CHAPTER

9

An Introduction to C++

The C++ language was developed at Bell Laboratories in the early 1980s by Bjarne Stroustrup. The language is more than an extension of the powerful C language. It is an object-oriented programming language, whereas C is considered to be a procedural language. However, C is a subset of C++. That is, C programs can be compiled using a C++ compiler. The reverse is not true, meaning that C++ programs cannot be compiled using a pure C compiler.

C++ was developed so that object-oriented concepts such as *encapsulation*, *inheritance* and *polymorphism* could be used. We do not want to get into the details at this point. The best way to understand the capabilities of C++ is to examine a number of C++ programs.

This short chapter is not an exhaustive description of the C++ language. Instead, it introduces only the major features of C++, gives you the feel and look of the language, provides examples and demonstrates how to use the language to solve practical problems. To do all these things in just one chapter, we have deliberately avoided details and have not always followed standard practice. There is no ANSI standard for C++ currently although it exists for C. Despite these shortcomings, after reading this chapter, you will be well on your way to becoming an accomplished C++ programmer.

Some of the exercises in this chapter ask you to rewrite the programs written in C from the previous chapters. This approach allows you to concentrate on the C++ language itself instead of merely understanding the structure or algorithm of the program. In addition, it helps you understand how C++ can enhance and improve your C programs.

Now that you have come to this chapter, you must be adept at reading and understanding code. Therefore, we limit our introductory comments for each lesson and expect you to read all the notations in the code and the code itself. Read the explanations while referring back to the code, and do the exercises.

Lesson 9.1 C++ Comments and Basic Stream Input and Output

Topics

- Writing C++ comments
- Using streams for standard input and output

Let us now look at our first C++ program. This lesson's program shows you how to write C++ comments, receive input from the keyboard and display output on the screen. Read the notations and the code to see how comments and input/output (I/O) statements are written in C++.

Source Code

Output

```
                1. This is C++!
                2. This is C++!

                Greg is 21 years old and his units are 16.0

                Please type your name and the number of units you have:
Keyboard input  Linda 15.0
                Your name is Linda and your units are 15.0
```

Explanation

1. **How do we write C++ comments?** We begin C++ comments with a double-slash (//) token followed by a character string. Everything behind the token (unless the token is inside a string or a C comment) to the end of that line is the comment. For example,

```
// This is a C++ comment
```

is a C++ comment (see Fig. 9.1).

Note that C++ supports the C comment form, and you may enclose a C++ comment within a C comment. However, we recommend that you use C++ comments for C++ programs and C comments for C programs. Also notice that, if a comment is to extend over many lines, double slashes must be written at the beginning of each line.

C	C++
/*This is a C comment*/	//This is a C++ comment
#include <stdio.h>	#include <iostream.h>
.....	.....
printf("Please type your name");	cout << "Please type your name";
scanf("%s",name);	cin >> name;
.....	.....

Fig. 9.1 *A first look at C++*

2. **What is iostream.h and what are streams?** In C, the standard I/O routines are defined in the header file stdio.h. In C++, the equivalent header file is iostream.h. Streams are defined in this header file. In simple terms, streams can be thought of as memory cells connected to external devices such as the keyboard, screen or disk drive by the C++ I/O system. We can communicate with these devices by filling or reading the memory cells. In C++, this I/O concept allows us to read or fill the streams without regard to the devices connected to them. Therefore, whether it is communicating with a keyboard or a disk drive it is the same thing from the programmer's point of view.

   Both cout and cin are identifiers (defined in iostream.h) that, by default, refer to the standard output stream (connected to the screen) and standard input stream (connected to the keyboard), respectively. These streams are automatically opened when a C++ program is executed and therefore available for use in any programs that we write.

3. **How do we send output to the screen?** In C++, we can still use the C printf() function to display output on the screen. However, we can also use cout and the << operator directly to accomplish the task (see Fig. 9.2). The << operator is defined in C++ but not in C for use with I/O streams. It is called the *insertion operator.* This operator sends data from the right operand to where it is going (the cout stream). For instance, the statement

```
cout << "2. This is C++!\n\n";
```

   sends the string "2. This is C++!\n\n" to the cout stream; then it is automatically displayed on the screen.

   Note that you can use cout and the << operator to send any built-in data type, such as char, short, int, long, char* (string), float, double, long double or void*, to the output stream without using a format string. This system of I/O is smart enough to detect the differences among the various data types and display them correctly.

   You can use more than one << operator to output different types of data after cout. Between two << adjacent operators, however, you can insert only one item of data (expression or string). The operator associates from left to right. For example, the statement

```
cout << name << " is " << age << " years old and his units
                          are " << units;
```

   displays a character string (name), an int (age), a float (units) and the string constants on the screen. The output stream displays

```
Greg is 21 years old and his units are 16.0
```

   on the screen.

4. **How do we read input from the keyboard?** In C++, you can still use the C scanf() function to read input from the keyboard. However, you also can use cin and the >> operator directly to accomplish the task (see Fig. 9.2). The >> operator is defined in C++ but not in C for use with I/O streams. It is called the *extraction operator.* For example, the statement

```
cin >> name >> units;
```

sends the values (a string and a double data item) typed in at the keyboard automatically to the cin stream and thus to the variables *name* and *units*.

Note that you can use cin and the >> operator to send any built-in data type, such as char, short, int, long, char* (string), float, double, long double or void*, from the input stream without using a format string; this system of I/O is smart enough to detect the differences among the various data types and read them correctly. From the statement, notice that you can use more than one >> operator to input different types of data after cin.

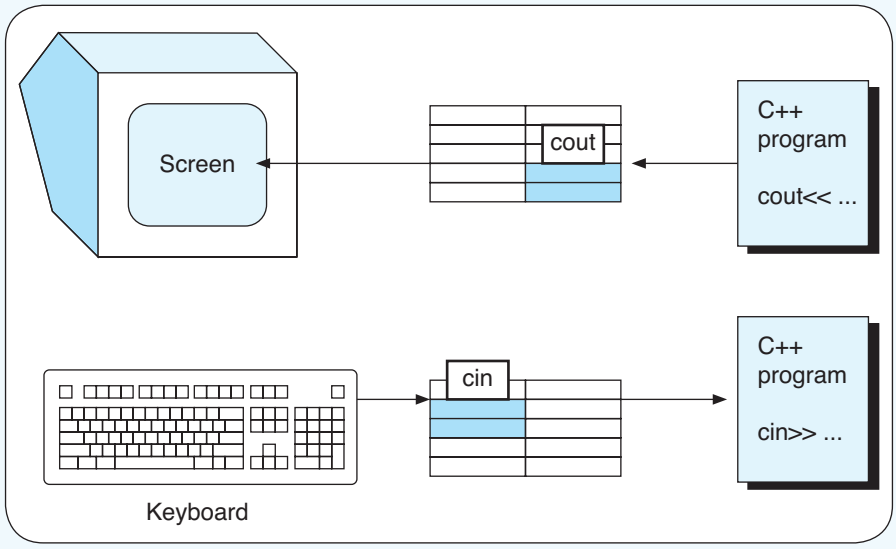


Fig. 9.2 ***Use of cout, cin, << and >> for input and output***

Concept Recap

1. C++ comments start with the double-slash (//) token. Everything behind the token to the end of that line is the comment.
2. In C++, the equivalent header file for stdio.h is iostream.h.
3. Output in C++ use cout and the << operator, for example,

```
cout << "Things to be out\n";
```

4. Input from the keyboard is processed in a similar way as that in cout.

```
cin >> var1 >> var2;
```

Exercises

1. Rewrite the program from Lesson 2.2 with C++ comments instead of C comments.

2. Use C++ I/O streams to read the following data from the keyboard:

```
char       'A'
int        123
long       987654
double     3.141592
char[20]   Welcome to C++!
```

and display them on the screen.

Lesson 9.2 Manipulators and Formatting Output

Topics

- Manipulators
- Basic iostream class

In the previous lesson, we used C++ I/O streams cin and cout with the << and >> operators to perform basic standard input and output. They are easy to use; however, the default format inherent in them may not be the one that we want. For example, suppose we want to display the value of π correct to only three decimal places. How can we do it? Review the code that follows. You will find that we can use cout to display data in any format as long as we add something to the statement. Two new terms are added, a manipulator and a class.

Source Code

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

To format output, we need iomanip.h.

```
void main(void)
 {
        int    ninety = 90;
        double pi = 3.141592654;

        cout << "Using manipulators to control format ----------\n\n";
        cout <<"Ninety in decimal  (default) is " <<        ninety << "\n";
```

Displaying ninety without manipulators.

```
cout <<"Ninety in octal is       " << oct << ninety << "\n";
cout <<"Ninety in hexadecimal is    "<< hex << ninety <<"\n\n";
```

We insert manipulators into the output stream to change the display.

endl creates a newline This is similar to "\n".

```
cout << "Using parameterised manipulators to control format -------\n\n";
cout << "1. PI="                          << pi << endl;
cout << "2. PI=" << setw(15)                 << pi << endl;
cout << "3. PI=" << setprecision(3)          << pi << endl;
cout << "4. PI=" << setw(20) << setfill ('*')<< pi << endl;
```

Parameterised manipulators with flags.

Class of flag. Flag.

```
cout << "5. PI=" <<setiosflags(ios::left) << setw(20)        << pi << endl;
cout << "6. PI=" <<setiosflags(ios::scientific | ios::showpos)<< pi << endl;
cout << "7. PI=" <<setiosflags(ios::fixed) << setprecision(5) << pi << endl;
```

We can have more than one flag using the | operator.

The scope resolution operator (::) is used between the class and the flag.

```
}
```

Output

```
Using manipulators to control format --------------------

Ninety in decimal (default) is 90
Ninety in octal is             132
Ninety in hexadecimal is       5a

Using parameterised manipulators to control format -------

1. PI=3.141593
2. PI=       3.141593
3. PI=3.142
4. PI=***************3.142
5. PI=3.142***************
6. PI=+3.142e+00
7. PI=+3.14159
```

Explanation

1. **What is a stream manipulator?** A stream manipulator is a special type of function/operator that can be used only with cout or cin and the << or >> operators. Stream manipulators may or may not have arguments. Manipulators that take no arguments do not need parentheses.

2. **How do we use stream manipulators?** We put them in a cin or cout statement adjacent to the << or >> operators. For instance, to display the decimal integer 90 in octal notation, we insert the manipulator oct into the cout statement as shown below:

```
cout <<"Ninety in octal is " << oct << ninety << "\n";
```

This statement uses the manipulator oct to convert the default format for the integer argument ninety from decimal to octal.

Other useful stream manipulators that do not require parameters to specify their actions are

- dec, which sets the conversion format to decimal base
- hex, which sets the conversion format to hexadecimal base
- endl, which inserts a newline and flushes the stream

For a complete list of manipulators that require no parameters, see the manual of your C++ compiler.

3. **How do we change the default field width set by cout?** We insert a *parameterised manipulator*, setw(), into the output stream to change the default field width. For example, the statement

```
cout << "2. PI=" << setw(15)<< pi << endl;
```

uses the manipulator setw(15) to change the default field width to 15. The parameter for setw() is of int type. The parameterised manipulators are declared in the header file iomanip.h. Include this file in your program if you want to use parameterised manipulators.

Other useful stream parameterised manipulators are these:

Manipulator	Action	Example
setfill (int f)	Set the fill character to f	setfill ("*")
setprecision(int p)	Set the precision of a floating point number to p	setprecision(3)
setw(int w)	Set the field width to w	setw(20)
setiosflags(long f)	Set the format flag to f	setiosflags(ios::left)

In the following list of examples, left is considered to be a flag. Flags used in this lesson are

Flag type	Usage
left	Left-adjust output
fixed	Use fixed decimal point notation for a floating point number
scientific	Use scientific notation for a floating point number
showpos	Show + sign for a positive number

To use a parameterised manipulator with more than one flag, you can repeatedly use the manipulator with one flag at a time. You can also combine the flags with the | operator. For example, the manipulator setiosflags(), which follows,

```
cout << "6. PI=" <<setiosflags(ios::scientific |
          ios::showpos)<< pi << endl;
```

uses the | operator to combine the scientific and showpos flags.

For a complete list of parameterised manipulators, see the manual of your C++ compiler.

4. **What is the notation ios::scientific?** C++ streams are defined in the stream library and determined by their *class* and by customised insertion and extraction operators. We discuss C++ classes in more detail later in this chapter; for now, just remember that a C++ class consists of *data* and *functions* that manipulate that data. A C++ class is similar to a C structure except that it may contain both data and functions as its members. The notation contains a C++ I/O stream class, ios. The format flag, such as scientific, left or showpos, is an enumerator (meaning that it represents an int value) specified in the class ios. To use these flags, we first write the class name, ios, followed by the *scope resolution operator* (::), and end with the flag name. The operator indicates that the flag is the member of the class name that precedes it.

Concept Recap

1. A stream *manipulator* is a special type of function/operator that is used in input/output operations, for example, oct, dec, endl, etc. They normally perform some additional operations on the data stream, for example, number base conversion.

2. Parameterised manipulators can perform even more complicated functions such as setting the print width of the output field.

3. The *scope resolution operator* (::) is used to address a particular value or member of a class inside C++. The concept is similar to that of the structure member operator. The details of C++ class will be elaborated upon later.

Exercises

1. Use C++ manipulators with no parameters to rewrite the program from Lesson 3.1.

2. Use C++ parameterised manipulators to rewrite the program from Lesson 3.2.

3. For each data type, use C++ I/O streams to read the following data from the keyboard (note: you need to use resetiosflags(long f) to clear the previous flag specified by f. We have not covered resetiosflags(long f) in this text, so you must experiment with it):

```
char            'A'
int             123
long            987654
double          3.141592
char[20]        Welcome to C++!
```

Display the data on the screen as follows:

```
12345678901234567890123456789012345678901234567890
Char        A                 A*********
Int         123               +123******
Long        987654            987654****
Double      3.141592          3.14e+00**
char[20]    Welcome to C++!   **********Welcome to C++!*****
```

Lesson 9.3 Function Overloading

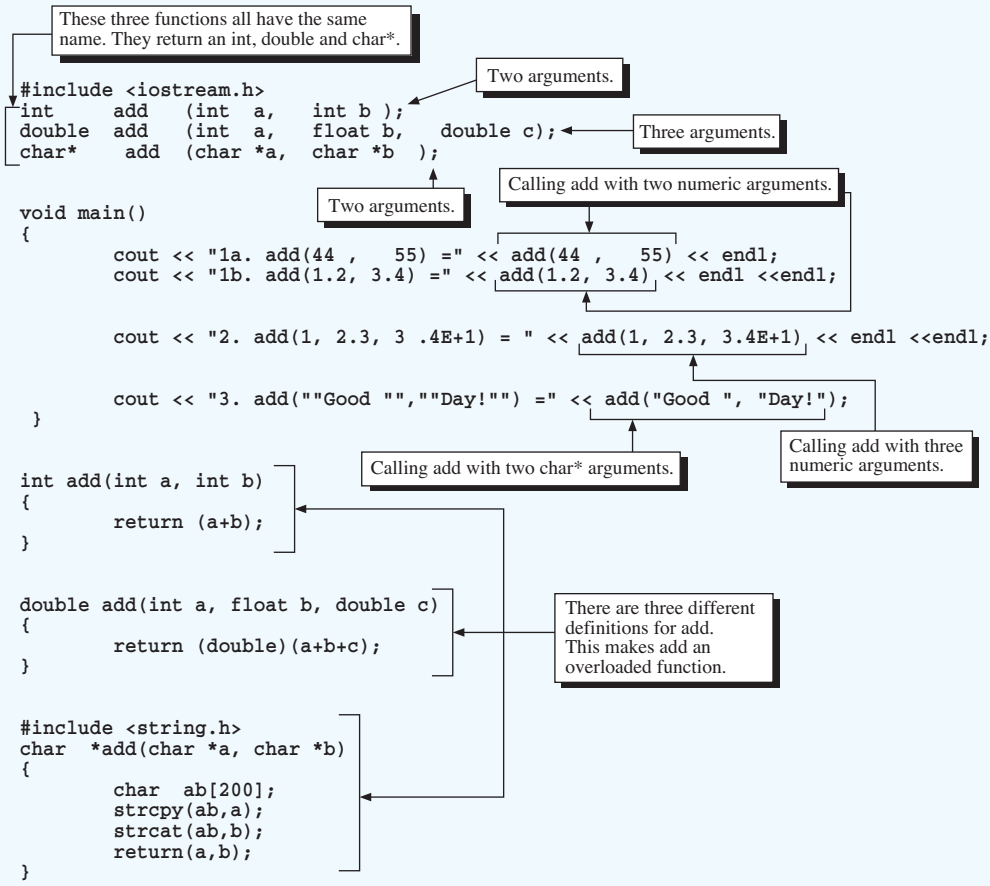
Topic

- Overloaded functions

Suppose you want to write a function to add two numbers and another function to concatenate (connect) one string to another. In C, you must give them two different names, say *add_num* and *add_string* even though both combine two variables into one. In C++, however, you can give them the same name, say *add*. This means that you may "overload" two different functions and create a common name for them. By doing so, you may generalise or standardise

functions that perform similar tasks. This will make your programs easier to manage and to understand. You may overload as many functions as you like. Function overloading is a useful feature of C++. In the program that follows, observe how functions are overloaded.

Source Code




```
#include <iostream.h>
int     add   (int  a,   int b );
double  add   (int  a,   float b,   double c);
char*    add  (char *a,  char *b  );

void main()
{
        cout << "1a. add(44 ,   55) =" << add(44 ,   55) << endl;
        cout << "1b. add(1.2, 3.4) =" << add(1.2, 3.4) << endl <<endl;

        cout << "2. add(1, 2.3, 3 .4E+1) = " << add(1, 2.3, 3.4E+1) << endl <<endl;

        cout << "3. add(""Good "","Day!"") =" << add("Good ", "Day!");
 }

int add(int a, int b)
{
        return (a+b);
}

double add(int a, float b, double c)
{
        return (double)(a+b+c);
}

#include <string.h>
char  *add(char *a, char *b)
{
        char  ab[200];
        strcpy(ab,a);
        strcat(ab,b);
        return(a,b);
}
```

Output

```
1a. add(44,   55) =99
1b. add(1.2, 3.4) =4

2. add(1, 2.3, 3.4E+1) = 37.3

3. add("Good", "Day!") =Good Day!
```

Explanation

1. **What is function overloading?** Function overloading is a C++ programming technique of supplying more than one definition for a given function name. This C++ feature allows you to develop functions that perform similar tasks, using the same name. The C++ compiler is left to decide which function should be used. For example, in this program, we have three functions, the first adds two int numbers; the second adds an int, a float and a double; and the third concatenates two strings. All three functions have the same name, *add*. Therefore, the add function is overloaded (see Fig. 9.3).

2. **How do we call an overloaded function?** To call an *overloaded function*, we first determine which of the overloaded functions we need. Then we place the appropriate number and types of arguments in the function call. For example, if we want to add two numbers, then we place two numeric arguments, such as

```
add(44,   55)
```

or

```
add(1.2, 3.4)
```

to make the function call. If we want to concatenate two strings, then we place two string arguments, such as

```
add("Good ", "Day!")
```

to make the function call. The number and types of arguments we place must match the arguments in one of the function definitions. If none of them matches or there is more than one match, we will receive an error message from the compiler.

3. **What is the main restriction on writing overloaded functions?** Not all functions can be overloaded. There are a number of rules and restrictions on developing overloaded functions. The main restriction you need to know at this point is that the function to be overloaded must have different argument lists, that is, the number or type of any two functions in the set must be different. Without this restriction, the C++ compiler will not be able to distinguish which function should be chosen to perform the work. For example, the prototypes of the overloaded functions in this lesson are

```
int       add (int   a,    int b               );
double    add (int   a,    float b,    double c);
char     *add (char *a,    char *b             );
```

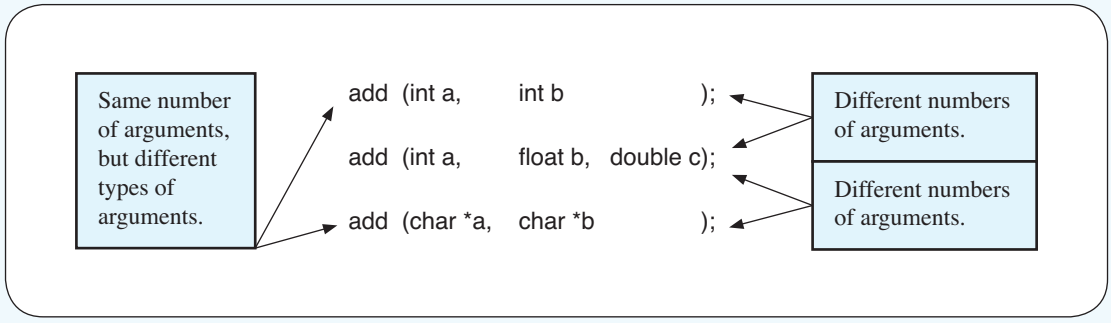


Fig. 9.3 ***An overloaded function***

Obviously, there is no ambiguity in the arguments between any two functions in the set. However, if we change the prototype of the second function to

```
double  add (float b, double c);
```

then there is an ambiguity in the arguments between the first and the second functions, even though they have different return values and different argument types. For example, if you use the statement

```
cout << add(1.2, 3.4);
```

then the C++ compiler might convert the floating number arguments to int and make a function call as

```
cout << add(1, 3);
```

At this point, the C++ compiler cannot select which function to use to add the two numbers and hence generates an error message. In general, when you develop a set of overloaded functions, make sure that there is no ambiguity in the arguments for different functions.

Concept Recap

1. Function overloading allows you to develop a set of functions that perform similar tasks, using the same name. Hence, the *number* and *types* of arguments in the function call must be exactly the same as that function that you want to call.

Exercises

1. Use overloaded functions to write a program that can find
   a. The maximum of three numbers.
   b. The longest and last alphabetical word in a character string.

Use the following data as your input:

```
1 23 4.56
Have a nice day !
100 3.14 999.9
```

The program should show the following output:

```
The maximum of 1, 23 and 4.56 is 23
The longest and last alphabetical word in "Have a nice day
!" is "nice"
The maximum of 100, 3.14 and 999.9 is 999.9
```

2. Use overloaded functions to write a program that can sort

   a. The elements of an int array of four in ascending order.
   b. The elements of a float array of five in descending order.

   Use the following data as your input:

```
33 22 11 55
8.8 6.6 9.9 7.7 5.5
```

   The program should show the following output:

```
11 22 33 55
9.9 8.8 7.7 6.6 5.5
```

Lesson 9.4 Default Function Arguments

Topic

- Functions with default arguments

At times you may need to call a function with the same arguments repeatedly in your program. In doing so, by mistake, you may enter a wrong argument. To avoid this error, C++ provides a solution that allows you to call the function without using any arguments. The prototype of the function in the program for this lesson has three arguments, but you may call them with three, two, one or even no arguments at all. Review the program and find out how to call a function without the required arguments.

Source Code

Function add has three default arguments.

Function subtract has two default arguments out of three arguments.

Calling a function with fewer than the number of arguments in the prototype causes the values of the leftmost argument(s) to be passed. The default values are used for the other arguments.

Function add returns a numeric sum of the first two arguments.

Function subtract subtracts three integers.

```
#include <iostream.h>
double  add (int aa=10,   double bb=20.0,   char *cc=" aa plus bb");
int     subtract (int xx,   int yy=11,   int zz=22);

// Wrong --- int subtract(int xx=11, int yy, int zz);

void main(void)
{
        cout << "1. add(  ) = "          << add(  );
        cout << "2. add(30  ) = "        << add(30 );
        cout << "3. add(40, 50  ) = "    << add(40, 50 );
        cout << "4. add(60, 70.0," "Result"")= " << add(60, 70.0,"Result")<<endl;

        cout << "5. subtract(77    ) = " << subtract(77 );
        cout << "6. subtract(99, 44) = " << subtract(99, 44) ;
}

double add(int a, double b, char *c)
{
        cout << endl << endl;
        cout << "aa = " << a << ",        ";
        cout << "bb = " << b << ",        ";
        cout << "cc = " << c << endl;

        return (a+b);
}

int subtract(int x, int y, int z)
{
        cout << "\n\n";
        cout << "xx = " << x << ",        ";
        cout << "yy = " << y << ",        ";
        cout << "zz = " << z << endl;

        return (x-y-z);
}
```

Output

```
aa = 10,      bb = 20,     cc =  aa plus bb
1. add(   ) = 30

aa = 30,      bb = 20,     cc =  aa plus bb
2. add(30 ) = 50

aa = 40,      bb = 50,     cc =  aa plus bb
3. add(40, 50  ) = 90

aa = 60,      bb = 70,     cc = Result
4. add(60, 70.0,Result)= 130
```

```
xx = 77,      yy = 11,     zz = 22
5. subtract(77 ) = 44

xx = 99,      yy = 44,     zz = 22
6. subtract(99, 44) = 33
```

Explanation

1. **How do we call a function without the required argument(s)?** To call a function without the required argument(s), we need to write a function that has default argument value(s). A *default argument* is initialised with a default value in the prototype of the function. For example, the prototype of the function

```
double  add (int aa=10,  double bb=20.0, char *cc=" aa
                        plus bb");
```

has three formal arguments: aa, bb and cc. Each argument is initialised with a default argument value. The C++ compiler uses the default values if the actual argument values are not provided when the function is called. However, the default value is overridden if the actual argument value is provided in the call.

The preceding function has three default argument values; therefore, we may call it with no arguments or with one, two or three arguments. Note that if a function has more than one default argument, when we call the function, we cannot arbitrarily input some actual arguments as non default argument values and assume the compiler will assign default argument values for the remaining arguments. The rule is that, if we want to replace a default argument value with a user-defined value, then we must replace all other default values on its left with user-defined values. For example, in the function add, if we want to override the second default value (for argument bb), then we must use non-default values for the first and second arguments and use the default value for the third argument. The following table shows the default values used when we call the function add with different numbers of arguments:

Function call	Default arguments
`add()`	`aa, bb, cc`
`add(30)`	`bb, cc`
`add(40, 50)`	`cc`
`add(60, 70.0,"Result")`	`None`

2. **Do we need to initialise default values for all formal arguments?** No, we may select some arguments as default arguments and the rest as normal arguments. However, this cannot be done randomly. The rule is that if we want to use the *n*th argument as a default argument, then all arguments after the *n*th argument must also be default arguments. For example, in the function subtract, we selected the second argument, yy, as a default argument, then the third was also required to be a default argument (see Fig. 9.4). If we had defined the function subtract as

```
int subtract(int xx=11, int yy, int zz);
```

then the C++ compiler would have generated an error message on compilation because the first argument is a default argument but the second and the third are non-default arguments.

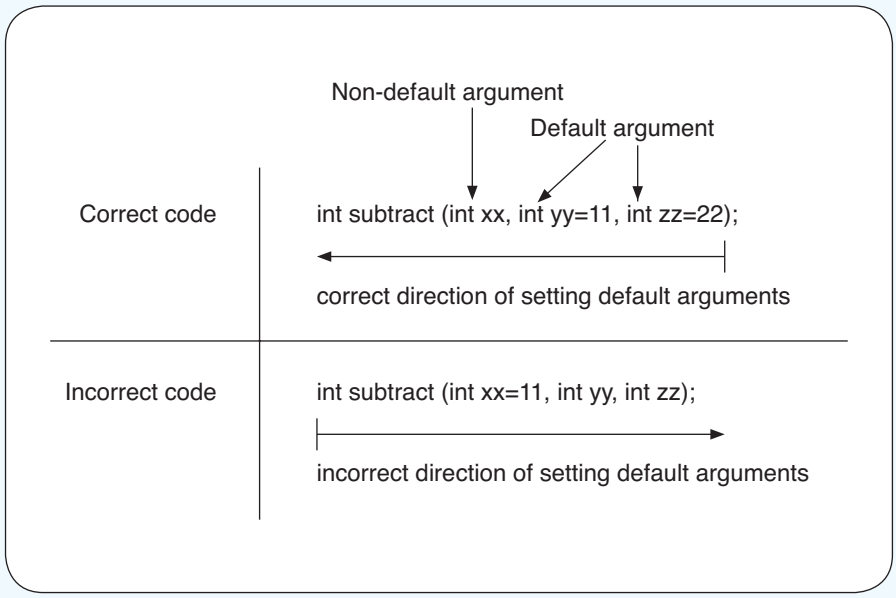


Fig. 9.4 ***Default function arguments***

Concept Recap

1. In a C++ function, a *default argument* is initialised with a default value in the prototype of the function. For example,

```
double  add (int aa=10,  double bb=20.0, char *cc=" aa
                        plus bb");
```

Exercises

1. Use default arguments to write a program that

   a. Creates the following default output:

   ```
   ABCD CORPORATION
   Project _________   Contract No. 3815-A   FileNo._________
   Designed: JKR       Checked ____________    Date _____/2011
   ```

   b. Gets input from the user on Project, File No., Checked and Date data.
   c. Generates a non-default output as follows:

   ABCD CORPORATION
   Project USA-OIL-1 Contract No. 3815-A File No. OIL-A12345
   Designed: JKR Checked John & Ken Date 12/13/2011

2. Use default arguments to write a program that

   a. Gets an input data filename from the user. If the user only presses the Return key, then the default name of "INPUT.DAT" is used.
   b. Reads the data in the file. The file always has four columns of data representing the point number, X, Y and Z coordinates. The number of rows may vary. Input the following data:

   ```
   No.      X          Y          Z
    1     755.0      221.9      696.4
    2     744.4      204.3      698.6
    3     743.1      206.8      689.9
    4     734.8      225.4      701.3
   ```

   c. Asks the user to input two point numbers. For example, if the user enters 34, then calculate the distance between points 3 and 4. However, if the user only presses the Return key, calculate the distances between points 1 and 2, 2 and 3, 3 and 4 and 4 and 1.
   d. Generates a neat screen output.

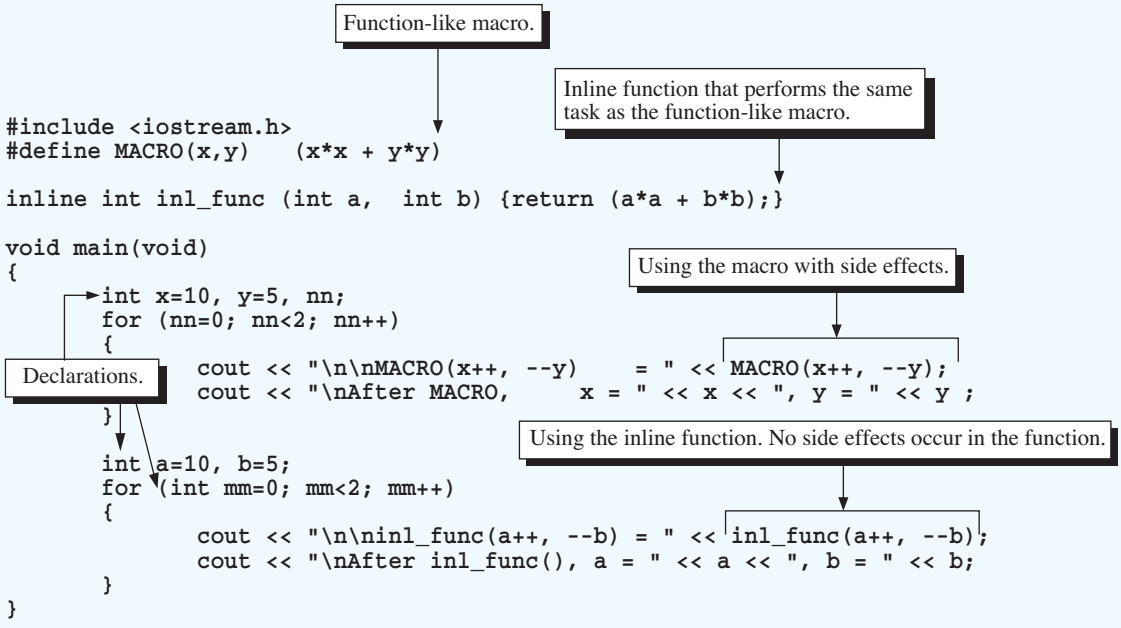
Lesson 9.5 Inline Functions and Position of Variable Declarations

Topics

- Inline functions
- Position of variable declarations

This lesson examines two enhancements of C++ to the C language. One is the use of inline functions versus macros and the other is the flexibility of placing variable declarations. Look at the inline function and the macro in the program that follows. Can you guess the advantages and disadvantages inline functions may have over function-like macros? Can you find instances in the program where variables are declared other than at the beginning of the code?

Source Code




```
#include <iostream.h>
#define MACRO(x,y)   (x*x + y*y)

inline int inl_func (int a,  int b) {return (a*a + b*b);}

void main(void)
{
      int x=10, y=5, nn;
      for (nn=0; nn<2; nn++)
      {
            cout << "\n\nMACRO(x++, --y)     = " << MACRO(x++, --y);
            cout << "\nAfter MACRO,      x = " << x << ", y = " << y ;
      }
      int a=10, b=5;
      for (int mm=0; mm<2; mm++)
      {
            cout << "\n\ninl_func(a++, --b) = " << inl_func(a++, --b);
            cout << "\nAfter inl_func(), a = " << a << ", b = " << b;
      }
}
```

Output

```
MACRO(x++, --y)     = 122
After MACRO,      x = 12, y = 3

MACRO(x++, --y)     = 158
After MACRO,      x = 14, y = 1

inl_func(a++, --b) = 116
After inl_func(), a = 11, b = 4

inl_func(a++, --b) = 130
After inl_func(), a = 12, b = 3
```

Explanation

1. **What is an inline function?** An inline function is like a regular function except that it uses the keyword *inline* as the function *qualifier*, meaning that the first word in the function prototype is *inline* and the definition of the function is followed directly by its function body (see Fig. 9.5). For example, the statement

```
inline int inl_func (int a, int b) {return (a*a + b*b);}
```

declares that inl_func is an inline function; it is of int type, has two arguments, a and b, and returns the value of (a*a + b*b) to the calling function.

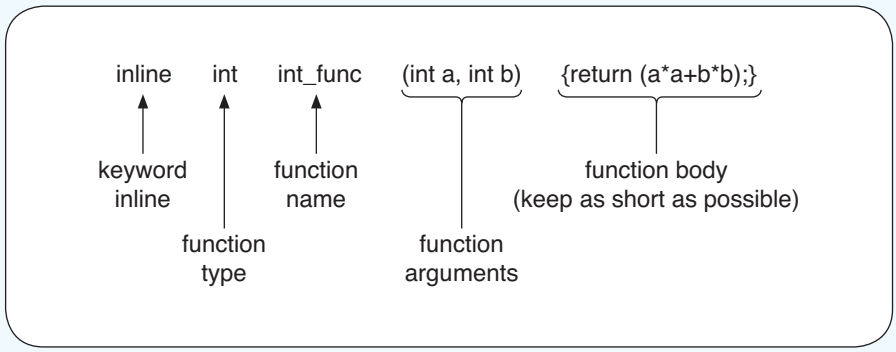


Fig. 9.5 *An Inline function*

2. **What effects will inline functions and macros have on your program?** For inline functions, the *inline* keyword causes the C++ compiler to insert a complete copy of the function at each place it is called. This eliminates loading function arguments each time it is called. Therefore, programs using inline functions run faster than those using ordinary functions for the same purpose. However, if an inline function is called many times, the multiple copies of the function will make a program larger. In general, an inline function is used only when it is short and called in only a few places.

   Macros are similar to inline functions. The symbolic name declared with the #define directive is replaced with the text each time the macro name appears in a program. Similar to an inline function, a macro adds more code to a program and also makes a program run faster.

   Inline functions are recognised by C++ compilers. The compiler type checks the function argument and its return type. Macros, on the other hand, are processed by preprocessor. The preprocessor simply replaces

the macro name with its replacement text and no type checking is performed.

Inline functions are easier to write than macros. You may use default arguments in inline functions but not in macros. In addition, inline functions behave like ordinary functions, without the side effects of macros. For example, with the initial input values of $x = 10$ and $y = 5$, the result of invoking

```
MACRO(x++, --y)
```

is 122, which is the value of $(10 * 11 + 4 \times 3 = 122)$. Each time MACRO is called, the value of x is incremented once (from 10 to 11) and the value of y is decremented twice (from 5 to 3). After the call, the value of x is incremented once more (from 11 to 12).

However, with the initial input values of $a = 10$ and $b = 5$, the result of calling

```
inl_func(a++, --b)
```

is 116, which is the value of $(10 \times 10 + 4 \times 4 = 116)$. Each time inl_func() is called, the value of a is not changed (10), and the value of b is decremented once (from 5 to 4). After the call, the value of a is incremented once (from 10 to 11). With multiple calls of MACRO and inl_func(), the differences produced by the side effects can be very large.

3. **Can we declare variables at any location in the program?** Yes, as long as the locations are meaningful and we place the declarations before we use the variables. We may place declarations at the beginning of our code, such as these declarations of variables x and y:

```
int x=10, y=5, nn;
```

Or we may put them somewhere in the code, such as these declarations of variables a and b:

```
int a=10, b=5;
```

yet another way is to place them within the block of the code, such as this declaration of variable mm inside the for loop,

```
for (int mm=0; mm<2; mm++)
```

Placing variable declarations before they are used may make our programs more readable. However, if we randomly place the declarations all over our code, the flexibility we gain from C++ may cause us and others who read our programs difficulties in finding where the variables are declared.

Concept Recap

1. In inline functions, a complete copy of the function is inserted at each place it is called. Thus, inline functions run faster than ordinary ones.
2. C++ allows variables to be declared anywhere before they are used.

Exercises

1. Write a program to read the following data

Radius at the bottom of a cone	Height of cone
10.1	66.6
20.2	55.5
30.3	44.4

Next, use an inline function and a macro to calculate the volume of each cone and display the output on the screen.

2. Use the data in Exercise 1 and calculate the surface area of each cone (including the bottom area). You are required to use a master inline function and a master Macro to perform the calculation. The master macro may invoke as many other macros as you need.

Lesson 9.6 C++ Classes and Objects With Data Members Only

Topics

- C++ classes
- Object-oriented programming

What we have covered up to this point are simply enhancements to the C language, not really the essence of C++. Beginning with this lesson, we deal with the core of the C++ language, classes and objects. Once you understand what classes and objects are, you will begin to appreciate their value.

This lesson's program assigns values to members of a structure and a class. It outputs the values of the members of the structure and class to the screen.

Source Code

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


struct Bus
{
        char  colour[10];
        float price;
};

class Car
{
        public:
        char  colour[10];
        private:
        float price;
};

void main(void)
{

        Bus newbus, oldbus;

        strcpy(newbus.colour, "Red");
        cout << "newbus.colour = " << newbus.colour << endl;
        newbus.price = 1234.5;
        cout << "newbus.price = " << newbus.price << "\n\n";

        Car newcar, oldcar;

        strcpy(newcar.colour, "Blue");
        cout << "newcar.colour = " << newcar.colour << endl;

        //newcar.price = 1234.5;
        //cout << "newcar.price = " << newcar.price << "\n\n";
}
```

Definition of struct Bus.

We can designate members of classes as either public or private. Members with the private designation cannot be accessed by a function outside of the class.

Definition of class Car.

Initialising and printing members of variables of type struct Bus.

Declaring newcar and oldcar to be objects of class Car.

We cannot access newcar.price from main because it is a private member.

Output

```
newbus.colour = Red
newbus.price = 1234.5

newcar.colour = Blue
```

Explanation

1. **What are classes and objects?** Classes in C++ are user-defined data types similar to the C user-defined data types, structures. But classes are more powerful than structures. Structures in C can contain data members only whereas classes in C++ may contain both data and function members. The function members manipulate the data members. In this lesson, we discuss classes with data members only.

   An instance of a class is called an *object*. An object is an entity that contains both data and functions. When we use classes and objects in our programs, our programs are object oriented; we are writing object-oriented programs.

2. **How do we define classes and declare objects in a program?** Defining a class in C++ is very similar to defining a structure in C. Compare these two:

```
struct Bus
{
   char  colour[10];
   float price;
};

class Car
{
   public:
   char  colour[10];
   private:
   float price;
};
```

   The primary differences between them are the keywords *public* and *private*. Both contain the data member colour[10] of type char and price of type float. Once we have defined the Bus structure, we can declare variables belonging to that structure. Similarly, once we have defined the Car class, we can declare objects belonging to that class. For example, the statements

```
Bus newbus, oldbus;
Car newcar, oldcar;
```

   declare the *variables* newbus and oldbus to belong to the Bus structure, and the *objects* newcar and oldcar as belonging to the Car class.

   The Bus structure has two data members. Therefore, any variable belonging to this type also has two data members. Similarly, the Car class has two data members. Therefore, any object belonging to this class also has

two data members. With proper treatment, any data member in a structure or class can be handled like any similar data type in C++.

Data members in classes are sorted into two groups (see Fig. 9.6): public and private (another group, named *protected*, will not be discussed here). The private or public labels in the class definition specify the availability of each data member in the class. You may have as many data members as you need in a class. Any data member declaration that appears after a specified label belongs to the specified group. You may place the public and private labels in any order or use as many as you need, but we recommend that you arrange them into two groups. Typically, we use the private label for data members and the public label for function members (not shown in this program). Public data members have higher availability than the private ones, meaning that there are more ways and fewer restrictions to access public data members than private ones (see explanation 3 below).

In C++, you may also define a class with the keyword *struct* or *union*. This means that C++ structures and unions offer more flexibility than their C counterparts because they may contain both data and functions as members. The main difference between class, struct and union, as shown in the following table, is the accessibility of their members.

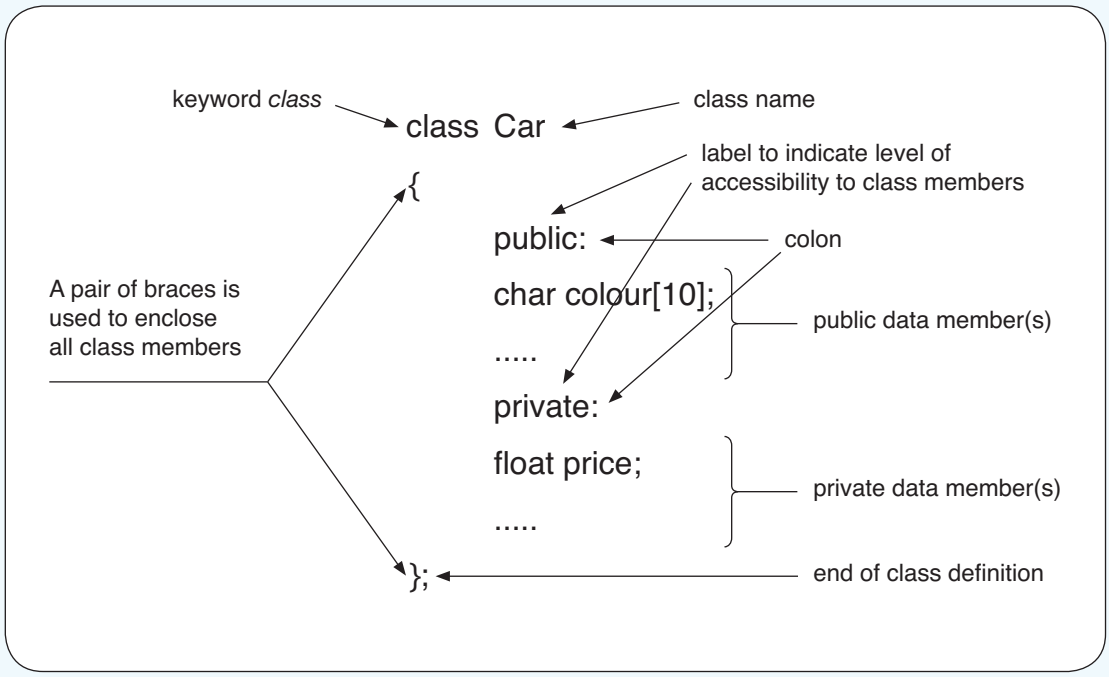


Fig. 9.6 ***A C++ class definition***

Class type	Member accessibility by default	Modification of accessibility
class	Private	Can be changed by user
struct	Public	Can be changed by user
union	Public	Cannot be changed by user

Note that a class can be defined within or outside a function. The class is local if it is defined within the function; otherwise, it is global. The Bus structure and Car class in this program are global.

The default accessibility for a class is private. If no accessibility is specified for any members, C++ makes the accessibility private.

3. **How do we access public data members of a class object?** We access public data members of a class object in the same way as we did data members of a structure variable. For example, to access the data members, colour and price, of structure variable newbus, we first use the statements

```
strcpy(newbus.colour, "Red");
newbus.price = 1234.5;
```

to initialise them. After that, the data members can be manipulated like any other regular data. The same approach can be used to access public class data members. For example, we may use the statements

```
strcpy(newcar.colour, "Blue");
cout << "newcar.colour = " << newcar.colour << endl;
```

to initialise and then display the public data member, colour of object newcar, on the screen. This example shows that public data members can be accessed by any function (user-defined or standard library function) in the program.

4. **How do we access private data members?** The data member, price, in class Car is private. A private data member cannot be accessed by a function outside of the class. For example, the statements

```
//newcar.price = 1234.5;
//cout << "newcar.price = " << newcar.price << "\n\n";
```

are invalid because newcar.price is private. It cannot be initialised by an assignment statement and cannot be displayed by cout in the function main because main is not a member of the Car class. Private data members can be accessed only by member functions belonging to the same class. We discuss how to create member functions in the next lesson.

Concept Recap

1. Classes in C++ contain both data and function members.
2. An instance of a class is called an *object*.
3. We use the member operator to access public data members of a class object. We use a dot operator to connect the object and its member function:

```
Object.member
```

4. A private data member cannot be accessed by a function outside of the class.

Exercises

1. In a lab class, you are asked to measure oxygen consumption. You will measure the change in gas volume during respiration in respirometers containing either soaked or dry peas at a given temperature. The following table shows the results of the measurements:

Temperature (C)	Time *x* (min)	Dry peas (reading at time *x*)	Soaked peas (reading at time *x*)
23	Initial, 0	0.89	0.65
23	0–5	0.86	0.39
23	0–10	0.85	0.18
23	0–15	0.84	0.02
23	0–20	0.83	0.00

Write a program to display the measurements. The program shall

a. Read the following original measurement data file,

Temp	Time	Dry peas	Soaked peas
23	0	0.89	0.65
23	5	0.86	0.39
23	10	0.85	0.18
23	15	0.84	0.02
23	20	0.83	0.00

using a structure that contains the following data members:

```
int temp;
int time[10];
```

```
double dry[10];
double soaked[10];
```

b. Repeat step a but use a class instead of a structure.

2. Light can be used to measure the change of contamination content in a lake. For example, light may reach a depth of only a few feet in a very muddy lake but a greater depth in a clearer one. The following table shows the percent of incident light at two different lakes:

Incident light (%)	Depth in Lake1 (ft)	Depth in Lake2 (ft)
100	0.0	0.0
65	1.5	2.4
25	2.3	12.0
10	7.4	25.3
2	12.0	30.2

Write a program to

a. Read the data using a class.
b. Convert the unit of depth from feet to metres.
c. Determine which lake is clearer.
d. Display the information in steps (b) and (c) on the screen.

Lesson 9.7 Classes with Data and Function Members, Encapsulation

Topics

- Member functions
- Encapsulation

The class we discussed in the previous lesson contained only data members. This type of class is similar to a C structure when its data members are public. To manipulate data in a traditional C structure (e.g. displaying the data on the screen), it is necessary to use a function. The data and the function for manipulating the data are different entities and not connected explicitly. In other words, we may use any function to handle the data or use the function to handle any data.

In C++, however, data members and function members are linked together and placed under a class object. Linking the data and the functions that manipulate the data into a single object is called *encapsulation*. Because classes link functions and data, we work with classes in a manner different from the way that we work with structures. This lesson illustrates how we call functions that are linked with data through classes and objects. It creates three objects: newcar, oldcar and mycar. The objects are declared to be of class Car. We assign values to the data members and print. We call the function members to assign values to some of the data members.

Source Code

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

class Car
{
        char  owner[11];
        public:
        char  colour[10];
        int   year_made;
        void  get_info(char *who, int year, double cost);
        void  display(void);

        private:
        double price;
        double   sellcar (double sell_price);
};

void main(void)
{
        Car newcar, oldcar, mycar;

        strcpy (newcar.colour, "Blue");
        cout << "newcar.colour     = " << newcar.colour << endl;
        newcar.get_info("Mary", 1998, 6543.2);
        newcar.display();

        strcpy (oldcar.colour, "White");
        cout << "oldcar.colour     = " << oldcar.colour << endl;
        oldcar.get_info("John", 1921, 1234.5);
        oldcar.display();

        oldcar.year_made=1934;
        mycar=oldcar;
        cout << "mycar.colour      = " << mycar.colour << endl;
        mycar.display();
}
```

Because this is neither a public nor a private function it is given the default specification (private).

Public members of class Car.

Public functions can be called from any function.

Private members of class Car. These can be accessed only from the member functions get_info and display.

Declaring the objects newcar, oldcar and mycar of class Car.

In main, we can access only the public data members colour and year_made.

We can also access the public functions get_info and display. To do so though, we must associate the call with a declared object. In this case, the object is oldcar.

We can copy one object into another using a single assignment statement.

```
void Car::get_info(char *who, int year, double cost)
{
        strcpy(owner, who);
        year_made = year;
        price = cost;
}

double Car::sellcar(double sell_price)
{
        if (sell_price < 5000) return (sell_price + 265.5 );
        else
        return (sell_price + 456.8 );
}
void Car::display(void)
{
        cout << "owner              = " << owner      << endl;
        cout << "year_made          = " << year_made  << endl ;
        cout << "price              = " << price      << endl;

        cout << "sellcar(price)     = " << sellcar(price) << "\n\n";
}
```

In the function header, we must indicate that get_info, sellcar and display are member functions of the Car class. The reason for this is that in C++ we are allowed to have functions with the same name but of different classes.

Because sellcar is a private member function, we can call it only from other member functions. Here, we call it from Car :: display.

Within a member function, we need not associate function calls or members with a specific object. This is because in calling the function we have already indicated an object. Notice the call oldcar.display( ) in main, which already indicates the object oldcar on calling the function.

Output

```
newcar.colour     = Blue
owner             = Mary
year_made         = 1998
price             = 6543.2
sellcar(price)    = 7000

oldcar.colour     = White
owner             = John
year_made         = 1921
price             = 1234.5
sellcar(price)    = 1500

mycar.colour      = White
owner             = John
year_made         = 1934
price             = 1234.5
sellcar(price)    = 1500
```

Explanation

1. **What are class member functions?** Class member functions are important components of C++ classes. Member functions are defined within the definition of the class to which they belong. For example, the class definition

```
class Car
{
    char  owner[++];

    public:
    char  colour[10];
    int   year_made;
    void  get_info(char *who, int year, double cost);
    void  display(void);

    private:
    double price;
    double   sellcar (double sell_price);
};
```

defines three member functions, get_info(), display and sellcar();. The prototype of a class member function is the same as the prototype of any other C or C++ regular function. A member function can have any number and any valid types of formal arguments and can return any types of data to its calling function. For example, the get_info() member function is of void type and has three formal arguments: who, year and cost. The sellcar() member function is of double type and contains only one formal argument, sell_price. The display() function is of void type and contains no arguments.

Similar to class data members, class member functions are classified as either private or public. For example, the get_info() and the display() functions are public but the sell_car() function is private.

2. **How do we call public member functions and private member functions?** Like any other ordinary function, a public member function can be called from any place in the program. To call a public member function, we need to provide not only the function name but also the object to which it belongs. For example, the statements

```
newcar.get_info("Mary", 1998, 6543.2);
oldcar.get_info("John", 1921, 1234.5);
```

call the member function get_info with different actual arguments. The first call is for the object newcar and the second call is for the object oldcar. Both objects belong to the same class, Car. Note that we use a dot operator to connect the object and its member function (if we call the member function through a pointer to the object, we use the -> instead of the dot operator. After the call, the actual arguments are passed to the object. For example, after the first call, the information Mary, 1998 and 6543.2 are assigned to the data members newcar.owner, newcar.year_made and newcar.price, respectively. These assignments are made because, in the body of the function, get_info, the variables owner[ ], year_made and price,

are used. Note that within function get_info, no declarations for the variables owner, year_made or price have been made. This is because they are members of the Car class. The display() function then displays the object information on the screen after being called with newcar.display() and oldcar.display().

Unlike a public member function, a private member function can be called only by other member functions within the same class. For example, the sellcar() member function is private, we cannot call it from the main() function. Instead, we call it from the display() member function, which also belongs to the class Car. Because display() is already associated with an object when it is called, within display() it is not correct to call sellcar() with an object, such as oldcar.sellcar().

3. **How do we access private data members?** The data member price in the class Car is private. Any private data member can be accessed only by member functions belonging to the same class. For example, the class Car's member function get_info() uses the statement

```
price = cost;
```

to assign the value of cost to the private data member price. Another member function, display(), then uses the statement

```
cout << "price = " << price << endl;
```

to display price on the screen.

Note that in calling the get_info member function, an object is already associated with the member. Therefore, it is not correct to use the object's name with the data member, such as oldcar.price, within the function.

4. **How do we develop a member function?** The procedure for developing a member function is similar to that for developing any other function. We must declare (write the function prototype) and define (write the function body) of the function. For example, the statements

```
prototype  -- void     get_info(char *who, int year, double cost);
definition --- void Car::get_info(char *who, int year, double cost)
{                      ...
                       ...function body
}
```

show how the get_info member function is declared and defined. Note that, in the function definition, the scope resolution operator (::) is used. This operator indicates that the function behind the symbol belongs to the class in front of the symbol. This means we can declare another member function as get_info() within the same program, as long as the second get_info() does not belong to the class Car.

5. **What is encapsulation?** The process of linking data and the functions that manipulate the data into a single object is called *encapsulation*. Encapsulation provides an efficient way of allowing and restricting access to data. This is a feature that C lacks. In C, data and functions are not encapsulated, they are separate entities.

   When the function display is called as oldcar.display, all of the oldcar data are automatically made accessible to the function display, even though no data are explicitly transferred (copied) through the argument list. You can see that this is true because the function clearly has no arguments. Note that the values of oldcar.owner, oldcar.year_made and oldcar.price have been accessed by the function. This is possible because of encapsulation (see Fig. 9.7).

6. **What is the effect of encapsulation on the availability of data to functions?** In C, we saw that the only way to transfer information to a function is through the parameter list or the use of global variables. (We

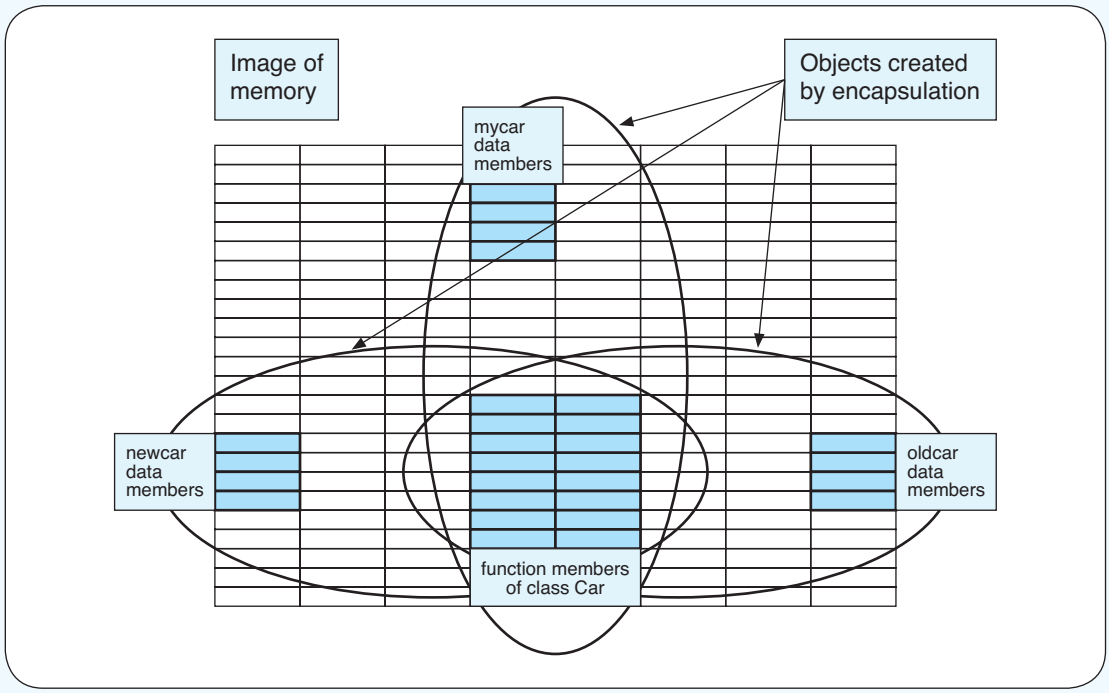


Fig. 9.7 ***An image of memory that illustrates the concept of objects and encapsulation. The data members of each object are separate and unique. However, the function members (i.e. the instructions stored in memory) are shared among the objects in a class. The data members are linked with the function members in an object.***

discourage the use of global variables unless they are absolutely necessary because they make a program less modular.) Because C++ uses encapsulation, a member function automatically has access to the data members of an object when it is called with that object's name. It is not necessary to pass data members to a member function through the parameter list. However, because data members are not global variables, we maintain a type of modularity to the program design. This is a concept of object-oriented programming that is lacking in C. It is illustrated in Fig. 9.8.

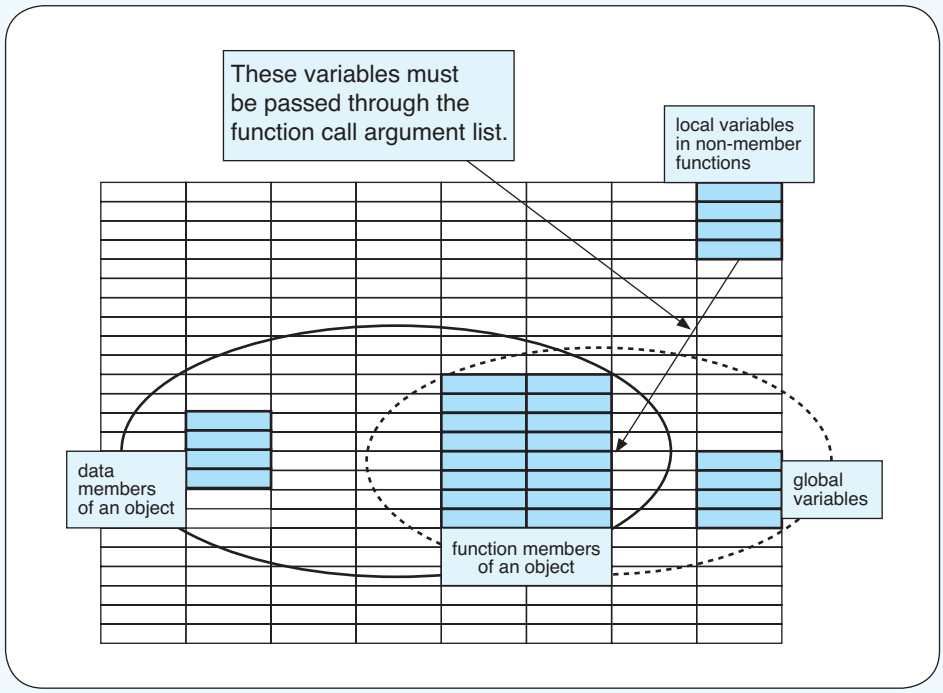


Fig. 9.8 ***Function members of an object have direct accessibility to the data members of the object, as well as to global variables (as indicated by the dashed ellipse). However, local variables in non-member functions must be passed through an argument list.***

7. **In general, how can we access both public and private data members from non-member functions?** From a non-member function we must use an object's name to access any member. This is illustrated in Fig. 9.9. We can access public data members directly by using the object's name as we did from main (which is a non-member function) in this lesson's program using oldcar.colour and oldcar.year_made. We can also call public function members using the object's name as we have done in main, using oldcar.get_info() and oldcar.display().

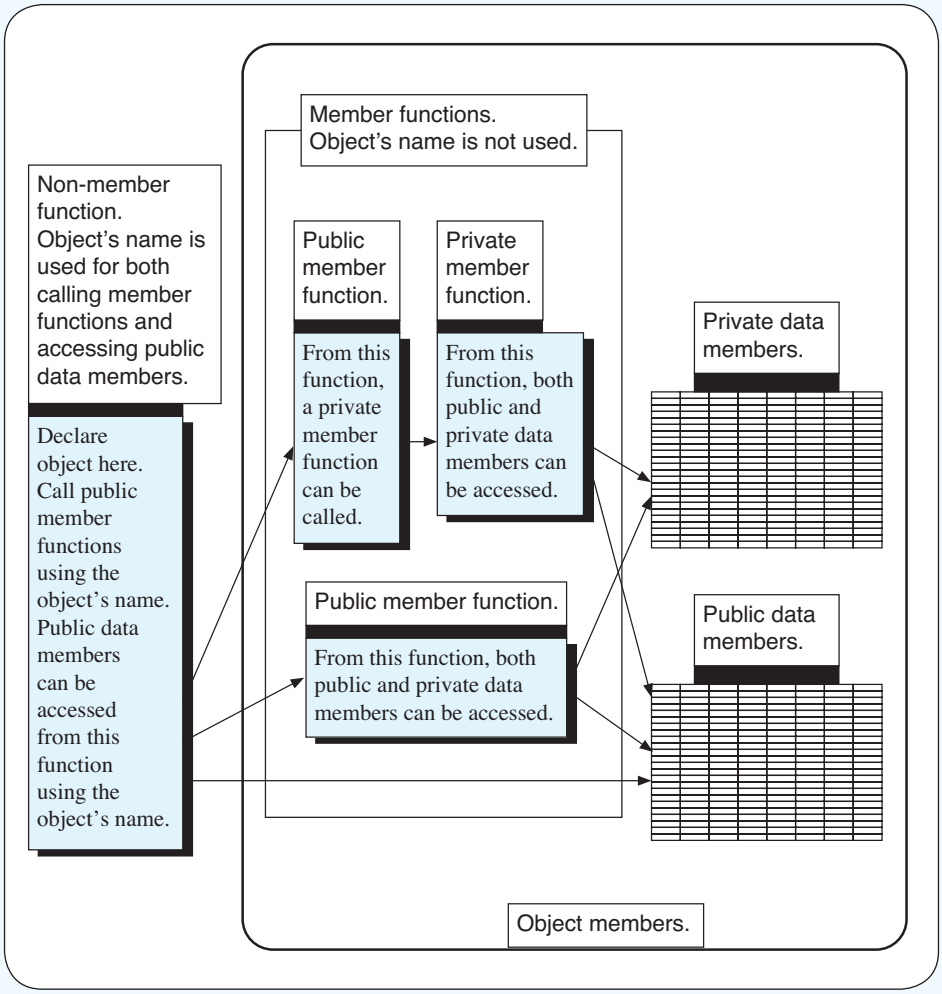


Fig. 9.9 ***Accessing both private and public data members from non-member functions. Notice that it is necessary to call a public member function from a non-member function to access private data members. Public data members can be accessed directly from non-member functions. In all cases, the object's name must be used in the non-member function to access the public member functions or public data members. Compare this figure to Fig. 9.10.***

However, from a non-member function, we cannot access directly any private member (which is neither a data nor function). Instead, we must first call a public member function and have it call a private member function or access private data members. In this lesson's program, from main we first call the public member function display with oldcar.display, which

in turn calls the private member function sellcar (see Fig. 9.10). Note that, within display, we do not use the call oldcar.sellcar because the object oldcar is already established with the call to display. Also, from main we call the public member function get_info with oldcar.get_info, which in turn uses the private data member price. We need not use oldcar.price within get_info because the object oldcar is already established with the call to get_info.

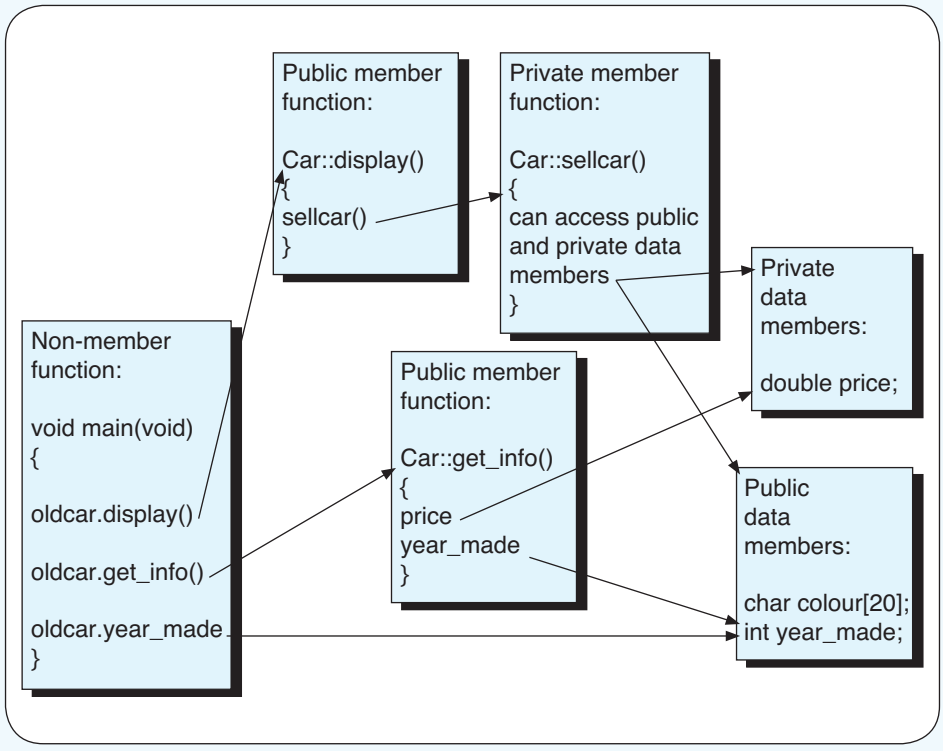


Fig. 9.10 ***How this lesson's program accesses public and private data members from the non-member function main. Compare this figure to Fig. 9.9.***

8. Why do we place the definition of class Car outside the main() function?
The class Car contains member functions that are not inline functions. The C++ compiler allows only classes with inline member functions (which have functions) to be local, otherwise, they must be global. Therefore, class Car must be global and be placed outside the main() function. The impact of this is that all non-member functions can call public member functions by first declaring an object of that class type.

9. **Does this lesson's program have a typical class definition?** No, a typical definition has the data members declared as private and the function members as public. This form allows any function to call a member function but allows only member functions to access data members. Such an arrangement protects the data members from being modified by unauthorised functions but allows great access to member functions. We have not used a typical form in this lesson's program simply because we are interested mainly in illustrating the many features of classes and objects.

Concept Recap

1. Member functions are defined within the definition of the class to which they belong.
2. To call a public member function, we need to provide both the function name and the object to which it belongs.
3. A private member function can be called only by other member functions within the same class.
4. The process of linking data and the functions (public and private) that manipulate the data into a single object is called *encapsulation*. Through encapsulation, a member function automatically has access to the data members of an object when it is called with that object's name.

Exercises

1. The blood pressure for a given individual depends on the person's age, sex, health and other environmental factors. Table 9.1 shows the average normal blood pressure for adults aged between 20 and 49 in a tested area. The information displayed in Table 9.2 is from four of the research subjects.

Table 9.1 ***Average normal blood pressure for adults***

Age (in years)	Blood pressure (in mm/Hg)
20–24	119
25–29	121
30–34	123
35–39	125
40–44	128
45–49	130

Table 9.2 ***Blood pressure information***

Patient's name	Age	Blood pressure
Jerry	42	132
Linda	36	124
Mary	22	118
Ken	46	144

Based on this information, write a program to

a. Develop a class named *Normal_pressure* to read the data in Table 9.1 (all data members are public).
b. Develop a class named *Individual_pressure* to read the data in Table 9.2 (all data members, except patient's name, are private).
c. Compare the blood pressure of each individual with the normal blood pressure information for his/her age group given in Table 9.1. Indicate (in percent) how high or how low the blood pressure of each person is.
d. Display the information obtained in step (c) as follows:

Patient's name	Age	Blood pressure	Result
Jerry	42	132	3.1% higher than normal
Linda	36	124	0.8% lower than normal
Mary	22	118	...
Ken	46	144	...

2. As a programmer, you are asked to develop a subject index for an Internet web page for a travel agency that specialises in the entertainment and leisure industries. Part of the subject index follows:

Subject	Index	Phone number
Amusement places	321	418-221-3098, 800-761-2001
Boat renting	456	798-652-1980
Campgrounds and parks	987	238-886-1899
Night clubs	765	457-734-1934, 888-856-3467

To use the program, the user should enter a password (which can be any one of the following: A123, X987 and K456) and then enter the index or the first four characters of the subject. If the input is correct, the program will display the subject and related telephone numbers on the screen. You are asked to

a. Declare a class and use one of its member functions to read the data listed (all data members, except the telephone number and password, are public).
b. Call another member function to check whether the password and other input data are correct.
c. If the input data are correct, then call another member function to display the output on the screen.

Lesson 9.8 Constructor and Destructor Functions

Topics

- Constructor functions
- Destructor functions

In the last lesson, we learnt that a C++ object encapsulates data and functions and thus provides an efficient way of managing data. As we have seen in many of our programs, initialisation of data is often necessary. C++ provides special functions, called *constructor functions*, that are called automatically when an object is declared. These functions can be used to initialise the values of data members and perform other necessary initial operations.

Also, C++ has destructor functions that are called automatically when an object goes out of scope (which is similar to a variable going out of scope in C, as occurs to a local variable when a function completes execution). These commonly have the purpose of freeing memory reserved with dynamic memory allocation. In this lesson, we perform no particular action with the destructor functions because we are simply illustrating their potential use.

This program prints the telephone numbers of three people, by creating a class with data members that hold the telephone numbers and function members that initialise and print the numbers. It prints the telephone numbers to the screen and an indication that an object has been destroyed.

Source Code

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


class Phone
{
        public:
        void     print_number (char *who);
        long     get_phone_no (char *who);
        Phone (char *city);
        ~Phone();

        private:
        long     phone_no;
        int      area_code;
};

Phone::Phone(char *city)
{
        if       (strcmp(city, "Denver")==0) area_code = 303;
        else if  (strcmp(city, "Boston")==0) area_code = 617;
        else                                 area_code = 800;
}

Phone::~Phone()
{
        cout<<"Object destroyed"<<endl;
}


void main(void)
{
        Phone caller1("Denver"), caller2("Boston"), caller3("USA");


        caller1.print_number    ("John");
        caller2.print_number    ("Mary");
        caller3.print_number    ("Tom");
}

void Phone::print_number(char *who)
{
        cout << "who          = " << who                << endl;
        cout << "Area_code    = " << area_code          << endl;
        cout << "Phone_no     = " << get_phone_no(who) << "\n\n";
}

long Phone::get_phone_no(char *who)
{
        if        (strcmp(who,"John")==0)      phone_no=1112233;
        else if   (strcmp(who,"Mary")==0)      phone_no=4445566;
        else                                   phone_no=7778899;

        return(phone_no);
}
```

The constructor function must have the same name as the class. It cannot have a return type. It can accept arguments. It is called automatically when any object of the given class is declared.

The destructor function must have the same name as the class but with the symbol ~ in front. It cannot have a return type. It is called when an object goes out of scope.

Definition of the constructor function.

Definition of the destructor function.

Arguments used in calling the constructor function.

Declarations of objects.

Calling the public function print_number through objects caller1, caller2 and caller3.

Output

```
who          = John
Area_code    = 303
Phone_no     = 1112233

who          = Mary
Area_code    = 617
Phone_no     = 4445566

who          = Tom
Area_code    = 800
Phone_no     = 7778899

Object destroyed
Object destroyed
Object destroyed
```

Explanation

1. **What are constructors?** A *constructor* is a special function used to initialise or allocate memory for an object. Each time an object of a specified class is declared, the function is called automatically.

   The name of a constructor function is always the same as the name of its class. The function may contain any number of arguments or no arguments at all. For example, the statement

   ```
   Phone (char *city);
   ```

   within the definition of class Phone declares that Phone is a constructor function, since the function name is identical to its class name. The function contains one formal argument, city (see Fig. 9.11).

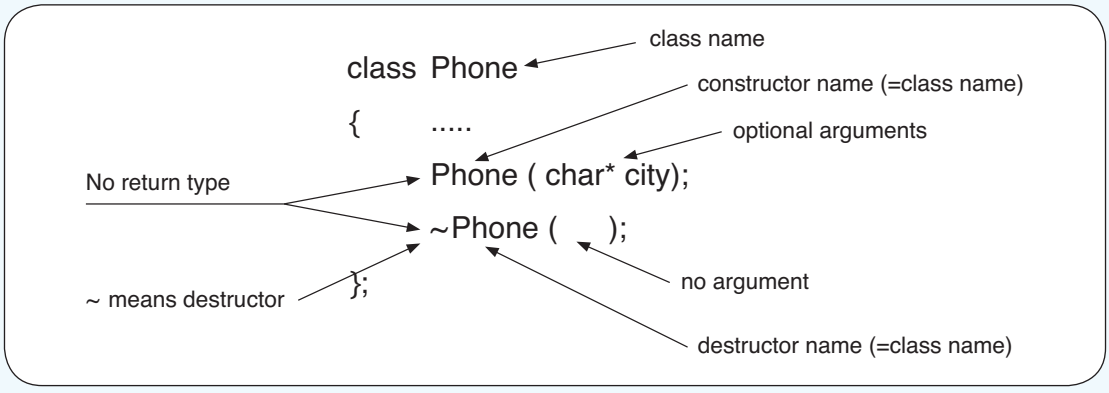


Fig. 9.11 ***Constructor and destructor***

A constructor is an optional function. When you define a class, you may define no constructor at all, in which case the class you define automatically uses a do-nothing default constructor. The default constructor does not initialise or check any data members in the class. You may define more than one constructor in a given class, each with a different argument list. These constructors behave like overloaded functions. For further details, please see the manual of your C++ compiler.

2. **How do we define a constructor?** A constructor is defined more or less like any other regular function. In general, the definition starts with the class name, followed by the scope resolution operator, the name of the constructor, the argument list and the function body. For example, the statements

```
Phone::Phone(char *city)
{
   if (strcmp(city, "Denver")==0) area_code = 303;
   else if  (strcmp(city, "Boston")==0) area_code = 617;
   else area_code = 800;
}
```

define the constructor function Phone. The function contains one argument, and the argument is used to select the appropriate area_code. The data initialisation and checking process can be as extensive as you like in a constructor function. The function body typically includes code for the assignment of values to data members, allocation of memory or checking the validity of input data.

Note that a constructor has no type (not even a void type) and can never return a value.

3. **How do we declare an object using a class that contains a constructor?** If the constructor contains arguments, then we declare the object with the actual arguments; otherwise, we declare the object with no actual arguments. For example, the statements

```
Phone caller1("Denver"), caller2("Boston"), caller3("USA");
```

declare three objects, caller1, caller2 and caller3, each with different actual arguments, Denver, Boston and USA, respectively. After the declaration, the constructor is called automatically, and the three objects are given values for the member area_code: 303, 617 and 800, respectively.

4. **What is a destructor?** A *destructor* is an optional member function that is called automatically when a class object is out of scope. The name of a destructor starts with a tilde (~), followed by its class name. A destructor has no arguments and returns no value. For a simple class, we usually need

not write a destructor. For a more complicated class, a destructor is used to do clean-up work when an object is destroyed. This program includes a destructor

```
~Phone();
```

that prints a message when an object is destroyed. In this program, the three objects – caller1, caller2 and caller3 – are out of scope when the program terminates. At that time, the destructor function for each object is called.

Concept Recap

1. A *constructor* is used to initialise or allocate memory for an object.
2. A destructor is an optional member function that is called automatically when a class object is out of scope. The destructor is usually used to do clean-up work in a complex class.

Exercises

1. Write a program that uses a specified character to draw a rectangle on the screen. The program should contain a constructor that uses the following formal arguments:

```
char border  - any printable character to be used to draw
               the border of a rectangle.
double left  - left coordinate of a rectangle on the
               screen, 0<=left<=80
double right - right coordinate of a rectangle on the
               screen, right>=left, 0<=right<=80
double top   - top coordinate of a rectangle on the
               screen, 0<=top<=25
double bot   - bottom coordinate of a rectangle on the
               screen, bot>=top, 0<=bot<=25
```

For example, if a user enters

```
* 10 60 5 20
```

the program should use the character * to draw a rectangle from $x = 10$ to $x = 60$ and $y = 5$ to $y = 20$. If the user enters incorrect data, such as left = 99, then the program should use the default left = 0 instead to draw the rectangle.

2. Write a program to calculate the total number of days between 1 January and a specified day, month and year. The program should contain a class that

a. Has a member function to read the input date. The input format is day/month/year.
b. Has a constructor function to initialise the number of days in any month of a given year. For a leap year, the total days in February is 29.
c. Has a member function to display the output.

For example, if a user enters

```
3/5/2012
```

The program should display

```
There are 65 days between 1/1/2012 and 3/5/2012.
```

Lesson 9.9 Inheritance

Topics

- Inheritance
- Base and derived classes
- Reusable code

In daily life, we classify items with common features into a group. The item with the most general characteristics is used to form the root or base, the base is then used to derive other items with more specific features. C++ allows us to model such a system using a base class and derived classes.

For example, in C++ we may select Motor_vehicle, a vehicle moving on wheels, as our base class. We give the base class the characteristic of num_headlights = 2. From Motor_vehicle, we can derive a class, Bus, and another class, Truck. The class Bus contains num_headlights = 2 and may also have seat_number = 50. Truck has num_headlights = 2 and may also have load_capacity = 10 tons. Using C++ base and derived classes, we need not include num_headlights = 2 in the Bus and Truck classes, as these characteristics are obtained automatically from the base class, Motor_vehicle.

The mechanism of obtaining features from simpler and more general types is called *inheritance*. Inheritance is one of the most important features that distinguishes C++ from C. Inheritance allows us to *reuse* and extend existing classes without having to rewrite the original code. In the program for this lesson, we develop a base class named *Parent* and a derived class named *Child*. The object of the Child class, son, inherits the members of the Parent class. The program prints the names and other information of a parent and a child.

Source Code

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


class Parent
{
     public:
     void display(void);
     Parent();


     private:
     char     last_name [20];
     char     first_name[20];
     double   income;
};


class Child :  public Parent
{
     public:
     void info(char *first, int age);
     void print_info(void);
     Child();


     private:
     char first_name[20];
     int  age;
};


Parent::Parent()
{
     strcpy(last_name,"Smith");
     strcpy(first_name,"John");
     income=1234.56;
}


Child::Child():Parent()
{
}


void main(void)
{
     Child    son;


     cout << "About son     information ---------------------------\n";
     son.info("Ali", 23);
     son.print_info();
```

The base class is Parent.

Both Parent and Child have first_name arrays. However, only Parent has a last_name array.

This indicates that Child inherits from Parent.

Child is a derived class.

Constructor function for Parent. This function gives the Parent class a first_name, last_name and income.

Constructor function for Child. Note that the base function Parent is given.

Declaring son to be an object of class Child. Both the Parent and Child constructor functions are executed with this declaration.

Calling Child function info as a member of object son. This function gives son a first name and age.

```
     cout << "\n\nChild uses Parent's member function in main-----\n";
     son.display();
}
```

Calling display() from main. We associate an object with it (son).

```
void Child::print_info(void)
{
     cout << "\nChild uses Parent's member function in print_info --";
     cout << "\nChild's first_name     = "<<first_name<<endl;
     display();
}
```

Calling display() from print_info. We need not associate an object with it because, in calling print_info, we have already referenced an object (son).

```
void  Parent::display(void)
{
     cout << "Parent firstname = " << first_name <<endl;
     cout << "Parent last_name  = " << last_name  <<endl;
     cout << "Parent income    = " << income    <<endl<<endl;
}
```

Here, first_name refers to first_name in the Parent class.

```
void Child::info(char *first, int years)
{
     strcpy(first_name, first);
     age=years  ;
     cout << "Child firstname = " << first_name       << endl;
     cout << "Child age       = " << age      << endl;
}
```

Here, first_name refers to first_name in the Child class.

Output

```
About son    information -------------------------------
Child firstname = Ali
Child age       = 23

Child uses Parent's member function in print_info ------
Child's firstname = Ali
Parent firstname  = John
Parent last_name  = Smith
Parent income     = 1234.56

Child uses Parent's member function in main ------------
Parent firstname  = John
Parent last_name  = Smith
Parent income     = 1234.56
```

Explanation

1. **What are base and derived classes?** A class from which new classes are derived is called a *base class*. A class derived from a base class is called a *derived class*. A derived class can further be used as a base class to derive more next generation classes. Therefore, we can create class hierarchies where each class serves as a parent or root of a new class.

2. **How do we define a C++ derived class?** A C++ derived class is generated from a base class. Therefore, before we can define a derived class, we have to have a base class defined. Any regular C++ class, with or without a constructor, can be used as a base class. However, if we want the derived class to inherit some features from the base class, we need to include a constructor function in the base class to initialise those features. For example, in this program, we use a base class named *Parent* and the constructor function

```
Parent::Parent()
{
   strcpy(last_name,"Smith");
   strcpy(first_name,"John");
   income=1234.56;
}
```

to initialise the data members, last_name, first_name and income, of class Parent. Any class derived from the base class will inherit the values of all these data automatically. Inheritance is achieved by taking existing classes and deriving new classes from them. A derived class may selectively inherit some members, reject or modify other members from the base class and add new members of its own. If we do not include a constructor in our base class, any object defined by its derived class will still contain all its base class data members and member functions, although the data members will not have initial values.

Once we have defined a base class, we can define derived classes for it. A base class can be used by any number of derived classes. Each derived class may have data members or member functions that are different from that of the other derived classes. In this lesson, we define only one derived class, named *Child*. The syntax for a derived class is as follows:

```
class derived_class_name : access_modifier   base_class_name
{
   derived class data and function members
   ...
}
```

access_modifier is used to specify the accessibility of the derived class and must be one of the keywords *private*, *protected* or *public*. By selecting the proper *access_modifier*, we control how data members and member functions of its base class are to be accessed from the derived class. In general, for a given derived class, the access to any member of its base class can be controlled to be more restrictive but never less restrictive. This means that lower level classes may not be allowed to access members of an upper class. This is true in life and also true in C++. Without this control, anyone can write a derived class to modify or destroy data in a base class. The statements below

```
class Child :  public Parent
 {
         public:
         void info(char *first, int age);
         void print_info(void);
         Child();

         private:
         char first_name[20];
         int  age;
 };
```

define Child to be a derived class of the base class Parent; *access_modifier* is set to public. The public modifier allows objects of the Child class to have the same freedom to access members of the Parent class as any other objects of the Parent class. If the access modifier is private or protected instead of public, Child objects may not be able to reach certain members of the Parent class. For further details, please see the manual of your C++ compiler. The definition also tells us that the derived class Child contains two data members, first_name and age, member functions info() and print_info(), and a constructor Child(). For the general form of a derived class, see Fig. 9.12.

3. **How do we define a constructor for a derived class?** The syntax of the constructor for a derived class is as follows:

```
dc_name::dc_name(dc_list):Bc_name(Bc_list)
{
 derived class constructor function body
...
}
```

where *dc_name* is the derived class name, *dc_list* is the derived class parameter list, *Bc_name* is the base class name and *Bc_list* is the base class parameter list.

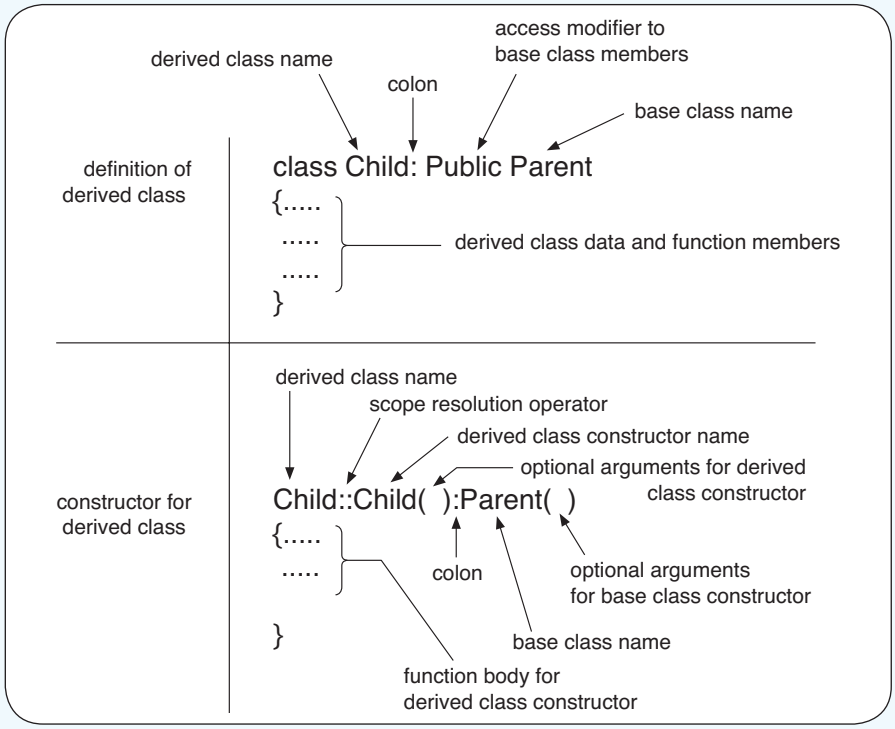


Fig. 9.12 *The general form of a derived class*

For example, the statements

```
Child::Child():Parent()
{
}
```

define the constructor for the derived class Child. In this lesson's program, both the base class and the derived class constructors have empty parameter lists. This text includes no example of a base class constructor with a non-empty parameter list.

4. **How do we declare objects of a derived class?** Objects of any derived class can be declared directly. We are not required to declare objects of a base class before declaring objects of a derived class. For example, the statement

```
Child    son;
```

declares son as an object of Child without declaring any object of the Parent class. After we declared the son object of the Child class, the constructor of its base class, Parent, and then the constructor of the derived class Child are executed automatically.

5. **When an object of a derived class is declared, how much memory is reserved?** New memory for data members of *both* the derived and base classes is reserved.

6. **When the data members are private and the function members are public for both the base and derived classes, how can we access a derived class object's data members from a non-member function?** First, we declare an object of the derived class type in the function. Then, we call a public derived class or base class function to access the private data members. This is illustrated in Fig. 9.13. Study the figure to understand why we cannot go directly from the non-member function to the private data members.

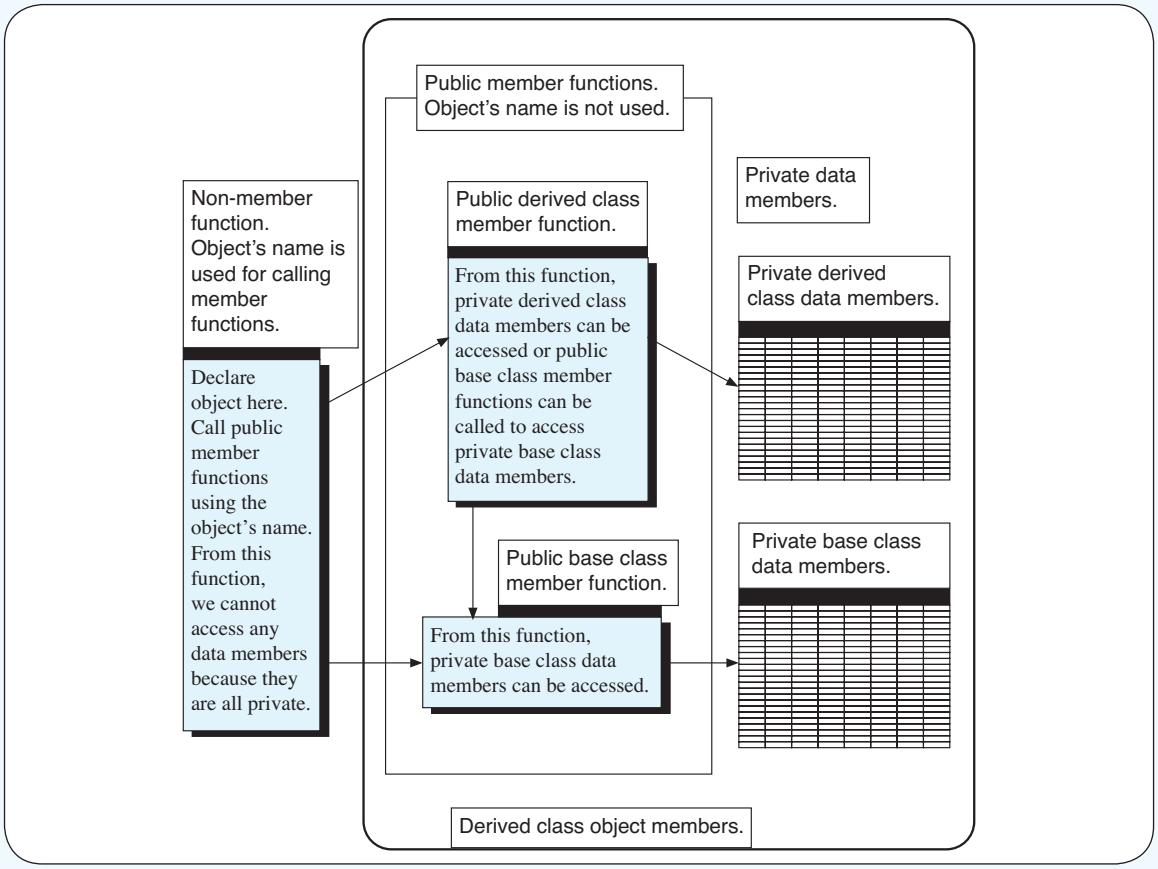


Fig. 9.13 ***In this illustration we assume that all data members are private and all function members are public. To access the private data members of the derived class from a non-member function, we must call a public derived class function. To access the private data members of the base class from a non-member function, we can call a public base class function directly or first call a public derived class function, which in turn calls a public base class function. Compare this figure to Fig. 9.14.***

7. **How have we accessed private data members of both the derived and base classes in this lesson's program?** This is illustrated in Fig. 9.14. The figure shows that it is necessary to use only the object's name in the non-member function and that we can access private base class data members only from a base class member function. Compare this figure to Fig. 9.13. Notice that Fig. 9.14 parallels Fig. 9.13.

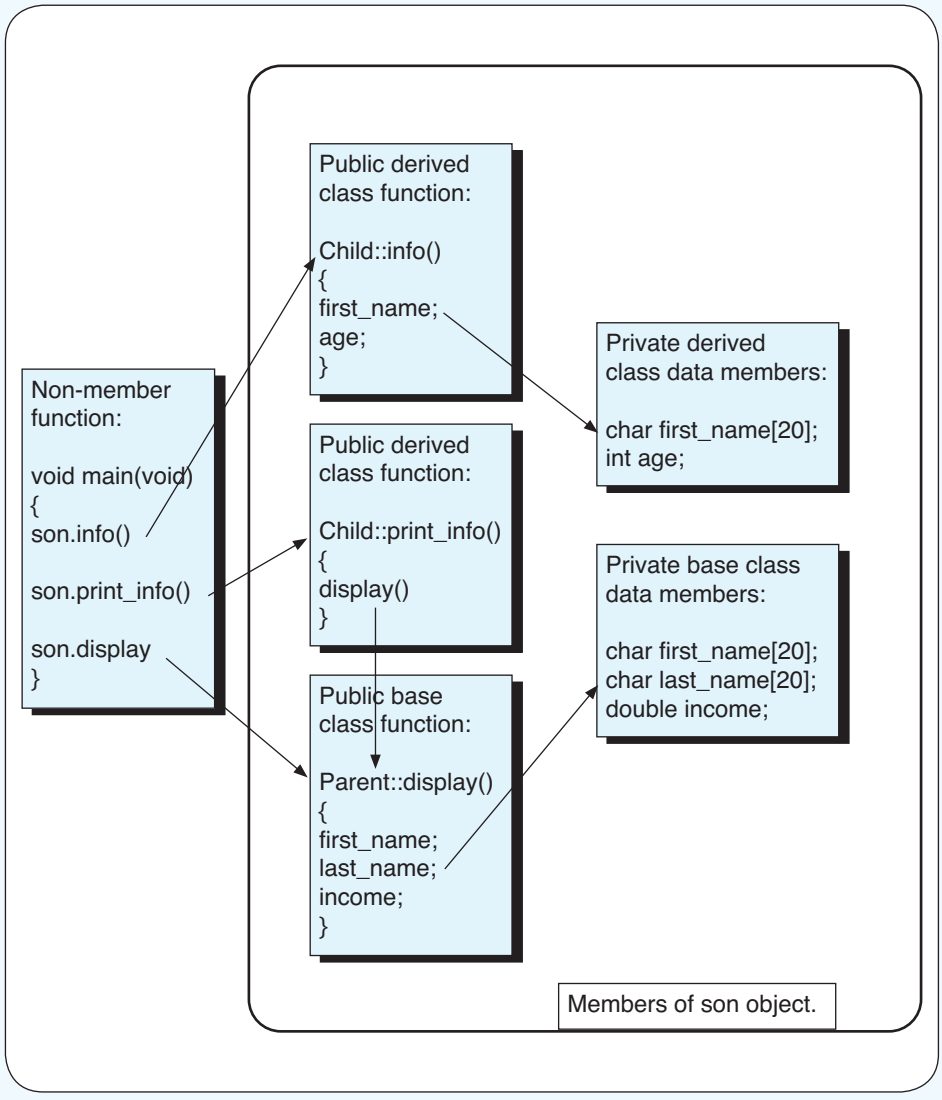


Fig. 9.14 ***How this lesson's program accesses private data members in both the derived and base classes from the non-member function main. Compare this figure to Fig. 9.13.***

Concept Recap

1. We can use the derived class to extend the function of a base class, so as to reuse the code of the base class

   ```
   class derived_class_name : access_modifier   base_class_name
   {
      derived class data and function members
      ...
   }
   ```

   access_modifier is used to specify the accessibility of the derived class and must be one of the keywords *private*, *protected* or *public*.

2. Private data members of the base class can only be accessed from a base class member function of a derived class.

Exercises

1. Write a program that has a base class Car and two of its derived classes Bus and Truck. The classes must contain the following data members:

```
Car   : wheel, colour
Bus   : seat_num
Truck: load_capacity
```

The output should be as follows:

```
Base class CAR information---------------
Wheel = 4
Colour = White

Derived class Bus information------------
Wheel = 4
colour = Yellow
Seat_num = 50

Derived class truck information----------
Wheel = 4
colour = Blue
Load_capacity = 8 tons
```

2. Given the following definition for a Circle class

```
class Circle
{
     double x0, y0, radius;
}
```

derive a new class to represent a Cone class. The class contains a member function that can calculate and display the volume of a cone. Test your program with a cone that has the following parameters:

```
x0=100.0
y0=200.0
radius=300.0
cone height = 400.0
```

3. Based on the Cone class (which is derived from the Circle class) in Exercise 2, derive a new class to represent a Truncated_cone class. The class contains a member function that can calculate and display the volume of a truncated cone. Test your program with a truncated cone that has the following parameters:

```
Top of truncated cone
x0=100.0
y0=200.0
radius=300.0

Bottom of truncated cone
x0=100.0
y0=200.0
radius=500.0
height of truncated cone = 400.0
```

Application Program 9.1: Electrical Circuits

The design of classes is a very important part of writing C++ programs. In this text, we lack the space to devote to class design. Instead, we illustrate with an example some of the aspects of designing classes. What we have here is not meant to be a complete or thorough representation of the development of classes. Our main interest is to give you some exposure to C++. Hence, the representations are simple rather than general or efficient.

Problem Statement

The electrical circuit shown in Fig. 9.15 contains seven resistors and a power supply with a DC voltage of 110.

Write a program to find the magnitude of the current across the power supply.

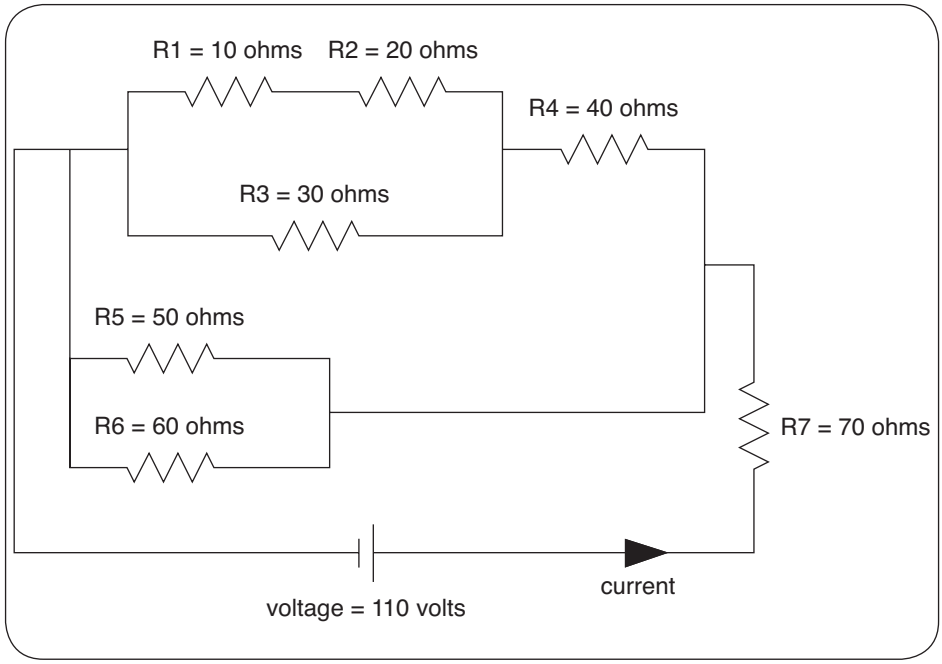


Fig. 9.15 ***An example of an electrical circuit***

For this circuit, the input is

```
s 10 20
s 30 0
p
s 40 0
s
p 50 60
p
s 70 0
s
```

where *s* indicates that two resistors are in series and *p* indicates they are in parallel.

Two resistors can be input per line. The first line indicates that R1 and R2 are in series. The second line uses a dummy value of 0 being in series with R3. The third line indicates that the two previous sets of resistances (lines 1 and 2) are in parallel with one another. The fourth line indicates that R4 and a dummy resistor of 0 are in series. The fifth line indicates that R4 and all of the rest of the resistance previously calculated are in series. The sixth line indicates R5 and R6 are in parallel. Then the seventh line indicates that the subcircuit formed by R5 and R6 is in parallel to the rest of the resistance previously calculated. The eighth

line indicates that R7 and a dummy resistor are in series. The last line indicates that R7 and the rest of the previously calculated resistance are in series. We use this input to calculate the total resistance of the circuit and the current at the power supply.

It can be seen from this that it takes some skill and thought to prepare the input data for this particular circuit. This is true for many advanced engineering problems. You will find that, in practice, it is necessary to be a talented software engineer to handle programs.

Solution

Relevant Equations and Background Information. To find the magnitude of the current I we use Ohm's law:

$$I = V/R \tag{9.1}$$

where V is the voltage and R is the total resistance in the circuit. In this example, the voltage is given (110 volts) but the total resistance must be calculated.

When two resistors, R1 and R2, are connected in parallel to each other, the combined resistance, R12, is

$$\text{R12} = 1/(1/\text{R1} + 1/\text{R2}) \tag{9.2}$$

However, when they are connected in series, the combined resistance is

$$\text{R12} = \text{R1} + \text{R2} \tag{9.3}$$

Specific Example

In this example, the circuit contains seven resistors connected in a mix of parallel and series patterns. We find the total resistance in a sequence of steps. For each step, we combine two resistors into one. By doing this a number of times, we can find the total resistance. The following hand calculation illustrates how the total resistance is found. We use two arrays, r[ ] and rtot[ ], to assist us in the calculation. Refer to Fig. 9.15 to see how the resistors are connected:

1. Combine r[0] = 10 and r[1] = 20 in series to get rtot[0] = 10 + 20 = 30, using equation (9.3).
2. Combine r[0] = 30 and r[1] = 0 in series to get rtot[1] = 30 + 0 = 30.
3. Combine rtot[0] = 30 and rtot[1] = 30 in parallel to get rtot[0] = 1/(1/30.0 + 1/30.0) = 15, using equation (9.2).
4. Combine r[0] = 40 and r[1] = 0 in series to get rtot[1] = 40 + 0 = 40.

5. Combine rtot[0] = 15 and rtot[1] = 40 in series to get rtot[0] = 15 + 40 = 55.
6. Combine r[0] = 50 and r[1] = 60 in parallel to get rtot[1] = 1/(1/50.0 + 1/60.0) = 27.27.
7. Combine rtot[0] = 55 and rtot[1] = 27.27 in parallel to get rtot[0] = 1/(1/55.0 + 1/27.27) = 18.23.
8. Combine r[0] = 70 and r[1] = 0 in series to get rtot[1] = 70 + 0 = 70.
9. Combine rtot[0] = 18.23 and rtot[1] = 70 in series to get rtot[0] = 18.23 + 70 = 88.23. This is the total resistance of the circuit.

The current then is obtained from equation (9.1) as

$$I = 110/88.23 = 1.30 \text{ amps}$$

Note that in this solution, we have used rtot[0] and rtot[1] alternately to hold the total resistance.

Data Structures and Classes

To keep this first problem as simple as possible, we use only one class and one object. The class is called *Circuit* and its public members are functions and private members are data. The class is

```
class Circuit
{
        public:
        double series(double r[ ]);
        double parallel(double r[ ]);
        void find_resistance(void);
        void find_current(void);
        Circuit (double dummy);

        private:
        double r[2], rtot[2];
        double voltage, current;
        char flag;
};
```

The meanings of the members are indicated in the source code annotations. The single object (res_circuit) is declared as

```
Circuit     res_circuit1(110.);
```

The argument in the declaration initialises the power supply to be 110 volts in the constructor.

Algorithm

The algorithm can be seen from the specific example to be as follows:

1. Read the input data for the first subcircuit and calculate its total resistance.
2. Read the input data for the second subcircuit and calculate its total resistance.
3. Read the relationship (parallel or series) for the two subcircuits and calculate the total resistance.
4. Read the input data for the next subcircuit and calculate its total resistance.
5. Read the relationship (parallel or series) for the previous two subcircuits and calculate the total resistance.
6. Repeat steps 4 and 5 until the entire circuit is complete.
7. Calculate the current using equation (9.1).

Because we do not focus on the procedural aspects of the programs for this chapter, we leave it to you to verify that the loop in function find_resistance performs these steps in a manner similar to the example calculation we have given.

Source Code

```
#include <iostream.h>

class Circuit
{
        public:
        double series(double r[ ]);
        double parallel(double r[ ]);
        void find_resistance(void);
        void find_current(void) ;
        Circuit (double dummy);

        private:
        double r[2], rtot[2];
        double voltage, current;
        char flag;
};
```

All of the function members are public.

Function to evaluate series subcircuit.

Function to evaluate parallel subcircuit.

Function to evaluate total resistance of entire circuit.

Function to evaluate current at the power supply.

All of the data members are private.

Resistances of subcircuit and entire circuit.

flag indicates parallel ("p") or series ("s").

```
Circuit::Circuit(double dummy)
{
        r[0]=0.0;
        r[1]=0.0;
        rtot[0]=0.0;
        rtot[1]=0.0;
        current=0.0;
        flag='s';
        voltage=dummy;
}

void main(void)
{
        Circuit res_circuit1(110.);

        res_circuit1.find_resistance();
        res_circuit1.find_current();
}

void Circuit::find_resistance(void)
{

        cout<<"Enter flag, r0, r1"<<endl;
        cin>>flag>>r[0]>>r[1];

        if(flag=='s') rtot[0]=series(r);
        if(flag=='p') rtot[0]=parallel(r);
        cout<<"Subtotal resistance="<<rtot[0]<<endl;

        for (int i=1; i<=4; i++)
        {
                cout<<"Enter flag, r0, r1"<<endl;
                cin>>flag>>r[0]>>r[1];
                if(flag=='s') rtot[1]=series(r);
                if(flag=='p') rtot[1]=parallel(r);
                cout<<"Subtotal resistance="<<rtot[1]<<endl;
                cout<<"Enter flag"<<endl;
                cin>>flag;
                if(flag=='s') rtot[0]=series(rtot);
                if(flag=='p') rtot[0]=parallel(rtot);
                cout<<"Subtotal resistance="<<rtot[0]<<endl;
        }

        cout<<"Total resistance="<<rtot[0]<<endl;

}
```

Constructor initialises the data.

The constructor function initialises the voltage to be 110 volts when the declaration is executed.

Finding the total resistance of res_circuit1.

Finding the current at the power supply for res_circuit1.

Entering and analysing the resistance of the first subcircuit.

Entering and analysing the resistances of the last four subcircuits.

Printing the total resistance of the circuit.

```
void Circuit::find_current(void)
{
        current = voltage/rtot[0];
        cout<<"Current at the power supply = "<<current<<endl;
}


double Circuit::series(double r[ ])
{
        double rtot;
        rtot=r[0]+r[1];
        return rtot;
}


double Circuit::parallel(double r[ ])
{
        double rtot;
        rtot=1./(1./r[0]+1./r[1]);
        return rtot;
}
```

Evaluating the current using equation (9.1).

Evaluating the resistance using equation (9.3).

Evaluating the resistance using equation (9.2).

Output

```
                Enter flag, r0, r1
Keyboard input  s 10 20
                Subtotal resistance=30
                Enter flag, r0, r1
Keyboard input  s 30 0
                Subtotal resistance=30
                Enter flag
Keyboard input  p
                Subtotal resistance=15
                Enter flag, r0, r1
Keyboard input  s 40 0
                Subtotal resistance=40
                Enter flag
Keyboard input  s
                Subtotal resistance=55
                Enter flag, r0, r1
Keyboard input  p 50 60
                Subtotal resistance=27.27
                Enter flag
Keyboard input  p
                Subtotal resistance=18.23
                Enter flag, r0, r1
Keyboard input  s 70 0
                Subtotal resistance=70
                Enter flag
Keyboard input  s
                Subtotal resistance=88.23
                Total resistance=88.23
                Current at the power supply=1.30
```

Comments

To keep this program as simple as possible, we eliminated data checking and avoided the possibility of a division by 0 if the data is entered incorrectly. However, these should be in a program that is meant for commercial use.

Modification Exercises

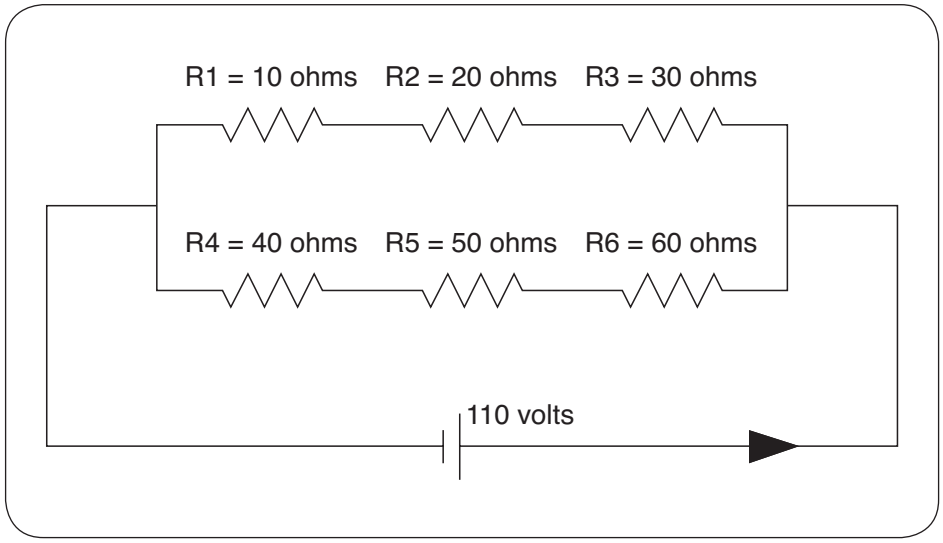
1. Modify the program so that it is not possible to have a division by 0 in function parallel.

2. Modify the program to handle two similar resistance circuits. The first should supply 300 volts and the second 450 volts.

3. Modify the program so that it can analyse three resistances instead of just two for each subcircuit.

Application Exercises

9.1. Write a program that can find the current in the circuit shown below and for any values of the resistances. Assume that many subcircuits of the type shown can be placed in the circuit.



9.2. The air pollution level of a city on a given day is a function of the time of day (in hours). As an environmental specialist, you have collected the following pollution level readings at different times:

Time	Pollution level
0:00	58
2:00	51
4:00	47
5:00	51
8:00	55
11:00	67
14:00	78
16:00	86
19:00	82
20:00	86
23:00	65

Write a program to plot the pollution level-time curve. The program should contain two classes. The first class should have a member function to read the input file as just shown and another member function to find the range of the data for plotting. The second class should handle the plotting routine.

9.3. As a software engineer, you are asked to write a section of a user-friendly interface for an application program. The section intends to multiply two numbers based on a user's input string. The numbers can be either real or complex. Use the following four input strings to test your program:

```
3 × 4
5 × (6 – 7i)
(–8 + 9i) × 10
(1 + 2i) × (–3 – 4i)
```

The program should generate the following output:

```
3 × 4 = 12
5 × (6 – 7i) = 30 – 35i
(–8 + 9i) × 10 = –80 + 90i
(1 + 2i) × (–3 – 4i) = 5 – 10i
```

The program should contain two classes. Objects belonging to the first class should be able to decompose the input string to correct numerical operators

and operands. Objects of the second class should perform the calculations and display the output on the screen.

9.4. Given three sides of a triangle *a*, *b* and *c*, its area can be calculated using the formula

heron's formula

$$\text{Area} = \sqrt{s'(s'-a)(s'-b)(s'-c)}$$

s' = semi-perimeter

$\Rightarrow$ formula should be

$$\text{Area} = \frac{1}{4}\sqrt{s(s-2a)(s-2b)(s-2c)}$$

$$\text{where } s' = \frac{a+b+c}{2} \text{ and } s = a+b+c$$

where *s* is the perimeter of the triangle. Write a program that contains a class named *Tri_area*. The class should contain a data member that accepts *a*, *b* and *c* as arguments and calculates the area of the triangle.