( )** function returns a pointer to the name of the type.

Here is a simple example that uses **typeid**.

```
// A simple example that uses typeid.
#include <iostream>
#include <typeinfo>
using namespace std;

class myclass1 {
  // ...
};

class myclass2 {
  // ...
};

int main()
{
  int i, j;
  float f;
  char *p;
  myclass1 ob1;
  myclass2 ob2;

  cout << "The type of i is: " << typeid(i).name();
  cout << endl;
  cout << "The type of f is: " << typeid(f).name();
  cout << endl;
  cout << "The type of p is: " << typeid(p).name();
  cout << endl;

  cout << "The type of ob1 is: " << typeid(ob1).name();
  cout << endl;
  cout << "The type of ob2 is: " << typeid(ob2).name();
  cout << "\n\n";

  if(typeid(i) == typeid(j))
    cout << "The types of i and j are the same\n";

  if(typeid(i) != typeid(f))
    cout << "The types of i and f are not the same\n";
```

```
dp = dynamic_cast<Derived *> (bp); // cast to derived pointer OK
if(dp) cout << "Cast OK";
```

Here, the cast from the base pointer **bp** to the derived pointer **dp** works because **bp** is actually pointing to a **Derived** object. Thus, this fragment displays **Cast OK**. But in the next fragment, the cast fails because **bp** is pointing to a **Base** object and it is illegal to cast a base object into a derived object.

```
bp = &b_ob; // base pointer points to Base object
dp = dynamic_cast<Derived *> (bp); // error
if(!dp) cout << "Cast Fails";
```

Because the cast fails, this fragment displays **Cast Fails**.

The following program demonstrates the various situations that **dynamic_cast** can handle.


```
// Demonstrate dynamic_cast.
#include <iostream>
using namespace std;

class Base {
public:
  virtual void f() { cout << "Inside Base\n"; }
  // ...
};

class Derived : public Base {
public:
  void f() { cout << "Inside Derived\n"; }
};

int main()
{
  Base *bp, b_ob;
  Derived *dp, d_ob;

  dp = dynamic_cast<Derived *> (&d_ob);
  if(dp) {
    cout << "Cast from Derived * to Derived * OK.\n";
    dp->f();
  } else
```

```
a pointer to a derived object.

Can't cast from Num<double>* to Num<int>*.
These are two different types.
```

A key point illustrated by this example is that it is not possible to use **dynamic_cast** to cast a pointer to one type of template instantiation into a pointer to another type of instance. Remember, the precise type of an object of a template class is determined by the type of data used to create an instance of the template. Thus, **Num<double>** and **Num<int>** are two different types.

const_cast

The **const_cast** operator is used to explicitly override **const** and/or **volatile** in a cast. The target type must be the same as the source type except for the alteration of its **const** or **volatile** attributes. The most common use of **const_cast** is to remove const-ness. The general form of **const_cast** is shown here.

const_cast<*type*> (*expr*)

Here, *type* specifies the target type of the cast, and *expr* is the expression being cast into the new type.

The following program demonstrates **const_cast**.


```
// Demonstrate const_cast.
#include <iostream>
using namespace std;

void sqrval(const int *val)
{
  int *p;

  // cast away const-ness.
  p = const_cast<int *> (val);

  *p = *val * *val; // now, modify object through v
}

int main()
{
```

Anything defined within a **namespace** statement is within the scope of that namespace.

Here is an example of a **namespace**. It localizes the names used to implement a simple countdown counter class. In the namespace are defined the **counter** class, which implements the counter, and the variables **upperbound** and **lowerbound**, which contain the upper and lower bounds that apply to all counters.

```
namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
     int count;
   public:
     counter(int n) {
       if(n <= upperbound) count = n;
       else count = upperbound;
     }

     void reset(int n) {
       if(n <= upperbound) count = n;
     }

     int run() {
       if(count > lowerbound) return count--;
       else return lowerbound;
     }
  };
}
```


Here, **upperbound**, **lowerbound**, and the class **counter** are part of the scope defined by the **CounterNameSpace** namespace.

Inside a namespace, identifiers declared within that namespace can be referred to directly, without any namespace qualification. For example, within **CounterNameSpace**, the **run( )** function can refer directly to **lowerbound** in the statement

```
if(count > lowerbound) return count--;
```

However, since **namespace** defines a scope, you need to use the scope resolution operator to refer to objects declared within a namespace from outside that namespace.

For example, to assign the value 10 to **upperbound** from code outside **CounterNameSpace**, you must use this statement:

```
CounterNameSpace::upperbound = 10;
```

Or to declare an object of type **counter** from outside **CounterNameSpace**, you will use a statement like this:

```
CounterNameSpace::counter ob;
```

In general, to access a member of a namespace from outside its namespace, precede the member's name with the name of the namespace followed by the scope resolution operator.

Here is a program that demonstrates the use of **CounterNameSpace**.

```
// Demonstrate a namespace.
#include <iostream>
using namespace std;

namespace CounterNameSpace {
  int upperbound;
  int lowerbound;

  class counter {
    int count;
  public:
    counter(int n) {
      if(n <= upperbound) count = n;
      else count = upperbound;
    }

    void reset(int n) {
      if(n <= upperbound) count = n;
    }

    int run() {
      if(count > lowerbound) return count--;
      else return lowerbound;
    }
```

```
  cout << NS::i * NS::j << "\n";

  // use NS namespace
  using namespace NS;

  cout << i * j;

  return 0;
}
```

This program produces the following output:

```
100
100
```

Here, **NS** is split into two pieces. However, the contents of each piece are still within the same namespace, that is, **NS**.

A namespace must be declared outside of all other scopes. This means that you cannot declare namespaces that are localized to a function, for example. There is, however, one exception: a namespace can be nested within another. Consider this program:


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

namespace NS1 {
  int i;
  namespace NS2 { // a nested namespace
    int j;
  }
}

int main()
{
  NS1::i = 19;
  // NS2::j = 10; Error, NS2 is not in view
  NS1::NS2::j = 10; // this is right

  cout << NS1::i << " "<<  NS1::NS2::j << "\n";

  // use NS1
```

```
  cout << "Enter a number: ";

  cin >> val;
  cout << "This is your number: ";
  cout << hex << val;
  return 0;
}
```

Here, **cin**, **cout**, and **hex** may be used directly, but the rest of the **std** namespace has not been brought into view.

As explained, the original C++ library was defined in the global namespace. If you will be converting older C++ programs, then you will need to either include a **using namespace std** statement or qualify each reference to a library member with **std::**. This is especially important if you are replacing old **.H** header files with the new-style headers. Remember, the old **.H** headers put their contents into the global namespace, the new-style headers put their contents into the **std** namespace.

Creating Conversion Functions

In some situations, you will want to use an object of a class in an expression involving other types of data. Sometimes, overloaded operator functions can provide the means of doing this. However, in other cases, what you want is a simple type conversion from the class type to the target type. To handle these cases, C++ allows you to create custom *conversion functions*. A conversion function converts your class into a type compatible with that of the rest of the expression. The general format of a type conversion function is

operator *type*( ) { return *value*; }

Here, *type* is the target type that you are converting your class to, and *value* is the value of the class after conversion. Conversion functions return data of type *type*, and no other return type specifier is allowed. Also, no parameters may be included. A conversion function must be a member of the class for which it is defined. Conversion functions are inherited and they may be virtual.

The following illustration of a conversion function uses the **stack** class first developed in Chapter 11. Suppose that you want to be able to use objects of type **stack** within an integer expression. Further, suppose that the value of a **stack** object used in an integer expression is the number of values currently on the stack. (You might want

```
  tos--;
  return stck[tos];
}

int main()
{
  stack stck;
  int i, j;

  for(i=0; i<20; i++)  stck.push(i);

  j = stck; // convert to integer

  cout << j << " items on stack.\n";

  cout << SIZE - stck << " spaces open.\n";
  return 0;
}
```

This program displays this output:

```
20 items on stack.
80 spaces open.
```

As the program illustrates, when a **stack** object is used in an integer expression, such as **j = stck**, the conversion function is applied to the object. In this specific case, the conversion function returns the value 20. Also, when **stck** is subtracted from **SIZE**, the conversion function is also called.

Here is another example of a conversion function. This program creates a class called **pwr( )** that stores and computes the outcome of some number raised to some power. It stores the result as a **double**. By supplying a conversion function to type **double** and returning the result, you can use objects of type **pwr** in expressions involving other **double** values.

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

class pwr {
  double b;
  int e;
  double val;
```

```
   This program won't compile.
*/
#include <iostream>
using namespace std;

class Demo {
  int i;
public:
  int geti() const {
    return i; // ok
  }

  void seti(int x) const {
    i = x; // error!
  }
};

int main()
{
  Demo ob;

  ob.seti(1900);
  cout << ob.geti();
  return 0;
}
```


This program will not compile because **seti( )** is declared as **const**. This means that it is not allowed to modify the invoking object. Since it attempts to change **i**, the program is in error. In contrast, since **geti( )** does not attempt to modify **i**, it is perfectly acceptable.

Sometimes there will be one or more members of a class that you want a **const** function to be able to modify even though you don't want the function to be able to modify any of its other members. You can accomplish this through the use of **mutable**. It overrides **const**ness. That is, a **mutable** member can be modified by a **const** member function. For example:

```
// Demonstrate mutable.
#include <iostream>
using namespace std;

class Demo {
```

Explicit Constructors

As explained in Chapter 12, any time you have a constructor that requires only one argument, you can use either *ob*(*x*) or *ob* = *x* to initialize an object. The reason for this is that whenever you create a constructor that takes one argument, you are also implicitly creating a conversion from the type of that argument to the type of the class. But there may be times when you do not want this automatic conversion to take place. For this purpose, C++ defines the keyword **explicit**. To understand its effects, consider the following program.

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

class myclass {
  int a;
public:
  myclass(int x) { a = x; }
  int geta() { return a; }
};

int main()
{
  myclass ob = 4; // automatically converted into myclass(4)

  cout << ob.geta();

  return 0;
}
```


Here, the constructor for **myclass** takes one parameter. Pay special attention to how **ob** is declared in **main( )**. The statement

```
myclass ob = 4; // automatically converted into myclass(4)
```

is automatically converted into a call to the **myclass** constructor with 4 being the argument. That is, the preceding statement is handled by the compiler as if it were written like this:

```
myclass ob(4);
```

If you do not want this implicit conversion to be made, you can prevent it by using **explicit**. The **explicit** specifier applies only to constructors. A constructor specified as **explicit** will only be used when an initialization uses the normal constructor syntax. It will not perform any automatic conversion. For example, by declaring the **myclass** constructor as **explicit**, the automatic conversion will not be supplied. Here is **myclass( )** declared as **explicit**.

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

class myclass {
  int a;
public:
  explicit myclass(int x) { a = x; }
  int geta() { return a; }
};
```

Now, only constructors of the form

```
myclass ob(4);
```

will be allowed and a statement like

```
myclass ob = 4; // now in error
```

will be invalid.

Using the asm Keyword

While C++ is a comprehensive and powerful programming language, there are a few highly specialized situations that it cannot handle. (For example, there is no C++ statement that disables interrupts.) To accommodate special situations, C++ provides a "trap door" that allows you to drop into assembly code at any time, bypassing the C++ compiler entirely. This "trap door" is the **asm** statement. Using **asm**, you can embed assembly language directly into your C++ program. This assembly code is compiled without any modification, and it becomes part of your program's code at the point at which the **asm** statement occurs.

The general form of the **asm** keyword is shown here:

asm ("*op-code*");

where *op-code* is the assembly language instruction that will be embedded in your program. However, several compilers also allow the following forms of **asm**:

asm *instruction* ;
asm *instruction newline*
asm {
  instruction sequence
}

Here, *instruction* is any valid assembly language instruction. Because of the implementation-specific nature of **asm**, you must check the documentation that came with your compiler for details.

At the time of this writing, Microsoft's Visual C++ uses _ _**asm** for embedding assembly code. It is otherwise similar to **asm**.

Here is a simple (and fairly "safe") example that uses the **asm** keyword:

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

int main()
{
  asm int 5; // generate interrupt 5

  return 0;
}
```


When run under DOS, this program generates an INT 5 instruction, which invokes the print-screen function.

Caution *A thorough working knowledge of assembly language programming is required for using the **asm** statement. If you are not proficient with assembly language, it is best to avoid using **asm** because very nasty errors may result.*

Linkage Specification

In C++ you can specify how a function is linked into your program. By default, functions are linked as C++ functions. However, by using a *linkage specification*, you can cause a function to be linked for a different type of language. The general form of a linkage specifier is

extern "*language*" *function-prototype*

```
  // reading 0x75 42.73 OK
  ins >> hex >> i;
  ins >> f;
  ins >> str;

  cout << hex << i << " " << f << " " << str;

  return 0;
}
```

If you want only part of a string to be used for input, use this form of the **istrstream** constructor:

istrstream *istr*(const char **buf*, streamsize *size*);

Here, only the first *size* elements of the array pointed to by *buf* will be used. This string need not be null terminated, since it is the value of *size* that determines the size of the string.

Streams linked to memory behave just like those linked to other devices. For example, the following program demonstrates how the contents of any text array can be read. When the end of the array (same as end-of-file) is reached, **ins** will be false.


```
/* This program shows how to read the contents of any
   array that contains text. */
#include <iostream>
#include <strstream>
using namespace std;

int main()
{
  char s[] = "10.23 this is a test <<>><<?!\n";

  istrstream ins(s);

  char ch;

  /* This will read and display the contents
     of any text array. */

  ins.unsetf(ios::skipws); // don't skip spaces
  while (ins) { // false when end of array is reached
```

In C, it is not an error to declare a global variable several times, even though this is bad programming practice. In C++, it is an error.

In C, an identifier will have at least 31 significant characters. In C++, all characters are significant. However, from a practical point of view, extremely long identifiers are unwieldy and seldom needed.

In C, although it is unusual, you can call **main( )** from within your program. This is not allowed by C++.

In C, you cannot take the address of a **register** variable. In C++, this is allowed.

In C, if no type specifier is present in some types of declaration statements, the type **int** is assumed. This "default-to-int" rule no longer applies to C++. (Future versions of C are also expected to drop the "default-to-int" rule.)

This chapter explores what is considered by many to be the most important new feature added to C++ in recent years: the *standard template library* (*STL*). The inclusion of the STL was one of the major efforts that took place during the standardization of C++. It provides general-purpose, templatized classes and functions that implement many popular and commonly used algorithms and data structures, including, for example, support for vectors, lists, queues, and stacks. It also defines various routines that access them. Because the STL is constructed from template classes, the algorithms and data structures can be applied to nearly any type of data.

The STL is a complex piece of software engineering that uses some of C++'s most sophisticated features. To understand and use the STL, you must have a complete understanding of the C++ language, including pointers, references, and templates. Frankly, the template syntax that describes the STL can seem quite intimidating—although it looks more complicated than it actually is. While there is nothing in this chapter that is any more difficult than the material in the rest of this book, don't be surprised or dismayed if you find the STL confusing at first. Just be patient, study the examples, and don't let the unfamiliar syntax override the STL's basic simplicity.

The purpose of this chapter is to present an overview of the STL, including its design philosophy, organization, constituents, and the programming techniques needed to use it. Because the STL is a large library, it is not possible to discuss all of its features here. However, a complete reference to the STL is provided in Part Four.

This chapter also describes one of C++'s most important new classes: **string**. The **string** class defines a string data type that allows you to work with character strings much as you do other data types: using operators. The **string** class is closely related to the STL.

An Overview of the STL

Although the standard template library is large and its syntax can be intimidating, it is actually quite easy to use once you understand how it is constructed and what elements it employs. Therefore, before looking at any code examples, an overview of the STL is warranted.

At the core of the standard template library are three foundational items: *containers*, *algorithms*, and *iterators*. These items work in conjunction with one another to provide off-the-shelf solutions to a variety of programming problems.

Containers

Containers are objects that hold other objects, and there are several different types. For example, the **vector** class defines a dynamic array, **deque** creates a double-ended queue, and **list** provides a linear list. These containers are called *sequence containers* because in STL terminology, a sequence is a linear list. In addition to the basic

Term	Represents
BiIter	Bidirectional iterator
ForIter	Forward iterator
InIter	Input iterator
OutIter	Output iterator
RandIter	Random access iterator

Other STL Elements

In addition to containers, algorithms, and iterators, the STL relies upon several other standard components for support. Chief among these are allocators, predicates, comparison functions, and function objects.

Each container has defined for it an *allocator*. Allocators manage memory allocation for a container. The default allocator is an object of class **allocator**, but you can define your own allocators if needed by specialized applications. For most uses, the default allocator is sufficient.

Several of the algorithms and containers use a special type of function called a *predicate*. There are two variations of predicates: unary and binary. A *unary* predicate takes one argument, while a *binary* predicate has two. These functions return true/false results. But the precise conditions that make them return true or false are defined by you. For the rest of this chapter, when a unary predicate function is required, it will be notated using the type **UnPred**. When a binary predicate is required, the type **BinPred** will be used. In a binary predicate, the arguments are always in the order of *first*,*second*. For both unary and binary predicates, the arguments will contain values of the type of objects being stored by the container.

Some algorithms and classes use a special type of binary predicate that compares two elements. Comparison functions return true if their first argument is less than their second. Comparison functions will be notated using the type **Comp**.

In addition to the headers required by the various STL classes, the C++ standard library includes the **<utility>** and **<functional>** headers, which provide support for the STL. For example, the template class **pair**, which can hold a pair of values, is defined in **<utility>**. We will make use of **pair** later in this chapter.

The templates in **<functional>** help you construct objects that define **operator( )**. These are called *function objects* and they may be used in place of function pointers in many places. There are several predefined function objects declared within **<functional>**. They are shown here:

plus	minus	multiplies	divides	modulus
negate	equal_to	not_equal_to	greater	greater_equal
less	less_equal	logical_and	logical_or	logical_not

```
  return 0;
}
```

The output of this program is shown here:

```
Size = 10
Current Contents:
a b c d e f g h i j

Expanding vector
Size now = 20
Current contents:
a b c d e f g h i j k l m n o p q r s t

Modified Contents:
A B C D E F G H I J K L M N O P Q R S T
```

Let's look at this program carefully. In **main( )** a character vector called **v** is created with an initial capacity of 10. That is, **v** initially contains 10 elements. This is confirmed by calling the **size( )** member function. Next, these 10 elements are initialized to the characters a through j and the contents of **v** are displayed. Notice that the standard array subscripting notation is employed. Next, 10 more elements are added to the end of **v** using the **push_back( )** function. This causes **v** to grow in order to accommodate the new elements. As the output shows, its size after these additions is 20. Finally, the values of **v**'s elements are altered using standard subscripting notation.

There is one other point of interest in this program. Notice that the loops that display the contents of **v** use as their target value **v.size( )**. One of the advantages that vectors have over arrays is that it is possible to find the current size of a vector. As you can imagine, this can be quite useful in a variety of situations.

Accessing a Vector Through an Iterator

As you know, arrays and pointers are tightly linked in C++. An array can be accessed either through subscripting or through a pointer. The parallel to this in the STL is the link between vectors and iterators. You can access the members of a vector using subscripting or through the use of an iterator. The following example shows how.

```
// Access the elements of a vector through an iterator.
#include <iostream>
#include <vector>
#include <cctype>
```

stored are determined automatically by the compiler rather than being explicitly specified by you.

The following program illustrates the basics of using a map. It stores key/value pairs that show the mapping between the uppercase letters and their ASCII character codes. Thus, the key is a character and the value is an integer. The key/value pairs stored are

A 65
B 66
C 67

and so on. Once the pairs have been stored, you are prompted for a key (i.e., a letter between A and Z), and the ASCII code for that letter is displayed.

```
// A simple map demonstration.
#include <iostream>
#include <map>
using namespace std;

int main()
{
  map<char, int> m;
  int i;

  // put pairs into map
  for(i=0; i<26; i++) {
    m.insert(pair<char, int>('A'+i, 65+i));
  }

  char ch;
  cout << "Enter key: ";
  cin >> ch;

  map<char, int>::iterator p;

  // find value given key
  p = m.find(ch);
  if(p != m.end())
    cout << "Its ASCII value is  " << p->second;
  else
    cout << "Key not in map.\n";
```

```
Input sequence:
The STL is power programming.
Result after removing spaces:
TheSTLispowerprogramming.

Input sequence:
The STL is power programming.
Result after replacing spaces with colons:
The:STL:is:power:programming.
```

Reversing a Sequence

An often useful algorithm is **reverse( )**, which reverses a sequence. Its general form is

template <class BiIter> void reverse(BiIter *start*, BiIter *end*);

The **reverse( )** algorithm reverses the order of the range specified by *start* and *end*.

The following program demonstrates **reverse( )**.


```
// Demonstrate reverse.
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  vector<int> v;
  int i;

  for(i=0; i<10; i++) v.push_back(i);

  cout << "Initial: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";
  cout << endl;

  reverse(v.begin(), v.end());

  cout << "Reversed: ";
  for(i=0; i<v.size(); i++) cout << v[i] << " ";

  return 0;
}
```

Here, *binfunc_obj* is a binary function object. **bind1st( )** returns a unary function object that has *binfunc_obj*'s left-hand operand bound to *value*. **bind2nd( )** returns a unary function object that has *binfunc_obj*'s right-hand operand bound to *value*. The **bind2nd( )** binder is by far the most commonly used. In either case, the outcome of a binder is a unary function object that is bound to the value specified.

To demonstrate the use of a binder, we will use the **remove_if( )** algorithm. It removes elements from a sequence based upon the outcome of a predicate. It has this prototype:

```
template <class ForIter, class UnPred>
  ForIter remove_if(ForIter start, ForIter end, UnPred func);
```

The algorithm removes elements from the sequence defined by *start* and *end* if the unary predicate defined by *func* is true. The algorithm returns a pointer to the new end of the sequence which reflects the deletion of the elements.

The following program removes all values from a sequence that are greater than the value 8. Since the predicate required by **remove_if( )** is unary, we cannot simply use the **greater( )** function object as-is because **greater( )** is a binary object. Instead, we must bind the value 8 to the second argument of **greater( )** using the **bind2nd( )** binder, as shown in the program.


```
// Demonstrate bind2nd().
#include <iostream>
#include <list>
#include <functional>
#include <algorithm>
using namespace std;

int main()
{
  list<int> lst;
  list<int>::iterator p, endp;

  int i;

  for(i=1; i < 20; i++) lst.push_back(i);

  cout << "Original sequence:\n";
  p = lst.begin();
  while(p != lst.end()) {
    cout << *p << " ";
    p++;
```

```
string &assign(const string &strob, size_type start, size_type num);
string &assign(const char *str, size_type num);
```

In the first form, *num* characters from *strob* beginning at the index specified by *start* will be assigned to the invoking object. In the second form, the first *num* characters of the null-terminated string *str* are assigned to the invoking object. In each case, a reference to the invoking object is returned. Of course, it is much easier to use the = to assign one entire string to another. You will need to use the **assign( )** function only when assigning a partial string.

You can append part of one string to another using the **append( )** member function. Two of its forms are shown here:

```
string &append(const string &strob, size_type start, size_type num);
string &append(const char *str, size_type num);
```

Here, *num* characters from *strob* beginning at the index specified by *start* will be appended to the invoking object. In the second form, the first *num* characters of the null-terminated string *str* are appended to the invoking object. In each case, a reference to the invoking object is returned. Of course, it is much easier to use the + to append one entire string to another. You will need to use the **append( )** function only when appending a partial string.

You can insert or replace characters within a string using **insert( )** and **replace( )**. The prototypes for their most common forms are shown here:

```
string &insert(size_type start, const string &strob);
string &insert(size_type start, const string &strob,
               size_type insStart, size_type num);
string &replace(size_type start, size_type num, const string &strob);
string &replace(size_type start, size_type orgNum, const string &strob,
                size_type replaceStart, size_type replaceNum);
```

The first form of **insert( )** inserts *strob* into the invoking string at the index specified by *start*. The second form of **insert( )** function inserts *num* characters from *strob* beginning at *insStart* into the invoking string at the index specified by *start*.

Beginning at *start*, the first form of **replace( )** replaces *num* characters from the invoking string, with *strob*. The second form replaces *orgNum* characters, beginning at *start*, in the invoking string with the *replaceNum* characters from the string specified by *strob* beginning at *replaceStart*. In both cases, a reference to the invoking object is returned.

You can remove characters from a string using **erase( )**. One of its forms is shown here:

```
Match found at 0
Remaining string is:
Quick of Mind, Strong of Body, Pure of Heart

Match found at 15
Remaining string is:
Strong of Body, Pure of Heart

Match found at 31
Remaining string is:
Pure of Heart

Match found at 36
Remaining string is:
of Heart
```

Comparing Strings

To compare the entire contents of one string object to another, you will normally use the overloaded relational operators described earlier. However, if you want to compare a portion of one string to another, you will need to use the **compare( )** member function, shown here:

```
int compare(size_type start, size_type num, const string &strob) const;
```

Here, *num* characters in *strob*, beginning at *start*, will be compared against the invoking string. If the invoking string is less than *strob*, **compare( )** will return less than zero. If the invoking string is greater than *strob*, it will return greater than zero. If *strob* is equal to the invoking string, **compare( )** will return zero.

Obtaining a Null-Terminated String

Although **string** objects are useful in their own right, there will be times when you will need to obtain a null-terminated character-array version of the string. For example, you might use a **string** object to construct a filename. However, when opening a file, you will need to specify a pointer to a standard, null-terminated string. To solve this problem, the member function **c_str( )** is provided. Its prototype is shown here:

```
const char *c_str( ) const;
```

This function returns a pointer to a null-terminated version of the string contained in the invoking **string** object. The null-terminated string must not be altered. It is also not guaranteed to be valid after any other operations have taken place on the **string** object.

Strings Are Containers

The **string** class meets all of the basic requirements necessary to be a container. Thus, it supports the common container functions, such as **begin( )**, **end( )**, and **size( )**. It also supports iterators. Therefore, a **string** object can also be manipulated by the STL algorithms. Here is a simple example:

```
// Strings as containers.
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

int main()
{
  string str1("Strings handling is easy in C++");
  string::iterator p;
  int i;

  // use size()
  for(i=0; i<str1.size(); i++)
    cout << str1[i];
  cout << endl;

  // use iterator
  p = str1.begin();
  while(p != str1.end())
    cout << *p++;
  cout << endl;

  // use the count() algorithm
  i = count(str1.begin(), str1.end(), 'i');
  cout << "There are " << i << " i's in str1\n";

  // use transform() to upper case the string
  transform(str1.begin(), str1.end(), str1.begin(),
            toupper);
  p = str1.begin();
  while(p != str1.end())
    cout << *p++;
  cout << endl;
```

This chapter describes the C-based I/O functions. These functions are defined by Standard C and Standard C++. While you will usually want to use C++'s object-oriented I/O system for new code, there is no fundamental reason that you cannot use the C I/O functions in a C++ program when you deem it appropriate. The functions in this chapter were first specified by the ANSI C standard, and they are commonly referred to collectively as the ANSI C I/O system.

The header associated with the C-based I/O functions is called **<cstdio>**. (A C program must use the header file **stdio.h**.) This header defines several macros and types used by the file system. The most important type is **FILE**, which is used to declare a file pointer. Two other types are **size_t** and **fpos_t**. The **size_t** type (usually some form of unsigned integer) defines an object that is capable of holding the size of the largest file allowed by the operating environment. The **fpos_t** type defines an object that can hold all information needed to uniquely specify every position within a file. The most commonly used macro defined by the headers is **EOF**, which is the value that indicates end-of-file.

Many of the I/O functions set the built-in global integer variable **errno** when an error occurs. Your program can check this variable when an error occurs to obtain more information about the error. The values that **errno** may take are implementation dependent.

For an overview of the C-based I/O system, see Chapters 8 and 9 in Part One.

Note *This chapter describes the character-based I/O functions. These are the functions that were originally defined for Standard C and C++ and are, by far, the most widely used. In C99, several wide-character (**wchar_t**) functions were added, and they are briefly described in Chapter 31.*

clearerr

```
#include <cstdio>
void clearerr(FILE *stream);
```

The **clearerr( )** function resets (i.e., sets to zero) the error flag associated with the stream pointed to by *stream*. The end-of-file indicator is also reset.

The error flags for each stream are initially set to zero by a successful call to **fopen( )**. Once an error has occurred, the flags stay set until an explicit call to either **clearerr( )** or **rewind( )** is made.

File errors can occur for a wide variety of reasons, many of which are system dependent. The exact nature of the error can be determined by calling **perror( )**, which displays what error has occurred (see **perror( )**).

Related functions are **feof( )**, **ferror( )**, and **perror( )**.

fclose

```
#include <cstdio>
int fclose(FILE *stream);
```

The **fclose( )** function closes the file associated with *stream* and flushes its buffer. After an **fclose( )**, *stream* is no longer connected with the file, and any automatically allocated buffers are deallocated.

If **fclose( )** is successful, zero is returned; otherwise **EOF** is returned. Trying to close a file that has already been closed is an error. Removing the storage media before closing a file will also generate an error, as will lack of sufficient free disk space.

Related functions are **fopen( )**, **freopen( )**, and **fflush( )**.

feof

```
#include <cstdio>
int feof(FILE *stream);
```

The **feof( )** function checks the file position indicator to determine if the end of the file associated with *stream* has been reached. A nonzero value is returned if the file position indicator is at end-of-file; zero is returned otherwise.

Once the end of the file has been reached, subsequent read operations will return **EOF** until either **rewind( )** is called or the file position indicator is moved using **fseek( )**.

The **feof( )** function is particularly useful when working with binary files because the end-of-file marker is also a valid binary integer. Explicit calls must be made to **feof( )** rather than simply testing the return value of **getc( )**, for example, to determine when the end of a binary file has been reached.

Related functions are **clearerr( )**, **ferror( )**, **perror( )**, **putc( )**, and **getc( )**.

ferror

```
#include <cstdio>
int ferror(FILE *stream);
```

The **ferror( )** function checks for a file error on the given *stream*. A return value of zero indicates that no error has occurred, while a nonzero value means an error.

The error flags associated with *stream* will stay set until either the file is closed, or **rewind( )** or **clearerr( )** is called.

The **freopen( )** function associates an existing stream with a different file. The new file's name is pointed to by *fname*, the access mode is pointed to by *mode*, and the stream to be reassigned is pointed to by *stream*. The string *mode* uses the same format as **fopen( )**; a complete discussion is found in the **fopen( )** description.

When called, **freopen( )** first tries to close a file that may currently be associated with *stream*. However, if the attempt to close the file fails, the **freopen( )** function still continues to open the other file.

The **freopen( )** function returns a pointer to *stream* on success and a null pointer otherwise.

The main use of **freopen( )** is to redirect the system defined files **stdin**, **stdout**, and **stderr** to some other file.

Related functions are **fopen( )** and **fclose( )**.

fscanf

```
#include <cstdio>
int fscanf(FILE *stream, const char *format, ...);
```

The **fscanf( )** function works exactly like the **scanf( )** function, except that it reads the information from the stream specified by *stream* instead of **stdin**. See **scanf( )** for details.

The **fscanf( )** function returns the number of arguments actually assigned values. This number does not include skipped fields. A return value of **EOF** means that a failure occurred before the first assignment was made.

Related functions are **scanf( )** and **fprintf( )**.

fseek

```
#include <cstdio>
int fseek(FILE *stream, long offset, int origin);
```

The **fseek( )** function sets the file position indicator associated with stream according to the values of *offset* and *origin*. Its purpose is to support random-access I/O operations. The *offset* is the number of bytes from *origin* to seek to. The values for *origin* must be one of these macros (defined in **<cstdio>**):

The **ftell( )** function returns –1 when an error occurs. If the stream is incapable of random seeks—if it is a modem, for instance—the return value is undefined.

Related functions are **fseek( )** and **fgetpos( )**.

fwrite

```
#include <cstdio>
size_t fwrite(const void *buf, size_t size,
              size_t count, FILE *stream);
```

The **fwrite( )** function writes *count* number of objects, each object being *size* bytes in length, to the stream pointed to by *stream* from the character array pointed to by *buf*. The file position indicator is advanced by the number of characters written.

The **fwrite( )** function returns the number of items actually written, which, if the function is successful, will equal the number requested. If fewer items are written than are requested, an error has occurred. For text streams, various character translations may take place but will have no effect upon the return value.

Related functions are **fread( )**, **fscanf( )**, **getc( )**, and **fgetc( )**.

getc

```
#include <cstdio>
int getc(FILE *stream);
```

The **getc( )** function returns the next character from the input stream and increments the file position indicator. The character is read as an **unsigned char** that is converted to an integer.

If the end of the file is reached, **getc( )** returns **EOF**. However, since **EOF** is a valid integer value, when working with binary files you must use **feof( )** to check for the end-of-file character. If **getc( )** encounters an error, **EOF** is also returned. If working with binary files, you must use **ferror( )** to check for file errors.

The functions **getc( )** and **fgetc( )** are identical, and in most implementations **getc( )** is simply defined as the macro shown here.

```
#define getc(fp) fgetc(fp)
```

This causes the **fgetc( )** function to be substituted for the **getc( )** macro.

Related functions are **fputc( )**, **fgetc( )**, **putc( )**, and **fopen( )**.

remove

```
#include <cstdio>
int remove(const char *fname);
```

The **remove( )** function erases the file specified by *fname*. It returns zero if the file was successfully deleted and nonzero if an error occurred.

A related function is **rename( )**.

rename

```
#include <cstdio>
int rename(const char *oldfname, const char *newfname);
```

The **rename( )** function changes the name of the file specified by *oldfname* to *newfname*. The *newfname* must not match any existing directory entry.

The **rename( )** function returns zero if successful and nonzero if an error has occurred.

A related function is **remove( )**.

rewind

```
#include <cstdio>
void rewind(FILE *stream);
```

The **rewind( )** function moves the file position indicator to the start of the specified stream. It also clears the end-of-file and error flags associated with *stream*. It has no return value.

A related function is **fseek( )**.

scanf

```
#include <cstdio>
int scanf(const char *format, ...);
```

```
scanf("%20s", address);
```

If the input stream were greater than 20 characters, a subsequent call to input would begin where this call left off. Input for a field may terminate before the maximum field length is reached if a white space is encountered. In this case, **scanf( )** moves on to the next field.

Although spaces, tabs, and newlines are used as field separators, when reading a single character, these are read like any other character. For example, with an input stream of **x y**,

```
scanf("%c%c%c", &a, &b, &c);
```

will return with the character x in **a**, a space in **b** and the character y in **c**.

Beware: Any other characters in the control string—including spaces, tabs, and newlines—will be used to match and discard characters from the input stream. Any character that matches is discarded. For example, given the input stream **10t20**,

```
scanf("%dt%d", &x, &y);
```

will place 10 into **x** and 20 into **y**. The **t** is discarded because of the **t** in the control string.

Another feature of **scanf( )** is called a *scanset*. A scanset defines a set of characters that will be read by **scanf( )** and assigned to the corresponding character array. A scanset is defined by putting the characters you want to scan for inside square brackets. The beginning square bracket must be prefixed by a percent sign. For example, this scanset tells **scanf( )** to read only the characters A, B, and C:

```
%[ABC]
```

When a scanset is used, **scanf( )** continues to read characters and put them into the corresponding character array until a character that is not in the scanset is encountered. The corresponding variable must be a pointer to a character array. Upon return from **scanf( )**, the array will contain a null-terminated string comprised of the characters read.

You can specify an inverted set if the first character in the set is a **^**. When the **^** is present, it instructs **scanf( )** to accept any character that *is not* defined by the scanset.

You can specify a range using a hyphen. For example, this tells **scanf( )** to accept the characters A through Z.

The standard function library has a rich and varied set of string and character handling functions. The string functions operate on null-terminated arrays of characters and require the header **<cstring>**. The character functions use the header **<cctype>**. C programs must use the header files **string.h** and **ctype.h**.

Because C/C++ has no bounds checking on array operations, it is the programmer's responsibility to prevent an array overflow. Neglecting to do so may cause your program to crash.

In C/C++, a *printable character* is one that can be displayed on a terminal. These are usually the characters between a space (0x20) and tilde (0xFE). *Control characters* have values between (0) and (0x1F) as well as DEL (0x7F).

For historical reasons, the parameters to the character functions are integers, but only the low-order byte is used; the character functions automatically convert their arguments to **unsigned char**. However, you are free to call these functions with character arguments because characters are automatically elevated to integers at the time of the call.

The header **<cstring>** defines the **size_t** type, which is essentially the same as **unsigned**.

This chapter describes only those functions that operate on characters of type char. These are the functions originally defined by Standard C and C++, and they are by far the most widely used and supported. Wide character functions that operate on characters of type **wchar_t** are discussed in Chapter 31.

isalnum

```
#include <cctype>
int isalnum(int ch);
```

The **isalnum( )** function returns nonzero if its argument is either a letter of the alphabet or a digit. If the character is not alphanumeric, zero is returned.

Related functions are **isalpha( )**, **iscntrl( )**, **isdigit( )**, **isgraph( )**, **isprint( )**, **ispunct( )**, and **isspace( )**.

isalpha

```
#include <cctype>
int isalpha(int ch);
```

The **isalpha( )** function returns nonzero if *ch* is a letter of the alphabet; otherwise zero is returned. What constitutes a letter of the alphabet may vary from language to language. For English, these are the upper- and lowercase letters A through Z.

The **strlen( )** function returns the length of the null-terminated string pointed to by *str*. The null terminator is not counted.

Related functions are **memcpy( )**, **strchr( )**, **strcmp( )**, and **strncmp( )**.

strncat

```
#include <cstring>
char *strncat(char *str1, const char *str2, size_t count);
```

The **strncat( )** function concatenates not more than *count* characters of the string pointed to by *str2* to the string pointed to by *str1* and terminates *str1* with a null. The null terminator originally ending *str1* is overwritten by the first character of *str2*. The string *str2* is untouched by the operation. If the strings overlap, the behavior is undefined.

The **strncat( )** function returns *str1*.

Remember that no bounds checking takes place, so it is the programmer's responsibility to ensure that *str1* is large enough to hold both its original contents and also those of *str2*.

Related functions are **strcat( )**, **strnchr( )**, **strncmp( )**, and **strncpy( )**.

strncmp

```
#include <cstring>
int strncmp(const char *str1, const char *str2, size_t count);
```

The **strncmp( )** function lexicographically compares not more than *count* characters from the two null-terminated strings and returns an integer based on the outcome, as shown here:

Value	Meaning
Less than zero	*str1* is less than *str2*.
Zero	*str1* is equal to *str2*.
Greater than zero	*str1* is greater than *str2*.

If there are less than *count* characters in either string, the comparison ends when the first null is encountered.

Related functions are **strcmp( )**, **strnchr( )**, and **strncpy( )**.

strspn

```
#include <cstring>
size_t strspn(const char *str1, const char *str2);
```

The **strspn( )** function returns the length of the initial substring of the string pointed to by *str1* that is made up of only those characters contained in the string pointed to by *str2*. Stated differently, **strspn( )** returns the index of the first character in the string pointed to by *str1* that does not match any of the characters in the string pointed to by *str2*.

Related functions are **strpbrk( )**, **strrchr( )**, **strstr( )**, and **strtok( )**.

strstr

```
#include <cstring>
char *strstr(const char *str1, const char *str2);
```

The **strstr( )** function returns a pointer to the first occurrence in the string pointed to by *str1* of the string pointed to by *str2*. It returns a null pointer if no match is found.

Related functions are **strchr( )**, **strcspn( )**, **strpbrk( )**, **strspn( )**, **strtok( )**, and **strrchr( )**.

strtok

```
#include <cstring>
char *strtok(char *str1, const char *str2);
```

The **strtok( )** function returns a pointer to the next token in the string pointed to by *str1*. The characters making up the string pointed to by *str2* are the delimiters that determine the token. A null pointer is returned when there is no token to return.

To tokenize a string, the first call to **strtok( )** must have *str1* point to the string being tokenized. Subsequent calls must use a null pointer for *str1*. In this way, the entire string can be reduced to its tokens.

It is possible to use a different set of delimiters for each call to **strtok( )**.

Related functions are **strchr( )**, **strcspn( )**, **strpbrk( )**, **strrchr( )**, and **strspn( )**.

```
double frexp(double num, int *exp);
long double frexp(long double num, int *exp);
```

The **frexp( )** function decomposes the number *num* into a mantissa in the range 0.5 to less than 1, and an integer exponent such that *num* = *mantissa* * 2^{exp}. The mantissa is returned by the function, and the exponent is stored at the variable pointed to by *exp*.

A related function is **ldexp( )**.

ldexp

```
#include <cmath>
float ldexp(float num, int exp);
double ldexp(double num, int exp);
long double ldexp(long double num, int exp);
```

The **ldexp( )** returns the value of *num* * 2^{exp}. If overflow occurs, **HUGE_VAL** is returned.

Related functions are **frexp( )** and **modf( )**.

log

```
#include <cmath>
float log(float num);
double log(double num);
long double log(long double num);
```

The **log( )** function returns the natural logarithm for *num*. A domain error occurs if *num* is negative, and a range error occurs if the argument is zero.

A related function is **log10( )**.

log10

```
#include <cmath>
float log10(float num);
double log10(double num);
long double log10(long double num);
```

LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME

LC_ALL refers to all localization categories. **LC_COLLATE** affects the operation of the **strcoll( )** function. **LC_CTYPE** alters the way the character functions work. **LC_MONETARY** determines the monetary format. **LC_NUMERIC** changes the decimal-point character for formatted input/output functions. Finally, **LC_TIME** determines the behavior of the **strftime( )** function.

The **setlocale( )** function returns a pointer to a string associated with the *type* parameter.

Related functions are **localeconv( )**, **time( )**, **strcoll( )**, and **strftime( )**.

strftime

```
#include <ctime>
size_t strftime(char *str, size_t maxsize, const char *fmt,
                const struct tm *time);
```


The **strftime( )** function places time and date information, along with other information, into the string pointed to by *str* according to the format commands found in the string pointed to by *fmt* and using the broken-down time *time*. A maximum of *maxsize* characters will be placed into *str*.

The **strftime( )** function works a little like **sprintf( )** in that it recognizes a set of format commands that begin with the percent sign (%) and places its formatted output into a string. The format commands are used to specify the exact way various time and date information is represented in *str*. Any other characters found in the format string are placed into *str* unchanged. The time and date displayed are in local time. The format commands are shown in the table below. Notice that many of the commands are case sensitive.

The **strftime( )** function returns the number of characters placed in the string pointed to by *str* or zero if an error occurs.

Command	Replaced By
%a	Abbreviated weekday name
%A	Full weekday name

calloc( )). Using an invalid pointer in the call most likely will destroy the memory management mechanism and cause a system crash.

Related functions are **calloc( )**, **malloc( )**, and **realloc( )**.

malloc

```
#include <cstdlib>
void *malloc(size_t size);
```

The **malloc( )** function returns a pointer to the first byte of a region of memory of size *size* that has been allocated from the heap. If there is insufficient memory in the heap to satisfy the request, **malloc( )** returns a null pointer. It is always important to verify that the return value is not null before attempting to use it. Attempting to use a null pointer will usually result in a system crash.

Related functions are **free( )**, **realloc( )**, and **calloc( )**.

realloc

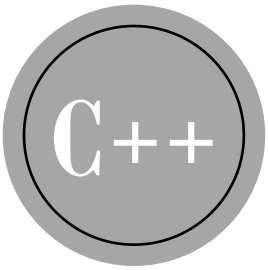
```
#include <cstdlib>
void *realloc(void *ptr, size_t size);
```

The **realloc( )** function changes the size of the previously allocated memory pointed to by *ptr* to that specified by *size*. The value of *size* may be greater or less than the original. A pointer to the memory block is returned because it may be necessary for **realloc( )** to move the block in order to increase its size. If this occurs, the contents of the old block are copied into the new block—no information is lost.

If *ptr* is null, **realloc( )** simply allocates *size* bytes of memory and returns a pointer to it. If *size* is zero, the memory pointed to by *ptr* is freed.

If there is not enough free memory in the heap to allocate *size* bytes, a null pointer is returned, and the original block is left unchanged.

Related functions are **free( )**, **malloc( )**, and **calloc( )**.



Chapter 30

Utility Functions

The standard function library defines several utility functions that provide various commonly used services. They include a number of conversions, variable-length argument processing, sorting and searching, and random number generation. Many of the functions covered here require the use of the header **<cstdlib>**. (A C program must use the header file **stdlib.h**.) In this header are defined **div_t** and **ldiv_t**, which are the types of values returned by **div( )** and **ldiv( )**, respectively. Also defined is the type **size_t**, which is the unsigned value returned by **sizeof**. The following macros are defined:

Macro	Meaning
NULL	A null pointer.
RAND_MAX	The maximum value that can be returned by the **rand( )** function.
EXIT_FAILURE	The value returned to the calling process if program termination is unsuccessful.
EXIT_SUCCESS	The value returned to the calling process if program termination is successful.

If a function requires a different header than **<cstdlib>**, that function description will discuss it.

abort

```
#include <cstdlib>
void abort(void);
```

The **abort( )** function causes immediate abnormal termination of a program. Generally, no files are flushed. In environments that support it, **abort( )** will return an implementation-defined value to the calling process (usually the operating system) indicating failure.

Related functions are **exit( )** and **atexit( )**.

abs

```
#include <cstdlib>
int abs(int num);
long abs(long num);
double abs(double num);
```

```
                size_t num, size_t size,
                int (*compare)(const void *, const void *));
```

The **bsearch( )** function performs a binary search on the sorted array pointed to by *buf* and returns a pointer to the first member that matches the key pointed to by *key*. The number of elements in the array is specified by *num*, and the size (in bytes) of each element is described by *size*.

The function pointed to by *compare* is used to compare an element of the array with the key. The form of the *compare* function must be as follows:

int *func_name*(const void **arg1*, const void **arg2*);

It must return values as described in the following table:

Comparison	Value Returned
arg1 is less than *arg2*	Less than zero
Arg1 is equal to *arg2*	Zero
Arg1 is greater than *arg2*	Greater than zero

The array must be sorted in ascending order with the lowest address containing the lowest element.

If the array does not contain the key, a null pointer is returned.

A related function is **qsort( )**.

div

```
#include <cstdlib>
div_t div(int numerator, int denominator);
ldiv_t div(long numerator, long denominator);
```

The **int** version of **div( )** function returns the quotient and the remainder of the operation *numerator / denominator* in a structure of type **div_t**. The **long** version of **div( )** returns the quotient and remainder in a structure of type **ldiv_t**. The **long** version of **div( )** provides the same capabilities as the **ldiv( )** function.

The structure type **div_t** will have at least these two fields:

```
int quot; /* quotient */
int rem;  /* remainder */
```

The **mblen( )** function returns the length (in bytes) of a multibyte character pointed to by *str*. Only the first *size* number of characters are examined. It returns –1 on error.

If *str* is null, then **mblen( )** returns non-zero if multibyte characters have state-dependent encodings. If they do not, zero is returned.

Related functions are **mbtowc( ) and wctomb( )**.

mbstowcs

```
#include <cstdlib>
size_t mbstowcs(wchar_t *out, const char *in, size_t size);
```

The **mbstowcs( )** function converts the multibyte string pointed to by *in* into a wide character string and puts that result in the array pointed to by *out*. Only *size* number of bytes will be stored in *out*.

The **mbstowcs( )** function returns the number of multibyte characters that are converted. If an error occurs, the function returns –1.

Related functions are **wcstombs( ), mbtowc( ).**

mbtowc

```
#include <cstdlib>
int mbtowc(wchar_t *out, const char *in, size_t size);
```

The **mbtowc( )** function converts the multibyte character in the array pointed to by *in* into its wide character equivalent and puts that result in the object pointed to by *out*. Only *size* number of characters will be examined.

This function returns the number of bytes that are put into *out*. –1 is returned if an error occurs. If *in* is null, then **mbtowc( )** returns non-zero if multibyte characters have state dependencies. If they do not, zero is returned.

Related functions are **mblen( ), wctomb( )**.

qsort

```
#include <cstdlib>
void qsort(void *buf, size_t num, size_t size,
           int (*compare) (const void *, const void *));
```

Function	char Equivalent
int iswalnum(wint_t *ch*)	isalnum( )
int iswalpha(wint_t *ch*)	isalpha( )
int iswcntrl(wint_t *ch*)	iscntrl( )
int iswdigit(wint_t *ch*)	isdigit( )
int iswgraph(wint_t *ch*)	isgraph( )
int iswlower(wint_t *ch*)	islower( )
int iswprint(wint_t *ch*)	isprint( )
int iswpunct(wint_t *c*)	ispunct( )
int iswspace(wint_t *ch*)	isspace( )
int iswupper(wint_t *ch*)	isupper( )
int iswxdigit(wint_t *ch*)	isxdigit( )
wint_t tolower(wint_t *ch*)	tolower( )
wint_t toupper(wint_t *ch*)	toupper( )

Table 31-1. *The Wide-Character Classification Functions*

have. The value in *attr_ob* used to determine if *ch* is a character that has that property. If it is, **iswctype( )** returns nonzero. Otherwise, it returns zero. The following property strings are defined for all execution environments.

alnum	alpha	cntrl	digit
graph	lower	print	punct
space	upper	xdigit	

The following program demonstrates the **wctype( )** and **iswctype( )** functions.

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

int main()
```

Multibyte/Wide-Character Conversion Functions

The Standard C++ function library supplies various functions that support conversions between multibyte and wide characters. These functions, shown in Table 31-6, use the header **<cwchar>**. Many of them are *restartable* versions of the normal multibyte functions. The restartable version utilizes the state information passed to it in a parameter of type **mbstate_t**. If this parameter is null, the function will provide its own **mbstate_t** object.

Function	Description
win_t btowc(int *ch*)	Converts *ch* into its wide-character equivalent and returns the result. Returns **WEOF** on error or if *ch* is not a one-byte, multibyte character.
size_t mbrlen(const char **str*, size_t *num*, mbstate_t **state*)	Restartable version of **mblen( )** as described by *state*. Returns a positive value that indicates the length of the next multibyte character. Zero is returned if the next character is null. A negative value is returned if an error occurs.
size_t mbrtowc(wchar_t **out*, const char **in*, size_t *num*, mbstate_t **state*)	Restartable version of **mbtowc( )** as described by *state*. Returns a positive value that indicates the length of the next multibyte character. Zero is returned if the next character is null. A negative value is returned if an error occurs. If an error occurs, the macro **EILSEQ** is assigned to **errno**.
int mbsinit(const mbstate_t **state*)	Returns true if *state* represents an initial conversion state.
size_t mbsrtowcs(wchar_t **out*, const char ***in*, size_t *num*, mbstate_t *state*)	Restartable version of **mbstowcs( )** as described by *state*. Also, **mbsrtowcs( )** differs from **mbstowcs( )** in that *in* is an indirect pointer to the source array. If an error occurs, the macro **EILSEQ** is assigned to **errno**.

Table 31-6. *Wide-Character/Multibyte Conversion Functions*

Manipulator	Purpose	Input/Output
hex	Turns on **hex** flag.	Input/Output
internal	Turns on **internal** flag.	Output
left	Turns on **left** flag.	Output
noboolalpha	Turns off **boolalpha** flag.	Input/Output
noshowbase	Turns off **showbase** flag.	Output
noshowpoint	Turns off **showpoint** flag.	Output
noshowpos	Turns off **showpos** flag.	Output
noskipws	Turns off **skipws** flag.	Input
nounitbuf	Turns off **unitbuf** flag.	Output
nouppercase	Turns off **uppercase** flag.	Output
oct	Turns on **oct** flag.	Input/Output
resetiosflags (fmtflags *f*)	Turn off the flags specified in *f*.	Input/Output
right	Turns on **right** flag.	Output
scientific	Turns on **scientific** flag.	Output
setbase(int *base*)	Set the number base to *base*.	Input/Output
setfill(int *ch*)	Set the fill character to *ch*.	Output
setiosflags(fmtflags *f*)	Turn on the flags specified in *f*.	Input/output
setprecision (int *p*)	Set the number of digits of precision.	Output
setw(int *w*)	Set the field width to *w*.	Output
showbase	Turns on **showbase** flag.	Output
showpoint	Turns on **showpoint** flag.	Output
showpos	Turns on **showpos** flag.	Output
skipws	Turns on **skipws** flag.	Input
unitbuf	Turns on **unitbuf** flag.	Output
uppercase	Turns on **uppercase** flag.	Output
ws	Skip leading white space.	Input

To use a manipulator that takes a parameter, you must include **<iomanip>**.

Related functions are **precision( )** and **width( )**.

flags

```
#include <iostream>
fmtflags flags() const;
fmtflags flags(fmtflags f);
```

The **flags( )** function is a member of **ios** (inherited from **ios_base**).

The first form of **flags( )** simply returns the current format flags settings of the associated stream.

The second form of **flags( )** sets all format flags associated with a stream as specified by *f*. When you use this version, the bit pattern found in *f* is copied into the format flags associated with the stream. This version also returns the previous settings.

Related functions are **unsetf( )** and **setf( )**.

flush

```
#include <iostream>
ostream &flush();
```

The **flush( )** function is a member of **ostream**.

The **flush( )** function causes the buffer connected to the associated output stream to be physically written to the device. The function returns a reference to its associated stream.

Related functions are **put( )** and **write( )**.

fstream, ifstream, and ofstream

```
#include <fstream>
fstream();
explicit fstream(const char *filename,
                 ios::openmode mode = ios::in | ios::out);
ifstream();
explicit ifstream(const char *filename, ios::openmode mode=ios::in);

ofstream();
explicit ofstream(const char *filename,
                  ios::openmode mode=ios::out | ios::trunc);
```

The **rdstate( )** function is a member of **ios.**

The **rdstate( )** function returns the status of the associated stream. The C++ I/O system maintains status information about the outcome of each I/O operation relative to each active stream. The current state of a stream is held in an object of type **iostate**, in which the following flags are defined:

Name	Meaning
goodbit	No errors occurred.
eofbit	End-of-file is encountered.
failbit	A nonfatal I/O error has occurred.
badbit	A fatal I/O error has occurred.

These flags are enumerated inside **ios** (via **ios_base**).

rdstate( ) returns **goodbit** when no error has occurred; otherwise, an error bit has been set.

Related functions are **eof( )**, **good( )**, **bad( )**, **clear( )**, **setstate( )**, and **fail( )**.

read

```
#include <iostream>
istream &read(char *buf, streamsize num);
```


The **read( )** function is a member of **istream**.

The **read( )** function reads *num* bytes from the associated input stream and puts them in the buffer pointed to by *buf*. If the end of the file is reached before *num* characters have been read, **read( )** simply stops, sets **failbit**, and the buffer contains as many characters as were available. (See **gcount( )**.) **read( )** returns a reference to the stream.

Related functions are **gcount( )**, **readsome( )**, **get( )**, **getline( )**, and **write( )**.

readsome

```
#include <iostream>
streamsize readsome(char *buf, streamsize num);
```

The **readsome( )** function is a member of **istream**.

The **readsome( )** function reads *num* bytes from the associated input stream and puts them in the buffer pointed to by *buf*. If the stream contains less than *num*

str

```
#include <sstream>
string str() const;
void str(string &s)
```

The **str( )** function is a member of **stringstream**, **istringstream**, and **ostringstream**.

The first form of the **str( )** function returns a **string** object that contains the current contents of the string-based stream.

The second form frees the string currently contained in the string stream and substitutes the string referred to by *s*.

Related functions are **get( )** and **put( )**.

stringstream, istringstream, ostringstream

```
#include <sstream>
explicit stringstream(ios::openmode mode = ios::in | ios::out);
explicit stringstream(const string &str,
                      ios::openmode mode = ios::in | ios::out);
explicit istringstream(ios::openmode mode=ios::in);
explicit istringstream(const string str, ios::openmode mode=ios::in);
explict ostringstream(ios::openmode mode=ios::out);
explict ostringstream(const string str, ios::openmode
                      mode=ios::out);
```


The **stringstream( )**, **istringstream( )**, and **ostringstream( )** functions are the constructors of the **stringstream**, **istringstream**, and **ostringstream** classes, respectively. These construct streams that are tied to strings.

The versions of **stringstream( )**, **istringstream( )**, and **ostringstream( )** that specify only the **openmode** parameter create empty streams. The versions that take a **string** parameter initialize the string stream.

Here is an example that demonstrates the use of a string stream.

```
// Demonstrate string streams.
#include <iostream>
#include <sstream>
using namespace std;
```

This page intentionally left blank.

```
explicit multimap(const Comp &cmpfn = Comp( ),
                  const Allocator &a = Allocator( ) );

multimap(const multimap<Key, T, Comp, Allocator> &ob);

template <class InIter> multimap(InIter start, InIter end,
                  const Comp &cmpfn = Comp( ),
                  const Allocator &a = Allocator( ));
```

The first form constructs an empty multimap. The second form constructs a multimap that contains the same elements as *ob*. The third form constructs a multimap that contains the elements in the range specified by *start* and *end*. The function specified by *cmpfn,* if present, determines the ordering of the multimap.

The following comparison operators are defined by **multimap:**

```
==, <, <=, !=, >, >=
```

The member functions contained by **multimap** are shown here. In the descriptions, **key_type** is the type of the key, **T** is the value, and **value_type** represents **pair<Key, T>**.

Member	Description
iterator begin( ); const_iterator begin( ) const;	Returns an iterator to the first element in the multimap.
void clear( );	Removes all elements from the multimap.
size_type count(const key_type &*k*) const;	Returns the number of times *k* occurs in the multimap.
bool empty( ) const;	Returns true if the invoking multimap is empty and false otherwise.
iterator end( ); const_iterator end( ) const;	Returns an iterator to the end of the list.

pair<iterator, iterator>
equal_range(const key_type &*k*);
pair<const_iterator, const_iterator>
equal_range(const key_type &*k*) const;
	Returns a pair of iterators that point to the first and last elements in the multimap that contain the specified key.

void erase(iterator *i*);	Removes the element pointed to by *i*.
void erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*.

The member functions contained by **set** are shown here.

Member	Description
iterator begin( ); const_iterator begin( ) const;	Returns an iterator to the first element in the set.
void clear( );	Removes all elements from the set.
size_type count(const key_type &*k*) const;	Returns the number of times *k* occurs in the set.
bool empty( ) const;	Returns true if the invoking set is empty and false otherwise.
const_iterator end( ) const; iterator end( );	Returns an iterator to the end of the set.
pair<iterator, iterator> equal_range(const key_type &*k*) const;	Returns a pair of iterators that point to the first and last elements in the set that contain the specified key.
void erase(iterator *i*);	Removes the element pointed to by *i*.
void erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*.
size_type erase(const key_type &*k*);	Removes from the set elements that have keys with the value *k*. The number of elements removed is returned.
iterator find(const key_type &*k*) const;	Returns an iterator to the specified key. If the key is not found, then an iterator to the end of the set is returned.
allocator_type get_allocator( ) const;	Returns set's allocator.
iterator insert(iterator *i*, const value_type &*val*);	Inserts *val* at or after the element specified by *i*. Duplicate elements are not inserted. An iterator to the element is returned.
template <class InIter> void insert(InIter *start*, InIter *end*);	Inserts a range of elements. Duplicate elements are not inserted.

fill and fill_n

```
template <class ForIter, class T>
  void fill(ForIter start, ForIter end, const T &val);
template <class ForIter, class Size, class T>
  void fill_n(ForIter start, Size num, const T &val);
```

The **fill( )** and **fill_n( )** algorithms fill a range with the value specified by *val*. For **fill( )** the range is specified by *start* and *end*. For **fill_n( )**, the range begins at *start* and runs for *num* elements.

find

```
template <class InIter, class T>
  InIter find(InIter start, InIter end, const T &val);
```

The **find( )** algorithm searches the range *start* to *end* for the value specified by *val*. It returns an iterator to the first occurrence of the element or to *end* if the value is not in the sequence.

find_end

```
template <class ForIter1, class ForIter2>
  FwdIter1 find_end(ForIter1 start1, ForIter1 end1,
                    ForIter2 start2, ForIter2 end2);
template <class ForIter1, class ForIter2, class BinPred>
  FwdIter1 find_end(ForIter1 start1, ForIter1 end1,
                    ForIter2 start2, ForIter2 end2, BinPred pfn);
```

The **find_end( )** algorithm finds the last iterator of the subsequence defined by *start2* and *end2* within the range *start1* and *end1*. If the sequence is found, an iterator to the last element in the sequence is returned. Otherwise, the iterator *end1* is returned.

The second form allows you to specify a binary predicate that determines when elements match.

find_first_of

```
template <class ForIter1, class ForIter2>
  FwdIter1 find_first_of(ForIter1 start1, ForIter1 end1,
```

swap_ranges

```
template <class ForIter1, class ForIter2>
  ForIter2 swap_ranges(ForIter1 start1, ForIter1 end1,
                       ForIter2 start2);
```

The **swap_ranges( )** algorithm exchanges elements in the range specified by *start1* and *end1* with elements in the sequence beginning at *start2*. It returns a pointer to the end of the sequence specified by *start2*.

transform

```
template <class InIter, class OutIter, class Func>
  OutIter transform(InIter start, InIter end,
                    OutIter result, Func unaryfunc);
template <class InIter1, class InIter2, class OutIter, class Func>
  OutIter transform(InIter1 start1, InIter1 end1,
                    InIter2 start2, OutIter result,
                    Func binaryfunc);
```

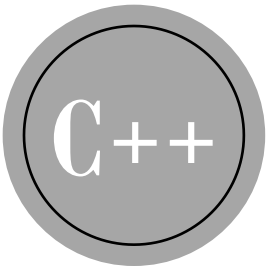
The **transform( )** algorithm applies a function to a range of elements and stores the outcome in *result*. In the first form, the range is specified by *start* and *end*. The function to be applied is specified by *unaryfunc*. This function receives the value of an element in its parameter and it must return its transformation.

In the second form, the transformation is applied using a binary operator function that receives the value of an element from the sequence to be transformed in its first parameter and an element from the second sequence as its second parameter.

Both versions return an iterator to the end of the resulting sequence.

unique and unique_copy

```
template <class ForIter>
  ForIter unique(ForIter start, ForIter end);
template <class ForIter, class BinPred>
  ForIter unique(ForIter start, ForIter end, BinPred pfn);
template <class ForIter, class OutIter>
  OutIter unique_copy(ForIter start, ForIter end, OutIter result);
template <class ForIter, class OutIter, class BinPred>
  OutIter unique_copy(ForIter start, ForIter end, OutIter result,
                      BinPred pfn);
```



Chapter 35

STL Iterators, Allocators, and Function Objects

reverse_iterator

The **reverse_iterator** class supports reverse iterator operations. A reverse iterator operates the opposite of a normal iterator. For example, **++** causes a reverse iterator to back up. Its template definition is shown here:

```
template <class Iter> class reverse_iterator:
  public iterator<iterator_traits<Iter>::iterator_category,
                  iterator_traits<Iter>::value_type,
                  iterator_traits<Iter>::difference_type,
                  iterator_traits<Iter>::pointer,
                  iterator_traits<Iter>::reference>
```

Here, **Iter** is either a random-access iterator or a bidirectional iterator. **reverse_iterator** has the following constructors:

```
reverse_iterator( );
explicit reverse_iterator(Iter itr);
```

Here, *itr* is an iterator that specifies the starting location.

If **Iter** is a random-access iterator, then the following operators are available: **–>**, **+**, **++**, **–**, **––**, *, <, >, <=, >=, **+=**, **–=**, **==**, **!=**, and [ ] . If **Iter** is a bidirectional iterator, then only **–>**, **++**, **––**, *, **==** and **!=** are available.

The **reverse_iterator** class defines a protected member called **current,** which is an iterator to the current location.

The function **base( )** is also defined by **reverse_iterator**. Its prototype is shown here:

```
Iter base( ) const;
```

It returns an iterator to the current location.

istream_iterator

The **istream_iterator** class supports input iterator operations on a stream. Its template definition is shown here:

```
template <class T, class CharType, class Attr = char_traits<CharType>,
          class Dist = ptrdiff_t> class istream_iterator:
  public iterator<input_iterator_tag, T, Dist, const T *, const T &>
```

Here, **T** is the type of data being transferred, and **CharType** is the character type (**char** or **wchar_t**) that the stream is operating upon. **Dist** is a type capable of holding the difference between two addresses. **istream_iterator** has the following constructors:

```
// Use ostream_iterator
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
  ostream_iterator<char> out_it(cout);

  *out_it = 'X';
  out_it++;
  *out_it = 'Y';
  out_it++;
  *out_it = ' ';

  char str[] = "C++ Iterators are powerful.\n";
  char *p = str;

  while(*p) *out_it++ = *p++;

  ostream_iterator<double> out_double_it(cout);
  *out_double_it = 187.23;
  out_double_it++;
  *out_double_it = -102.7;

  return 0;
}
```


The output from this program is shown here:

```
XY C++ Iterators are powerful.
187.23-102.7
```

ostreambuf_iterator

The **ostreambuf_iterator** class supports character output iterator operations on a stream. Its template definition is shown here:

```
template <class CharType, class Attr = char_traits<CharType> >
class ostreambuf_iterator:
  public iterator<output_iterator_tag, void, void, void, void>
```

Member	Description
int compare(size_type *indx*, size_type *len*, const string &*str*, size_type *indx2*, size_type *len2*) const;	Compares a substring of *str* to a substring within the invoking string. The substring in the invoking string begins at *indx* and is *len* characters long. The substring in *str* begins at *indx2* and is *len2* characters long. It returns one of the following: Less than zero if ***this** < *str* Zero if ***this** == *str* Greater than zero if ***this** > *str*
int compare(const CharType **str*) const;	Compares *str* to the invoking string. It returns one of the following: Less than zero if ***this** < *str* Zero if ***this** == *str* Greater than zero if ***this** > *str*
int compare(size_type *indx*, size_type *len*, const CharType **str*, size_type *len2* = npos) const;	Compares a substring of *str* to a substring within the invoking string. The substring in the invoking string begins at *indx* and is *len* characters long. The substring in *str* begins at zero and is *len2* characters long. It returns one of the following: Less than zero if ***this** < *str* Zero if ***this** == *str* Greater than zero if ***this** > *str*
size_type copy(CharType **str*, size_type *len*, size_type *indx* = 0) const;	Beginning at *indx*, copies *len* characters from the invoking string into the character array pointed to by *str*. Returns the number of characters copied.
const CharType *data( ) const;	Returns a pointer to the first character in the invoking string.
bool empty( ) const;	Returns **true** if the invoking string is empty and **false** otherwise.

Table 36-1. *The String Member Functions* (continued)

Member	Description
iterator end( ); const_iterator end( ) const;	Returns an iterator to the end of the string.
iterator erase(iterator *i*);	Removes character pointed to by *i*. Returns an iterator to the character after the one removed.
iterator erase(iterator *start*, iterator *end*);	Removes characters in the range *start* to *end*. Returns an iterator to the character after the last character removed.
string &erase(size_type *indx* = 0, size_type *len* = npos);	Beginning at *indx*, removes *len* characters from the invoking string. Returns ***this**.
size_type find(const string &*str*, size_type *indx* = 0) const;	Returns the index of the first occurrence of *str* within the invoking string. The search begins at index *indx*. **npos** is returned if no match is found.
size_type find(const CharType **str*, size_type *indx* = 0) const;	Returns the index of the first occurrence of *str* within the invoking string. The search begins at index *indx*. **npos** is returned if no match is found.
size_type find(const CharType **str*, size_type *indx*, size_type *len*) const;	Returns the index of the first occurrence of the first *len* characters of *str* within the invoking string. The search begins at index *indx*. **npos** is returned if no match is found.
size_type find(CharType *ch*, size_type *indx* = 0) const;	Returns the index of the first occurrence of *ch* within the invoking string. The search begins at index *indx*. **npos** is returned if no match is found.

Table 36-1. *The String Member Functions* (continued)

Member	Description
string &replace(size_type *indx*, size_type *len*, const string &*str*);	Replaces up to *len* characters in the invoking string, beginning at *indx* with the string in *str*. Returns ***this**.
string &replace(size_type *indx1*, size_type *len1*, const string &*str*, size_type *indx2*, size_type *len2*);	Replaces up to *len1* characters in the invoking string beginning at *indx1* with the *len2* characters from the string in *str* that begin at *indx2*. Returns ***this**.
string &replace(size_type *indx*, size_type *len*, const CharType **str*);	Replaces up to *len* characters in the invoking string, beginning at *indx* with the string in *str*. Returns ***this**.
string &replace(size_type *indx1*, size_type *len1*, const CharType **str*, size_type *len2*);	Replaces up to *len1* characters in the invoking string beginning at *indx1* with the *len2* characters from the string in *str* that begins at *indx2*. Returns ***this**.
string &replace(size_type *indx*, size_type *len1*, size_type *len2*, CharType *ch*);	Replaces up to *len1* characters in the invoking string beginning at *indx* with *len2* characters specified by *ch*. Returns ***this**.
string &replace(iterator *start*, iterator *start*, const string &*str*);	Replaces the range specified by *start* and *end* with *str*. Returns ***this**.
string &replace(iterator *start*, iterator *start*, const CharType **str*);	Replaces the range specified by *start* and *end* with *str*. Returns ***this**.
string &replace(iterator *start*, iterator *end*, const CharType **str*, size_type *len*);	Replaces the range specified by *start* and *end* with the first *len* characters from *str*. Returns ***this**.
string &replace(iterator *start*, interator *end*, size_type *len*, CharType *ch*);	Replaces the range specified by *start* and *end* with the *len* characters specified by *ch*. Returns ***this**.

Table 36-1. *The String Member Functions* (continued)

Function	Description
template <class T> complex<T> polar(const T &*v*, const T &*theta*=0);	Returns a complex number that has the magnitude specified by *v* and a phase angle of *theta*.
template <class T> complex<T> pow(const complex<T> &*b*, int *e*);	Returns b^e.
template <class T> complex<T> pow(const complex<T> &b, const T &*e*);	Returns b^e.
template <class T> complex<T> pow(const complex<T> &*b*, const complex<T> &*e*);	Returns b^e.
template <class T> complex<T> pow(const T &*b*, const complex<T> &*e*);	Returns b^e.
template <class T> T real(const complex<T> &*ob*);	Returns the real component of *ob*.
template <class T> complex<T> sin(const complex<T> &*ob*);	Returns the sine of *ob*.
template <class T> complex<T> sinh(const complex<T> &*ob*);	Returns the hyperbolic sine of *ob*.
template <class T> complex<T> sqrt(const complex<T> &*ob*);	Returns the square root of *ob*.
template <class T> complex<T> tan(const complex<T> &*ob*);	Returns the tangent of *ob*.
template <class T> complex<T> tanh(const complex<T> &*ob*);	Returns the hyperbolic tangent of *ob*.

Table 37-1. *Functions Defined for* ***complex*** *(continued)*

Function	Description
valarray<T> &operator=(const indirect_array<T> &*ob*);	Assigns a subset. Returns a reference to the invoking array.
valarray<T> operator+( ) const;	Unary plus applied to each element in the invoking array. Returns the resulting array.
valarray<T> operator–( ) const;	Unary minus applied to each element in the invoking array. Returns the resulting array.
valarray<T> operator~( ) const;	Unary bitwise NOT applied to each element in the invoking array. Returns the resulting array.
valarray<T> operator!( ) const;	Unary logical NOT applied to each element in the invoking array. Returns the resulting array.
valarray<T> &operator+=(const T &*v*) const;	Adds *v* to each element in the invoking array. Returns a reference to the invoking array.
valarray<T> &operator–=(const T &*v*) const;	Subtracts *v* from each element in the invoking array. Returns a reference to the invoking array.
valarray<T> &operator/=(const T &*v*) const;	Divides each element in the invoking array by *v*. Returns a reference to the invoking array.
valarray<T> &operator*=(const T &*v*) const;	Multiplies each element in the invoking array by *v*. Returns a reference to the invoking array.
valarray<T> &operator%=(const T &*v*) const;	Assigns each element in the invoking array the remainder of a division by *v*. Returns a reference to the invoking array.

Table 37-2. *The Member Functions of* ***valarray*** *(*continued*)*

Function	Description
valarray<T> &operator%=(const valarray<T> &*ob*) const;	The elements in the invoking array are divided by their corresponding elements in *ob* and the remainder is stored. Returns a reference to the invoking array.
valarray<T> &operator^=(const valarray<T> &*ob*) const;	The XOR operator is applied to corresponding elements in *ob* and the invoking array. Returns a reference to the invoking array.
valarray<T> &operator&=(const valarray<T> &*ob*) const;	The AND operator is applied to corresponding elements in *ob* and the invoking array. Returns a reference to the invoking array.
valarray<T> &operator|=(const valarray<T> &*ob*) const;	The OR operator is applied to corresponding elements in *ob* and the invoking array. Returns a reference to the invoking array.
valarray<T> &operator<<=(const valarray<T> &*ob*) const;	Elements in the invoking array are left-shifted by the number of places specified in the corresponding elements in *ob*. Returns a reference to the invoking array.
valarray<T> &operator>>=(const valarray<T> &*ob*) const;	Elements in invoking array are right-shifted by the number of places specified in the corresponding elements in *ob*. Returns a reference to the invoking array.

Table 37-2. *The Member Functions of* ***valarray*** *(continued)*

```
  interval[0] = 2; interval[1] = 3;

  cout << "Contents of v: ";
  for(i=0; i<12; i++)
    cout << v[i] << " ";
  cout << endl;

  result = v[gslice(0,len,interval)];

  cout << "Contents of result: ";
  for(i=0; i<result.size(); i++)
    cout << result[i] << " ";

  return 0;
}
```

The output is shown here:

```
Contents of v: 0 1 2 3 4 5 6 7 8 9 10 11
Contents of result: 0 3 6 2 5 8 4 7 10
```


The Helper Classes

The numeric classes rely upon these "helper" classes, which your program will never instantiate directly: **slice_array**, **gslice_array**, **indirect_array**, and **mask_array**.

The Numeric Algorithms

The header **<numeric>** defines four numeric algorithms that can be used to process the contents of containers. Each is examined here.

accumulate

The **accumulate( )** algorithm computes a summation of all of the elements within a specified range and returns the result. Its prototypes are shown here:

```
template <class InIter, class T> T accumulate(InIter start, InIter end, T v);
template <class InIter, class T, class BinFunc>
        T accumulate(InIter start, InIter end, T v, BinFunc func);
```

Here, **T** is the type of values being operated upon. The first version computes the sum of all elements in the range *start* to *end*. The second version applies *func* to the running

As you can see, it returns a pair object consisting of values of the types specified by *Ktype* and *Vtype*. The advantage of **make_pair( )** is that the types of the objects being stored are determined automatically by the compiler rather than being explicitly specified by you.

The **pair** class and the **make_pair( )** function require the header **<utility>**.

Localization

Standard C++ provides an extensive localization class library. These classes allow an application to set or obtain information about the geopolitical environment in which it is executing. Thus, it defines such things as the format of currency, time and date, and collation order. It also provides for character classification. The localization library uses the header **<locale>**. It operates through a series of classes that define facets (bits of information associated with a locale). All facets are derived from the class **facet**, which is a nested class inside the **locale** class.

Frankly, the localization library is extraordinarily large and complex. A description of its features is beyond the scope of this book. While most programmers will not make direct use of the localization library, if you are involved in the preparation of internationalized programs, you will want to explore its features.

Other Classes of Interest

Here are a few other classes defined by the Standard C++ library that may be of interest.

Class	Description
type_info	Used in conjunction with the **typeid** operator and fully described in Chapter 22. Uses the header **<typeinfo>**.
numeric_limts	Encapsulates various numeric limits. Uses the header **<limits>**.
raw_storage_iterator	Encapsulates allocation of uninitialized memory. Uses the header **<memory>**.

```
  StrType(char *str);
  StrType(const StrType &o); // copy constructor

  ~StrType() { delete [] p; }

  friend ostream &operator<<(ostream &stream, StrType &o);
  friend istream &operator>>(istream &stream, StrType &o);

  StrType operator=(StrType &o); // assign a StrType object
  StrType operator=(char *s); // assign a quoted string

  StrType operator+(StrType &o); // concatenate a StrType object
  StrType operator+(char *s); // concatenate a quoted string
  friend StrType operator+(char *s, StrType &o); /*  concatenate
                a quoted string with a StrType object */

  StrType operator-(StrType &o); // subtract a substring
  StrType operator-(char *s); // subtract a quoted substring

  // relational operations between StrType objects
  int operator==(StrType &o) { return !strcmp(p, o.p); }
  int operator!=(StrType &o) { return strcmp(p, o.p); }
  int operator<(StrType &o) { return strcmp(p, o.p) < 0; }
  int operator>(StrType &o) { return strcmp(p, o.p) > 0; }
  int operator<=(StrType &o) { return strcmp(p, o.p) <= 0; }
  int operator>=(StrType &o) { return strcmp(p, o.p) >= 0; }

  // operations between StrType objects and quoted strings
  int operator==(char *s) { return !strcmp(p, s); }
  int operator!=(char *s) { return strcmp(p, s); }
  int operator<(char *s) { return strcmp(p, s) < 0; }
  int operator>(char *s) { return strcmp(p, s) > 0; }
  int operator<=(char *s) { return strcmp(p, s) <= 0; }
  int operator>=(char *s) { return strcmp(p, s) >= 0; }

  int strsize() { return strlen(p); } // return size of string
  void makestr(char *s) { strcpy(s, p); } // make quoted string

  operator char *() { return p; } // conversion to char *
};
```

```
  return stream;
}

// Input a string.
istream &operator>>(istream &stream, StrType &o)
{
  char t[255]; // arbitrary size - change if necessary
  int len;

  stream.getline(t, 255);
  len = strlen(t) + 1;

  if(len > o.size) {
    delete [] o.p;
    try {
      o.p = new char[len];
    } catch (bad_alloc xa) {
      cout << "Allocation error\n";
      exit(1);
    }
    o.size = len;
  }
  strcpy(o.p, t);
  return stream;
}
```


As you can see, output is very simple. However, notice that the parameter **o** is passed by reference. Since **StrType** objects may be quite large, passing one by reference is more efficient than passing one by value. For this reason, all **StrType** parameters are passed by reference. (Any function you create that takes **StrType** parameters should probably do the same.)

Inputting a **string** proves to be a little more difficult than outputting one. First, the string is read using the **getline( )** function. The length of the largest string that can be input is limited to 254 plus the null terminator. As the comments indicate, you can change this if you like. Characters are read until a newline is encountered. Once the string has been read, if the size of the new string exceeds that of the one currently held by **o**, that memory is released and a larger amount is allocated. The new string is then copied into it.

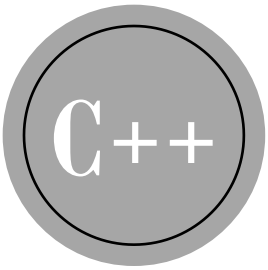
```
  char *s1;
  int i, j;

  s1 = p;
  for(i=0; *s1; i++) {
    if(*s1!=*substr) { // if not first letter of substring
      temp.p[i] = *s1; // then copy into temp
      s1++;
    }
    else {
      for(j=0; substr[j]==s1[j] && substr[j]; j++) ;
      if(!substr[j]) { // is substring, so remove it
        s1 += j;
        i--;
      }
      else { // is not substring, continue copying
        temp.p[i] = *s1;
        s1++;
      }
    }
  }
  temp.p[i] = '\0';
  return temp;
}
```


These functions work by copying the contents of the left-hand operand into **temp**, removing any occurrences of the substring specified by the right-hand operand during the process. The resulting **StrType** object is returned. Understand that neither operand is modified by the process.

The **StrType** class allows substring subtractions like these:

```
StrType x("I like C++"), y("like");
StrType z;

z = x - y; // z will contain "I C++"

z = x - "C++"; // z will contain "I like "

// multiple occurrences are removed
z = "ABCDABCD";
x = z -"A"; // x contains "BCDBCD"
```

```
    }
    o.size = len;
  }
  strcpy(o.p, t);
  return stream;
}

// Assign a StrType object to a StrType object.
StrType StrType::operator=(StrType &o)
{
  StrType temp(o.p);

  if(o.size > size) {
    delete [] p; // free old memory
    try {
      p = new char[o.size];
    } catch (bad_alloc xa) {
      cout << "Allocation error\n";
      exit(1);
    }
    size = o.size;
  }

  strcpy(p, o.p);
  strcpy(temp.p, o.p);

  return temp;
}

// Assign a quoted string to a StrType object.
StrType StrType::operator=(char *s)
{
  int len = strlen(s) + 1;
  if(size < len) {
    delete [] p;
    try {
      p = new char[len];
    } catch (bad_alloc xa) {
      cout << "Allocation error\n";
      exit(1);
    }
    size = len;
```



Chapter 40

An Object-Oriented Expression Parser

The Parser Class

The expression parser is built upon the **parser** class. The first version of **parser** is shown here. Subsequent versions of the parser build upon it.

```
class parser {
  char *exp_ptr;  // points to the expression
  char token[80]; // holds current token
  char tok_type;  // holds token's type

  void eval_exp2(double &result);
  void eval_exp3(double &result);
  void eval_exp4(double &result);
  void eval_exp5(double &result);
  void eval_exp6(double &result);
  void atom(double &result);
  void get_token();
  void serror(int error);
  int isdelim(char c);
public:
  parser();
  double eval_exp(char *exp);
};
```


The **parser** class contains three private member variables. The expression to be evaluated is contained in a null-terminated string pointed to by **exp_ptr**. Thus, the parser evaluates expressions that are contained in standard ASCII strings. For example, the following strings contain expressions that the parser can evaluate:

"10 – 5"

"2 * 3.3 / (3.1416 * 3.3)"

When the parser begins execution, **exp_ptr** must point to the first character in the expression string. As the parser executes, it works its way through the string until the null-terminator is encountered.

The meaning of the other two member variables, **token** and **tok_type**, are described in the next section.

The entry point to the parser is through **eval_exp( )**, which must be called with a pointer to the expression to be analyzed. The functions **eval_exp2( )** through **eval_exp6( )** along with **atom( )** form the recursive-descent parser. They implement an enhanced set of the expression production rules discussed earlier. In subsequent versions of the parser, a function called **eval_exp1( )** will also be added.

```
// Process an assignment.
void parser::eval_exp1(double &result)
{
  int slot;
  char ttok_type;
  char temp_token[80];

  if(tok_type==VARIABLE) {
    // save old token
    strcpy(temp_token, token);
    ttok_type = tok_type;

    // compute the index of the variable
    slot = toupper(*token) - 'A';

    get_token();
    if(*token != '=') {
      putback(); // return current token
      // restore old token - not assignment
      strcpy(token, temp_token);
      tok_type = ttok_type;
    }
    else {
      get_token(); // get next part of exp
      eval_exp2(result);
      vars[slot] = result;
      return;
    }
  }

  eval_exp2(result);
}

// Add or subtract two terms.
void parser::eval_exp2(double &result)
{
  register char op;
  double temp;

  eval_exp3(result);
  while((op = *token) == '+' || op == '-') {
    get_token();
```

```
  *temp = '\0';

  if(!*exp_ptr) return; // at end of expression

  while(isspace(*exp_ptr)) ++exp_ptr; // skip over white space

  if(strchr("+-*/%^=()", *exp_ptr)){
    tok_type = DELIMITER;
    // advance to next char
    *temp++ = *exp_ptr++;
  }
  else if(isalpha(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = VARIABLE;
  }
  else if(isdigit(*exp_ptr)) {
    while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
    tok_type = NUMBER;
  }

  *temp = '\0';
}

// Return true if c is a delimiter.
int parser::isdelim(char c)
{
  if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
    return 1;
  return 0;
}

// Return the value of a variable.
double parser::find_var(char *s)
{
  if(!isalpha(*s)){
    serror(1);
    return 0.0;
  }
  return vars[toupper(*token)-'A'];
}
```

```
// Multiply or divide two factors.
template <class PType> void parser<PType>::eval_exp3(PType &result)
{
  register char op;
  PType temp;

  eval_exp4(result);
  while((op = *token) == '*' || op == '/' || op == '%') {
    get_token();
    eval_exp4(temp);
    switch(op) {
      case '*':
        result = result * temp;
        break;
      case '/':
        result = result / temp;
        break;
      case '%':
        result = (int) result % (int) temp;
        break;
    }
  }
}

// Process an exponent
template <class PType> void parser<PType>::eval_exp4(PType &result)
{
  PType temp, ex;
  register int t;

  eval_exp5(result);
  if(*token== '^') {
    get_token();
    eval_exp4(temp);
    ex = result;
    if(temp==0.0) {
      result = (PType) 1;
      return;
    }
    for(t=(int)temp-1; t>0; --t) result = result * ex;
  }
```

Index