OBJECT ORIENTED PROGRAMMING USING C++

Contents

Chapter 1
Function

1.1 Simple Function

A function groups a number of program statements into a unit and gives it a name.

The most important reason to use functions is to aid in the conceptual organization of a program. Another reason to use functions is to reduce program size. Any sequence of instructions that appears in a program more than once is a candidate for being made into a function. The function's code is stored in only one place in memory, even though the function is executed many times in the course of the program. Figure 1.1 shows how a function is invoked from different sections of a program. Functions in C++ are similar to subroutines and procedures in various other languages.

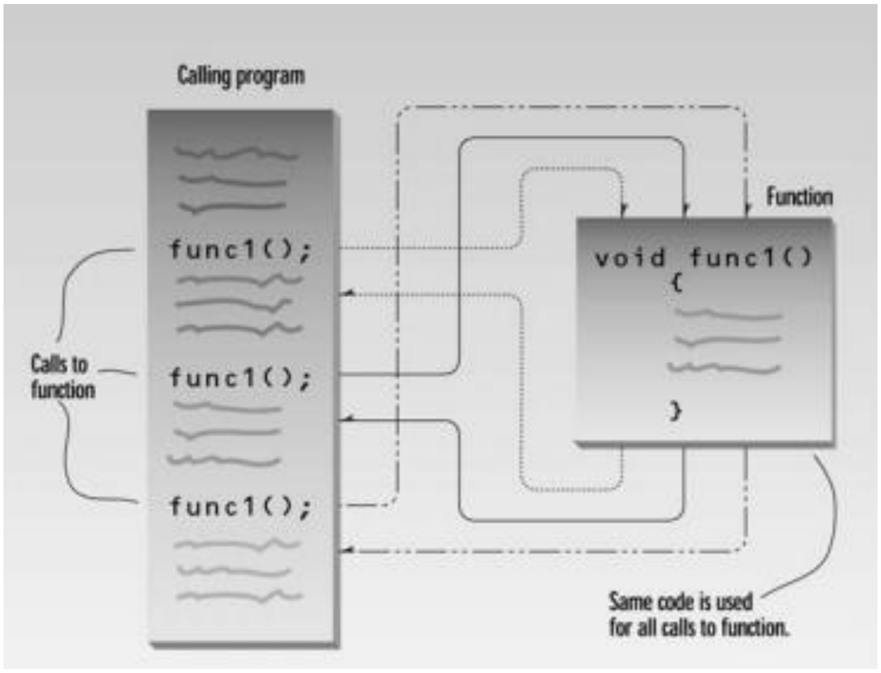


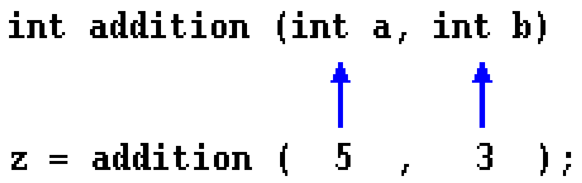
Figure 1.1 : Flow of control to a function

Example 1:

```
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
}
```

This program is divided in two functions: addition and main. Remember that no matter the order in which they are defined, a C++ program always starts by calling main. In fact, main is the only function called automatically, and the code in any other function is only executed if its function is called from main (directly or indirectly).

In the example above, main begins by declaring the variable z of type int, and right after that, it performs the first function call: it calls addition. The call to a function follows a structure very similar to its declaration. In the example above, the call to addition can be compared to its definition just a few lines earlier:



```
int addition (int a, int b)

z = addition (  5  ,  3  );
```

The parameters in the function declaration have a clear correspondence to the arguments passed in the function call. The call passes two values, 5 and 3, to the function; these correspond to the parameters a and b, declared for function addition.

Example 2:

```
// function example
#include <iostream>
using namespace std;

int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return r;
}

int main ()
{
  int x=5, y=3, z;
  z = subtraction (7,2);
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction (7,2) << '\n';
  cout << "The third result is " << subtraction (x,y) << '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " << z << '\n';
}
```

Similar to the addition function in the previous example, this example defines a subtract function, that simply returns the difference between its two parameters. This time, main calls this function several times, demonstrating more possible ways in which a function can be called.

Let's examine each of these calls, bearing in mind that each function call is itself an expression that is evaluated as the value it returns. Again, you can think of it as if the function call was itself replaced by the returned value:

```
z = subtraction (7,2);
cout << "The first result is " << z;
```

If we replace the function call by the value it returns (i.e., 5), we would have:

```
z = 5;
cout << "The first result is " << z;
```

With the same procedure, we could interpret:

```
cout << "The second result is " << subtraction (7,2);
```

as:

```
cout << "The second result is " << 5;
```

since 5 is the value returned by subtraction (7,2).

In the case of:

```
cout << "The third result is " << subtraction (x,y);
```

The arguments passed to subtraction are variables instead of literals. That is also valid, and works fine. The function is called with the values x and y have at the moment of the call: 5 and 3 respectively, returning 2 as result.

The fourth call is again similar:

```
z = 4 + subtraction (x,y);
```

The only addition being that now the function call is also an operand of an addition operation. Again, the result is the same as if the function call was replaced by its result: 6. Note, that thanks to the commutative property of additions, the above can also be written as:

```
z = subtraction (x,y) + 4;
```

With exactly the same result. Note also that the semicolon does not necessarily go after the function call, but, as always, at the end of the whole statement. Again, the logic behind may be easily seen again by replacing the function calls by their returned value:

```
z = 4 + 2;    // same as z = 4 + subtraction (x,y);
z = 2 + 4;    // same as z = subtraction (x,y) + 4;
```

Example 3:

This example demonstrates a simple function whose purpose is to print a line of 45 asterisks. The example program generates a table, and lines of asterisks are used to make the table more readable.

```
// table.cpp
// demonstrates simple function
#include <iostream>
using namespace std;

void starline();                          //function declara
                                          //   (prototype)
int main()
   {
   starline();                            //call to functio
   cout << "Data type   Range" << endl;
   starline();                            //call to functio
   cout << "char        -128 to 127" << endl
        << "short       -32,768 to 32,767" << endl
        << "int         System dependent" << endl
        << "long        -2,147,483,648 to 2,147,483,647" <<
   starline();                            //call to functio
   return 0;
   }
//-----------------------------------------------------------
// starline()
// function definition
void starline()                           //function declara
   {
   for(int j=0; j<45; j++)                //function body
      cout << '*';
   cout << endl;
   }
```

Output :

```
*********************************************
Data type   Range
*********************************************
char        -128 to 127
short       -32,768 to 32,767
int         System dependent
long     -2,147,483,648 to 2,147,483,647
*********************************************
```

The program consists of two functions: main() and starline(). You’ve already seen many programs that use main() alone.

What other components are necessary to add a function to the program? There are three:

(1) The function declaration
(2) The calls to the function
(3) the function definition.

The Function Declaration:-

Just as you can’t use a variable without first telling the compiler what it is, you also can’t use a function without telling the compiler about it. There are two ways to do this. The approach we show here is to declare the function before it is called. (The other approach is to define it before it’s called; we’ll examine that next.) In the TABLE program, the function starline() is declared in the line

```
void starline();
```

- The declaration tells the compiler that at some later point we plan to present a function called starline.
- The keyword void specifies that the function has no return value
- The empty parentheses indicate that it takes no arguments.

Notice :-

- You can also use the keyword void in parentheses to indicate that the function takes no arguments, as is often done in C, but leaving them empty is the more common practice in C++.
- Notice that the function declaration is terminated with a semicolon. It is a complete statement in itself.

Function declarations are also called **prototypes**, since they provide a model or blueprint for the function. They tell the compiler, "a function that looks like this is coming up later in the program, so it’s all right if you see references to it before you see the function itself.” The information in the declaration (the return type and the number and types of any arguments) is also sometimes referred to as the **function signature**.

Calling the Function

The function is called (or invoked, or executed) three times from main(). Each of the three calls looks like this:

```
starline();
```

This is all we need to call the function:

The function name, **followed by parentheses**.

The syntax of the call is very similar to that of the declaration, except that the return type is not used. The call is terminated by a semicolon. Executing the call statement causes the function to execute; that is, control is transferred to the function, the statements in the function definition (which we'll examine in a moment) are executed, and then control returns to the statement following the function call.

The Function Definition:-

Finally we come to the function itself, which is referred to as the **function definition**. The definition contains the actual code for the function. Here's the definition for starline():

```
void starline()                  //declarator
   {
   for(int j=0; j<45; j++)       //function body
      cout << '*';
   cout << endl;
   }
```

The definition consists of:

1- a line called the declaratory
2- followed by the function body.

(The function body is composed of the statements that make up the function, delimited by braces**)**

The declarator must agree with the declaration: It must use the same function name, have the same argument types in the same order (if there are arguments), and have the same return type.

Notice that the declarator is not terminated by a semicolon. Figure 1.2 shows the syntax of the function declaration, function call, and function definition

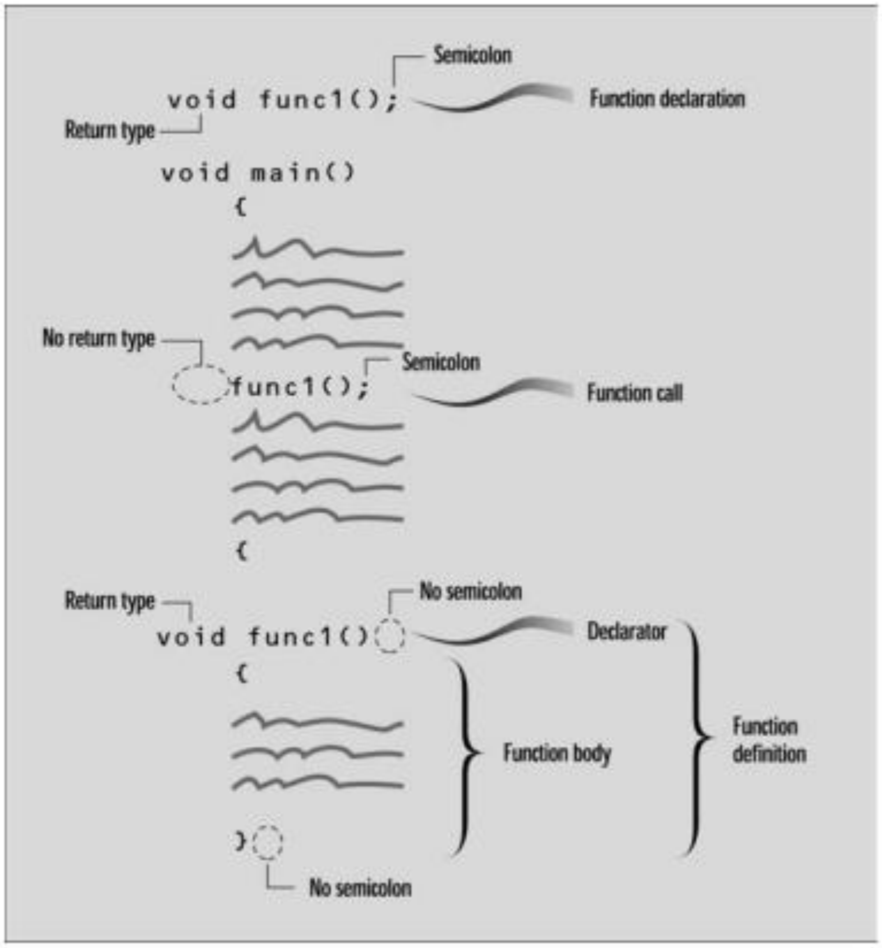
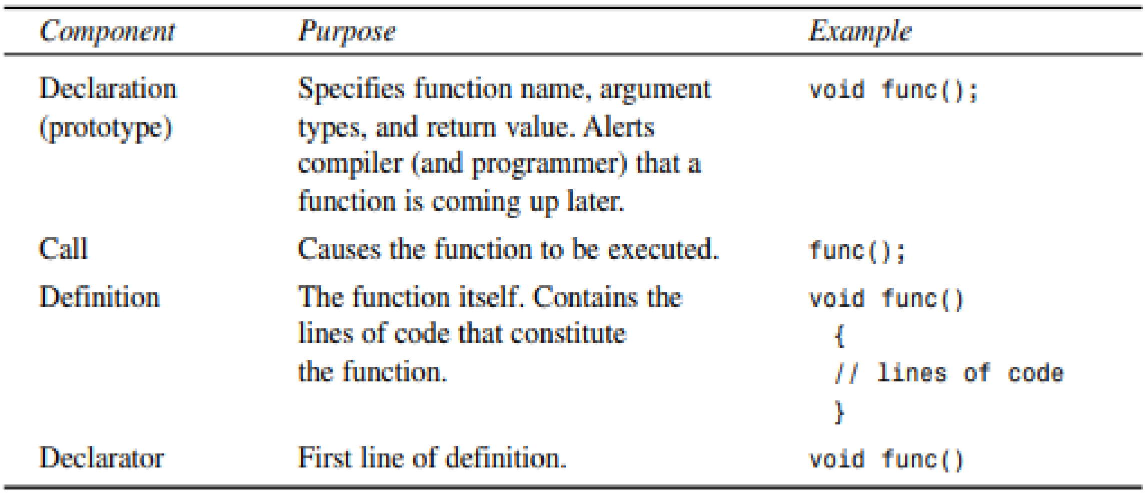


Figure 1.2 Function Syntax

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed, and when the closing brace is encountered, control returns to the calling program.

Table 1.1 summarizes the different function components.



Component	*Purpose*	*Example*
Declaration (prototype)	Specifies function name, argument types, and return value. Alerts compiler (and programmer) that a function is coming up later.	`void func();`
Call	Causes the function to be executed.	`func();`
Definition	The function itself. Contains the lines of code that constitute the function.	`void func() { // lines of code }`
Declarator	First line of definition.	`void func()`

Comparison with Library Functions:-

We've already seen some library functions in use. We have embedded calls to library functions, such as

```
ch = getche();
```

in our program code. Where are the declaration and definition for this library function? The declaration is in the header file specified at the beginning of the program (CONIO.H, for getche()). The definition (compiled into executable code) is in a library file that's linked automatically to your program when you build it.

When we use a library function we don't need to write the declaration or definition. But when we write our own functions, the declaration and definition are part of our source file, as we've shown in the TABLE example.

Library Function

C++ provides many built in functions that can be used in programs. However, the header file containing the function must be included before **int main()** using #include directive.

E.g. we must write statement **#include <iostream.h>** for using cin, cout statement in the program.

Following are the different library functions category wise:-

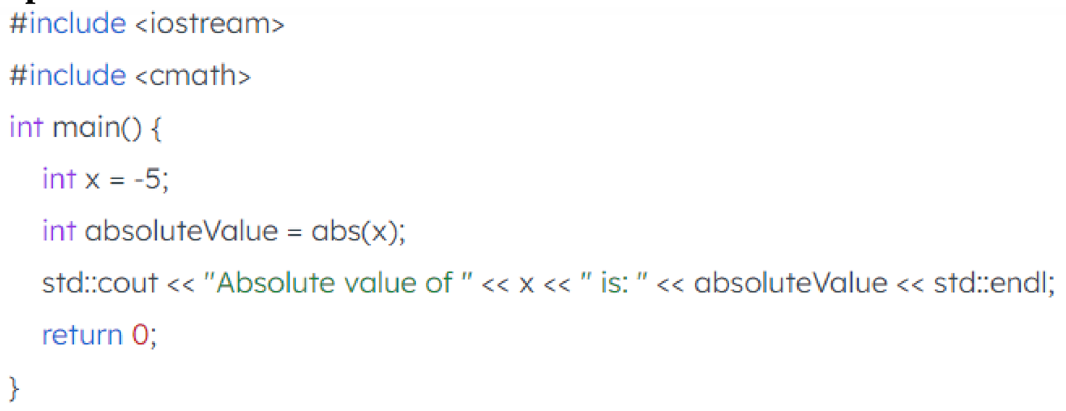
✓ Mathematical Function

Mathematical functions like sin( ), cos( ) are defined in header file math.h

Table 1.2 Some standard mathematical functions

Function	Meaning
sin(x)	This function returns the sine of **x** in radians.
cos(x)	This function returns the cosine of **x** in radians.
tan(x)	This function returns the tangent of **x** in radians.
asin(x)	This function returns the arcsine of **x** in radians.
acos(x)	This function returns the arccosine of **x** in radians.
exp(x)	This function multiples the value of **e** (2.71828) to the power of **x.**
log(x)	This function returns the natural logarithm of a number.
$\log_{10}$(x)	This function calculates the base-10 logarithm of a number.
sqrt(x)	This function returns the square root of a number
pow(x,y)	This function multiples **x** to the power of **y**.
abs(x)	This function returns the absolute value of a number.
floor(x)	This function returns the largest integer that's less than or equal to **x**.
ceil(x)	This function returns the smallest integer that's greater than or equal to **x**.

Example:-

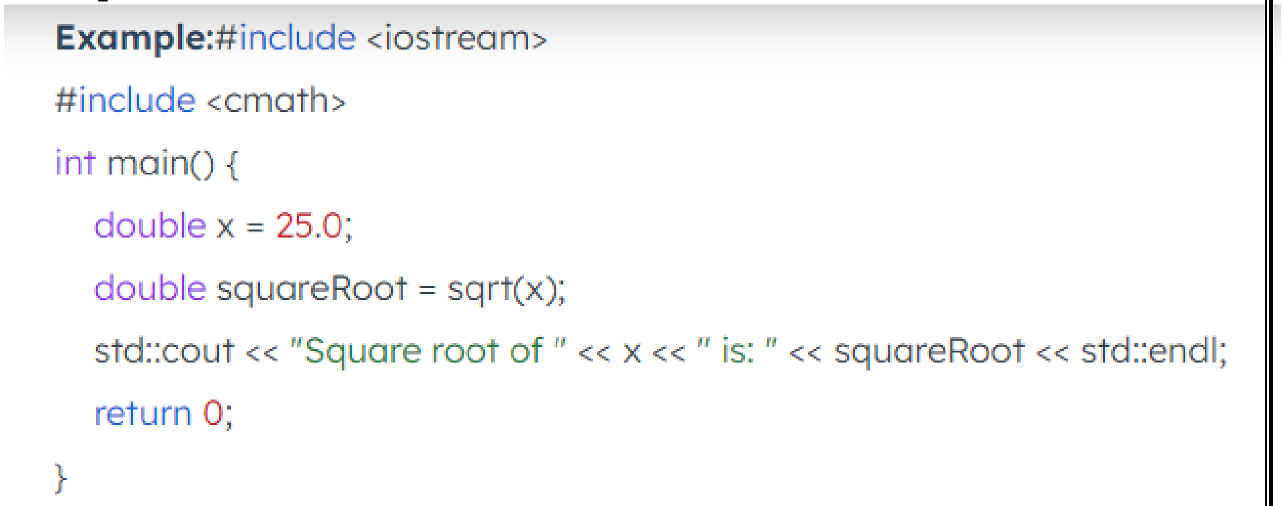


```
#include <iostream>
#include <cmath>
int main() {
   int x = -5;
   int absoluteValue = abs(x);
   std::cout << "Absolute value of " << x << " is: " << absoluteValue << std::endl;
   return 0;
}
```

Output:

```
Absolute value of -5 is: 5
```

Example:-



```
Example:#include <iostream>
#include <cmath>
int main() {
   double x = 25.0;
   double squareRoot = sqrt(x);
   std::cout << "Square root of " << x << " is: " << squareRoot << std::endl;
   return 0;
}
```

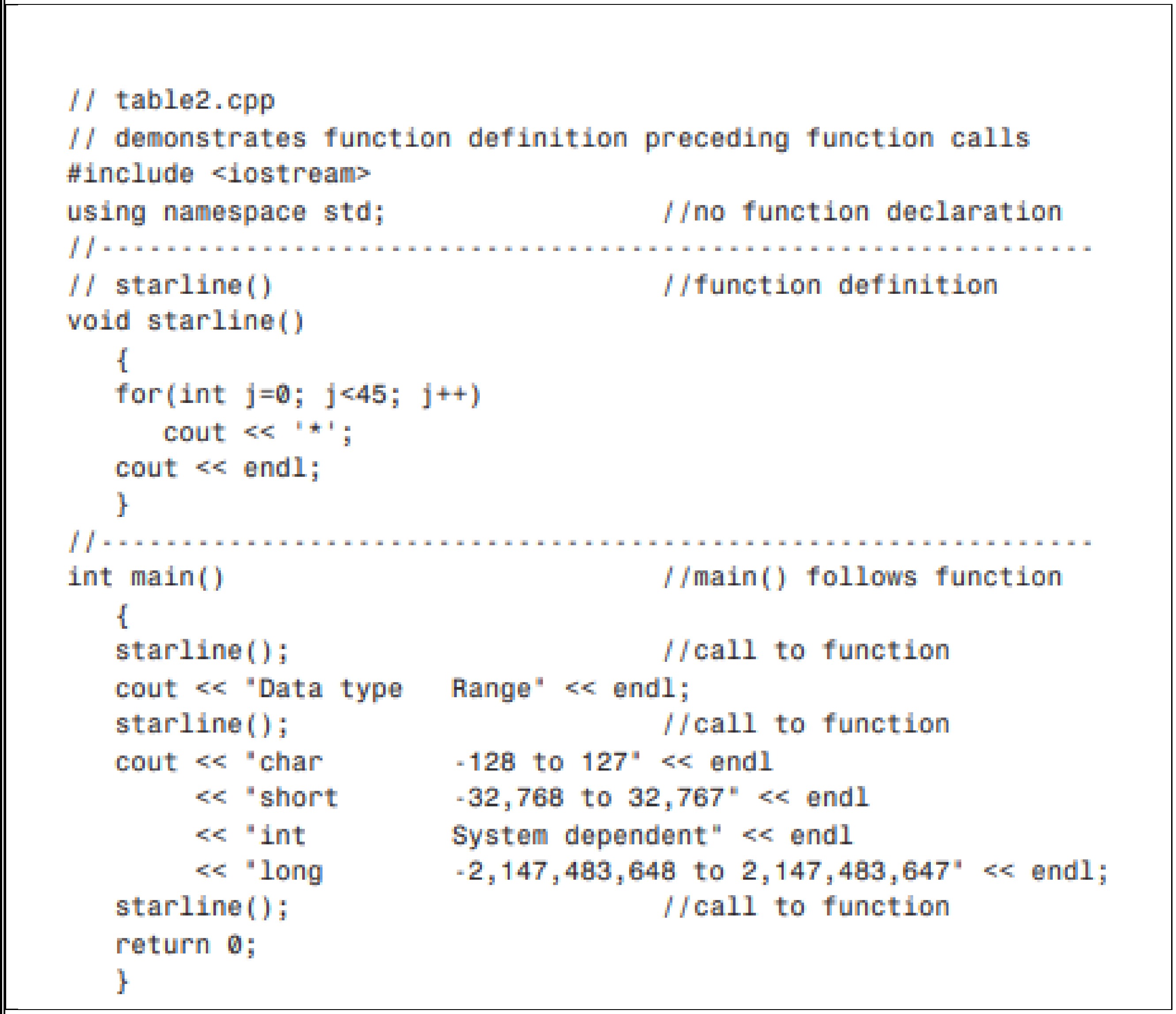
Output :

```
Square root of 25 is
```

Eliminating the Declaration:-

The second approach to inserting a function into a program is to eliminate the function declaration and place the function definition (the function itself) in the listing before the first call to the function. For example, we could rewrite

TABLE to produce table2.cpp, in which the definition for starline() appears first.



```
// table2.cpp
// demonstrates function definition preceding function calls
#include <iostream>
using namespace std;                  //no function declaration
//-------------------------------------------------------------
// starline()                         //function definition
void starline()
   {
   for(int j=0; j<45; j++)
      cout << '*';
   cout << endl;
   }
//-------------------------------------------------------------
int main()                            //main() follows function
   {
   starline();                        //call to function
   cout << "Data type   Range" << endl;
   starline();                        //call to function
   cout << "char        -128 to 127" << endl
        << "short       -32,768 to 32,767" << endl
        << "int         System dependent" << endl
        << "long        -2,147,483,648 to 2,147,483,647" << endl;
   starline();                        //call to function
   return 0;
   }
```

This approach is simpler for short programs, in that it removes the declaration, but it is less flexible. To use this technique when there **are more than a few functions**, the programmer must give considerable thought to arranging the functions so that each one appears before it is called by any other. Sometimes this is impossible. Also, many programmers prefer to place main() first in the listing, since it is where execution begins. In general we'll stick with the first approach, using declarations and starting the listing with main().

Functions with no type. The use of void

The syntax shown above for functions:

```
type name ( argument1, argument2 ...) { statements }
```

Requires the declaration to begin with a type. This is the type of the value returned by the function. But what if the function does not need to return a value? In this case, the type to be used is void, which is a special type to represent the absence of value. For example, a function that simply prints a message may not need to return any value:

```
// void function example
#include <iostream>
using namespace std;

void printmessage ()
{
  cout << "I'm a function!";
}

int main ()
{
  printmessage ();
}
```

Output :

```
I'm a function!
```

void can also be used in the function's parameter list to explicitly specify that the function takes no actual parameters when called. For example, printmessage could have been declared as:

```
void printmessage (void)
{
  cout << "I'm a function!";
}
```

In C++, an empty parameter list can be used instead of void with same meaning, but the use of void in the argument list was popularized by the C language, where this is a requirement.

Something that in no case is optional are the parentheses that follow the function name, neither in its declaration nor when calling it. And even when the function takes no parameters, at least an empty pair of parentheses shall always be appended to the function name. See how printmessage was called in an earlier example:

```
printmessage ();
```

The parentheses are what differentiate functions from other kinds of declarations or statements. The following would not call the function:

```
printmessage;
```

The return value of main

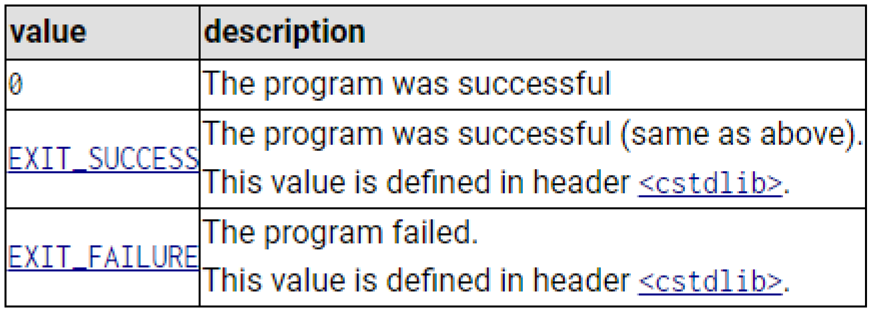
You may have noticed that the return type of main is int, but most examples in this and earlier chapters did not actually return any value from main.

Well, there is a catch: If the execution of main ends normally without encountering a return statement the compiler assumes the function ends with an implicit return statement:

```
return 0;
```

Note that this only applies to function main for historical reasons. All other functions with a return type shall end with a proper return statement that includes a return value, even if this is never used.

When main returns zero (either implicitly or explicitly), it is interpreted by the environment as that the program ended successfully. Other values may be returned by main, and some environments give access to that value to the caller in some way, although this behavior is not required nor necessarily portable between platforms. The values for main that are guaranteed to be interpreted in the same way on all platforms are:



value	description
0	The program was successful
EXIT_SUCCESS	The program was successful (same as above). This value is defined in header <cstdlib>.
EXIT_FAILURE	The program failed. This value is defined in header <cstdlib>.

Because the implicit return 0; statement for main is a tricky exception, some authors consider it good practice to explicitly write the statement.

1.2 Passing Arguments to Functions

An argument is a piece of data (an int value, for example) passed from a program to the function. Arguments allow a function to operate with different values, or even to do different things, depending on the requirements of the program calling it.

Passing Constants:

As an example, let’s suppose we decide that the starline() function in the last example is too rigid. Instead of a function that always prints 45 asterisks, we want a function that will print any character any number of times.

Here’s a program, TABLEARG, that incorporates just such a function. We use arguments to pass the character to be printed and the number of times to print it.

```
// tablearg.cpp
// demonstrates function arguments
#include <iostream>
using namespace std;
void repchar(char, int);                //function declaration

int main()
   {
   repchar('-', 43);                    //call to function
   cout << "Data type   Range" << endl;
   repchar('=', 23);                    //call to function
   cout << "char        -128 to 127" << endl
        << "short       -32,768 to 32,767" << endl
        << "int         System dependent" << endl
        << "double      -2,147,483,648 to 2,147,483,647" << endl;
   repchar('-', 43);                    //call to function
   return 0;
   }
//--------------------------------------------------------------
// repchar()
// function definition
void repchar(char ch, int n)            //function declarator
   {
   for(int j=0; j<n; j++)               //function body
      cout << ch;
   cout << endl;
   }
```

The new function is called repchar(). Its declaration looks like this:

void repchar(char, int); // declaration specifies data types

The items in the parentheses are the data types of the arguments that will be sent to repchar(): char and int.

In a function call, specific values—constants in this case—are inserted in the appropriate place in the parentheses:

repchar(‘-’, 43); // function call specifies actual values

This statement instructs repchar() to print a line of 43 dashes. The values supplied in the call must be of the types specified in the declaration: the first argument, the - character, must be of type char; and the second argument, the number 43, must be of type int. The types in the declaration and the definition must also agree.

The next call to repchar()

repchar('=', 23);

tells it to print a line of 23 equal signs. The third call again prints 43 dashes. Here's the output from TABLEARG:

```
-------------------------------------------
Data type   Range
=======================
char        -128 to 127
short       -32,768 to 32,767
int         System dependent
long         -2,147,483,648 to 2,147,483,647
-------------------------------------------
```

The calling program supplies arguments, such as '–' and 43, to the function. The variables used within the function to hold the argument values are called parameters; in repchar() they are ch and n. (We should note that many programmers use the terms argument and parameter somewhat interchangeably.) The declarator in the function definition specifies both the data types and the names of the parameters:

```
void repchar(char ch, int n)  //declarator specifies parameter
                              //names and data types
```

These parameter names, ch and n, are used in the function as if they were normal variables.

Placing them in the declarator is equivalent to defining them with statements like

char ch;

int n;

When the function is called, its parameters are automatically initialized to the values passed by the calling program.

Overloaded Functions

This means that same function name can be used to create functions that perform a variety of different tasks. This is known as function polymorphism in OOP. The function would perform different operations depending on the argument list in function call. The correct function to be invoked is list in function call. The correct function to be invoked is determined by checking the number and the type of arguments.

For example, an overloaded function area( ) handles different types of data as follows:-

```
   // Declaration

    void area (int , int);                    // prototype 1
    void area (float);                        // prototype 2
    void area (float , int);                  // prototype 3

 // Function calls
      cout<< area(5);                     // uses prototype 1
     cout<< area(5.2.6);                      // uses prototype 3
     cout<< area(7,6.2);                      // uses prototype 2
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique.

The function selection involves the following steps:

1. The compiler tries to find an exact match in which types of actual arguments are the same and use that function.
2. If an exact match is not found , the compiler uses integral promotions to the actual arguments , such as chat to int, float to double to find a match.
3. When either of them fails , the compiler tries to use the built in conversions(the implicit conversions) to the actual arguments then uses the function whose match is unique. If the conversion is possible

to have multi matches, then the compiler will generate an error message.

Suppose , we use the following two functions :-

```
long abc(long n);

Double abc (double x);
```

A function call such as

abc(10)

Will cause an error because int argument can be converted to either long or double, thereby creating an ambiguous situation.

If all the steps fail, then the compiler will try the user defined conversions in combination with integral promotions and built in conversions to find a unique match.

Example : the following program illustrate function overloading

Overloading Using Different Number of Parameters

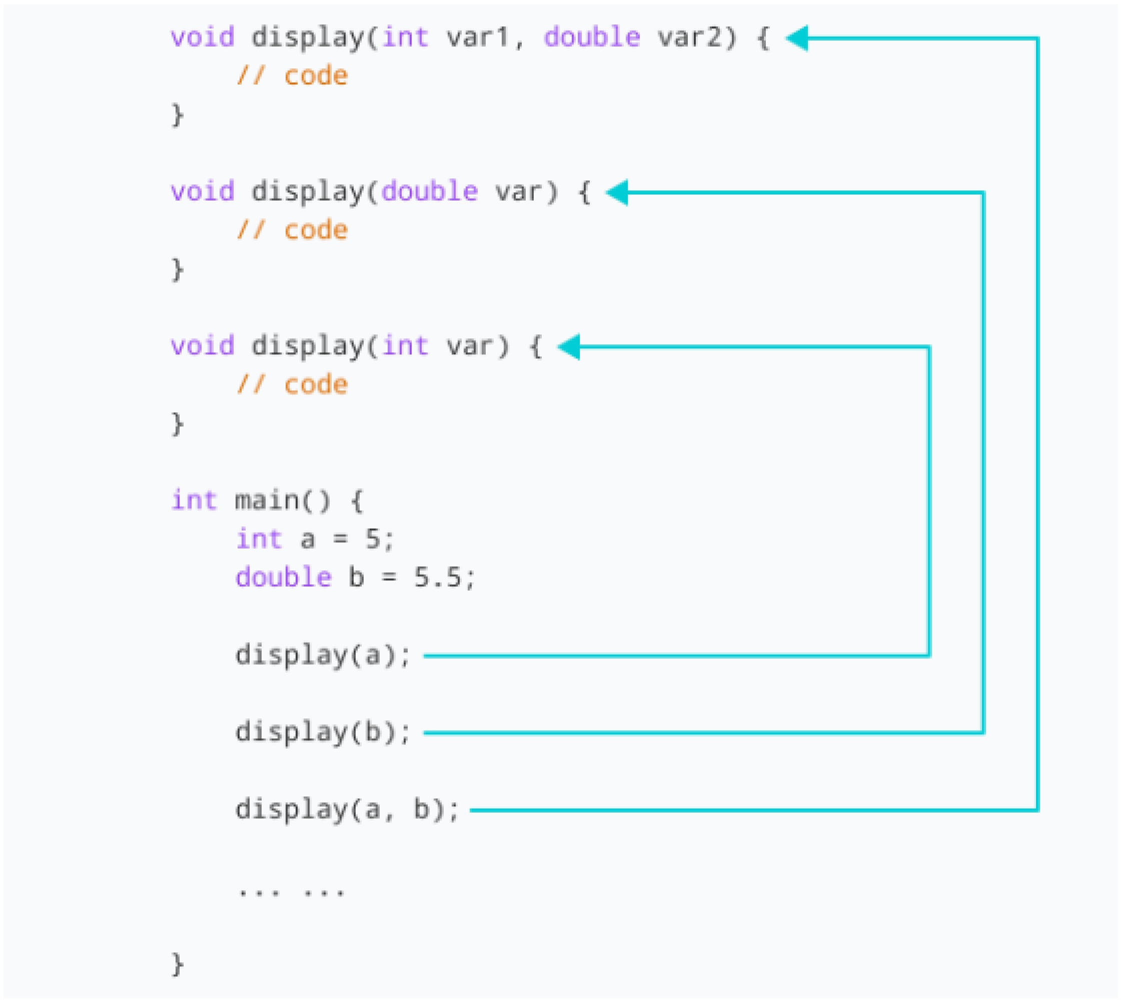
```
#include <iostream>
using namespace std;

// function with 2 parameters
void display(int var1, double var2) {
   cout << "Integer number: " << var1;
   cout << " and double number: " << var2 << endl;
}

// function with double type single parameter
void display(double var) {
   cout << "Double number: " << var << endl;
}

// function with int type single parameter
void display(int var) {
   cout << "Integer number: " << var << endl;
}
```

```
int main() {

    int a = 5;
    double b = 5.5;

    // call function with int type parameter
    display(a);

    // call function with double type parameter
    display(b);

    // call function with 2 parameters
    display(a, b);

    return 0;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and double number: 5.5
```

Here, the display() function is called three times with different arguments. Depending on the number and type of arguments passed, the corresponding display() function is called.



```
void display(int var1, double var2) {
    // code
}

void display(double var) {
    // code
}

void display(int var) {
    // code
}

int main() {
    int a = 5;
    double b = 5.5;

    display(a);

    display(b);

    display(a, b);

    ... ...

}
```

Recursive Function

Recursively is the property that functions have to be called by themselves. It is useful for some tasks, such as sorting elements, or calculating the factorial of numbers. For example, in order to obtain the factorial of a number (n!) the mathematical formula would be:

```
n! = n * (n-1) * (n-2) * (n-3) ... * 1
```

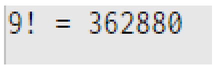
More concretely, 5! (factorial of 5) would be:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

And a recursive function to calculate this in C++ could be:

```cpp
// factorial calculator
#include <iostream>
using namespace std;

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return 1;
}

int main ()
{
  long number = 9;
  cout << number << "! = " << factorial (number);
  return 0;
}
```

Output :



```
9! = 362880
```

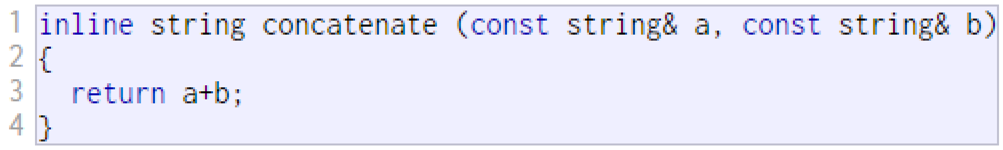
Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since, otherwise, the function would perform an infinite recursive loop, in which once it arrived to 0, it would continue multiplying by all the negative numbers (probably provoking a stack overflow at some point during runtime).

Inline Functions

Calling a function generally causes a certain overhead (stacking arguments, jumps, etc...), and thus for very short functions, it may be more efficient to simply insert the code of the function where it is called, instead of performing the process of formally calling a function.

Preceding a function declaration with the inline specifier informs the compiler that inline expansion is preferred over the usual function call mechanism for a specific function. This does not change at all the behavior of a function, but is merely used to suggest the compiler that the code generated by the function body shall be inserted at each point the function is called, instead of being invoked with a regular function call.

For example, the concatenate function above may be declared inline as:



```
inline string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

This informs the compiler that when concatenate is called, the program prefers the function to be expanded inline, instead of performing a regular call. inline is only specified in the function declaration, not when it is called.

Note that most compilers already optimize code to generate inline functions when they see an opportunity to improve efficiency, even if not explicitly marked with the inline specifier. Therefore, this specifier merely indicates the compiler that inline is preferred for this function, although the compiler is free to not inline it, and optimize otherwise. In C++, optimization is a task delegated to the compiler, which is free to generate any code for as long as the resulting behavior is the one specified by the code.

Review Questions

1. A function's single most important role is to

   a. give a name to a block of code.

   b. reduce program size.

   c. accept arguments and provide a return value.

   d. help organize a program into conceptual units.

2. A function itself is called the function d_________.

3. Write a function called foo() that displays the word foo.

4. A one-statement description of a function is referred to as a function d_________ or a p_________.

5. The statements that carry out the work of the function constitute the function _________.

6. A program statement that invokes a function is a function _________.

7. The first line of a function definition is referred to as the _________.

8. A function argument is

    a. a variable in the function that receives a value from the calling program.

    b. a way that functions resist accepting the calling program's values.

    c. a value sent to the function by the calling program.

    d. a value returned by the function to the calling program.

9. True or false: When arguments are passed by value, the function works with the original arguments in the calling program.

10. What is the purpose of using argument names in a function declaration?

11. Which of the following can legitimately be passed to a function?

    a. A constant

    b. A variable

    c. A structure

    d. A header file

12. What is the significance of empty parentheses in a function declaration?

13. How many values can be returned from a function?

14. True or false: When a function returns a value, the entire function call can appear on the right side of the equal sign and be assigned to another variable.

15. Where is a function's return type specified**?**

Exercises:-

(1) Write a function that takes two Distance values as arguments and returns the larger one.
Include a main() program that accepts two Distance values from the user, compares them, and displays the larger.

(2) Write a function called swap() that interchanges two int values passed to it by the calling program. (Note that this function swaps the values of the variables in the calling program, not those in the function.) You'll need to decide how to pass the arguments. Create a main() program to exercise the function.

(3) Write a C++ program to implement the following function :-
(a) To accept two integers x and y and return x raised to y.
(b) To accept a number and check if it is a prime number .
(c) To accept a number and return the sum of its digits.
(d) To accept a number and return reversed number.
(e) To accept a capital letter and convert it into smaller letter.

Chapter 2
Object and Classes

Introduction

Central to C++ is object-oriented programming (OOP). OOP was the impetus for the creation of C++ Because of this it is useful to understand OOP’s basic principles before you write even a simple C++ program.

Object-oriented programming took the best ideas of structured programming and combined them with several new concepts. The result was a different and better way of organizing a program. In the most general sense, a program can be organized in one of two ways: around its code (what is happening) or around its data (who is being affected). Using only structured programming techniques, programs are typically organized around code. This approach can be thought of as “code acting on data ”.

Objects and classes are used to wrap related functions and data in one place in C++. Suppose we need to store the length, breadth, and height of a rectangular room and calculate its area and volume.

To handle this task, we can create three variables, say, length, breadth, and height, along with the functions calculate_area() and calculate_volume().

However, in C++, rather than creating separate variables and functions, we can also wrap the related data and functions in a single place (by creating **objects**).

This programming paradigm is known as object-oriented programming.

Object-oriented programs work the other way around. They are organized around data, with the key principle being "data controlling access to code " In an object -oriented language; you define the data and the routines that are permitted to act on that data. Thus, a data type defines precisely what sort of operations can be applied to that data. To support the principles of object-oriented programming, all OOP languages, including C++, have three traits in common encapsulation, polymorphism and inheritance. Let's examine each.

ENCAPSULATION: is a programming mechanism that binds together code and the data it manipulates, and that keeps both safe from outside interference and misuse. In an object-oriented language, code and data can be bound together in such a way that a self-contained black box is created. Within the box are all necessary data and code. When code and data are linked together in this fashion, an object is created. In other words, an object is the device that supports encapsulation.

POLYMORPHISM: (from Greek meaning "many forms") is the quality that allows one interface to access a general class of actions. A simple example of polymorphism is found in the steering wheel of an automobile. The steering wheel (the interface) is the same no matter what type of actual steering mechanism is used. That is, the steering wheel works the same whether your car has manual steering, power steering, or rack-and-pinion steering. Thus, turning the steering wheel left causes the car to go left no matter what type of steering is used. The benefit of the uniform interface is, of course, that once you know how to operate the steering wheel, you can drive any type of car.

The same principle can also apply to programming. For example, consider a stack (which is a first-in, last- out list). You might have a program that requires three different types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. In this case, the algorithm that implements each stack is the same, even though the data being stored differs. In a non–object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in C++ you can create one general set of stack routines that works for all three situations. This way, once you know how to use one stack, you can use them all.

INHERITANCE: is the process by which one object can acquire the properties of another object. This is important because it supports the concept of hierarchical classification. If you think about it, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Red Delicious apple is part of the classification apple, which in turn is part of the fruit class, which is under the larger class food. That is, the food class possesses certain qualities (edible, nutritious, and so on) which also, logically, apply to its subclass, fruit. In addition to these qualities, the fruit class has specific characteristics (juicy, sweet, and so on) that distinguish it from other food. The apple class defines those qualities specific to an apple (grows on trees, not tropical, and so on). A Red Delicious apple would, in turn, inherit all the qualities of all preceding classes and would define only those qualities that make it unique.

Without the use of hierarchies, each object would have to explicitly define all of its characteristics. Using inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

CHARACTERISTICS OF OOPS

OOP offers several benefits to both the program designer and the user. The principal advantages are.

1) Emphasis is on data rather than procedure.

2) Programs are divided into what are known as objects.

3) Data structures are designed such that they characterize the objects.

4) Functions that operate on the data of an object are tied together in the data structure.

5) Data is hidden and cannot be accessed by external functions.

6) Objects may communicate with each other through functions.

7) New data and functions can be easily added wherever necessary.

8) Follows bottom up approach in program design.

9) Through inheritance, we can eliminate redundant code and extend the use of existing classes

10) We can build program from the standard working module that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.

11) The principal of data hiding helps the programmer to build secure programs that cannot be invaded by code in other part of the program.

12) It is possible to have multiple instance of an object to co-exist without any interference

13) It is easy to partition the work in a project, based on objects.

14) Object oriented systems can be easily upgraded from small to large systems.

15) Message passing techniques for communication between objects makes the interface description with external systems much simpler.

16) Software complexity can be easily managed.

2.2 A Simple Class

Classes are an expanded concept of *data structures*: like data structures, they can contain data members, but they can also contain functions as members.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

A class is defined in C++ using the keyword class followed by the name of the class.

The body of the class is defined inside curly brackets and terminated by a semicolon at the end.

```
class ClassName {
  // some data
  // some functions
};
```

For example,

```
class Room {
   public:
       double length;
       double breadth;
       double height;

       double calculate_area(){
           return length * breadth;
       }

       double calculate_volume(){
           return length * breadth * height;
       }

};
```

Here, we defined a class named Room.

The variables length, breadth, and height declared inside the class are known as **data members**.

And the functions calculate_area() and calculate_volume () are known as **member functions** of a class.

Classes have the same format as plain *data structures*, except that they can also include functions and have these new things called *access specifiers*. An *access specifier* is one of the following three keywords: private, public or protected. These specifiers modify the access rights for the members that follow them:
private members of a class are accessible only from within other members of the same class (or from their *"friends"*).

- protected members are accessible from other members of the same class (or from their *"friends"*), but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before any other *access specifier* has private access automatically. For example:

```
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area (void);
} rect;
```

Declares a class (i.e., a type) called Rectangle and an object (i.e., a variable) of this class, called rect. This class contains four members: two data members of type int (member width and member height) with *private access* (because private is the default access level) and two member functions with *public access*: the functions set_values and area, of which for now we have only included their declaration, but not their definition.
Notice the difference between the *class name* and the *object name*: In the previous example, Rectangle was the *class name* (i.e., the type), whereas rect was an object of type Rectangle. It is the same relationship int and a have in the following declaration:

```
int a;
```

where int is the type name (the class) and a is the variable name (the object).
After the declarations of Rectangle and rect, any of the public members of object rect can be accessed as if they were normal functions or normal variables, by simply inserting a dot (.) between *object name* and *member name*. This follows the same syntax as accessing the members of plain data structures. For example:

```
rect.set_values (3,4);
myarea = rect.area();
```

The only members of rect that cannot be accessed from outside the class are width and height, since they have private access and they can only be referred to from within other members of that same class.

Here is the complete example of class Rectangle:

```
// classes example
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

C++ Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated.

To use the data and access functions defined in the class, we need to create objects.

Syntax to Define Object in C++

```
ClassName object_name;
```

We can create objects of Room class as follows:

```
/ sample function
void sample_function() {
   // create objects
   Room room1, room2;
}

int main(){
   // create objects
   Room room3, room4;
}
```

Here, two objects room1 and room2 of the Room class are created in sample_function().

Similarly, the objects room3 and room4 are created in main().

As we can see, we can create objects of a class in any function of the program.

We can also create objects of a class within the class itself or in other classes.

Also, we can create as many objects as we want from a single class.

Example:

The following program contains a class and two objects of that class. Although it's simple, the program demonstrates the syntax and general features of classes in C++. Here's the listing for the SMALLOBJ program:

```
// smallobj.cpp
// demonstrates a small, simple object
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class smallobj               //define a class
   {
   private:
      int somedata;          //class data
   public:
      void setdata(int d)    //member function to set data
         { somedata = d; }
      void showdata()        //member function to display data
         { cout << "Data is " << somedata << endl; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   smallobj s1, s2;   //define two objects of class smallobj

   s1.setdata(1066);  //call member function to set data
   s2.setdata(1776);

   s1.showdata();     //call member function to display data
   s2.showdata();
   return 0;
   }
```

The class **smallobj** defined in this program contains one data item and two member functions. The two member functions provide the only access to the data item from outside the class. The first member function sets the data item to a value, and the second displays the value.

Placing data and functions together into a single entity is a central idea in object-oriented programming. This is shown in Figure 2.1.

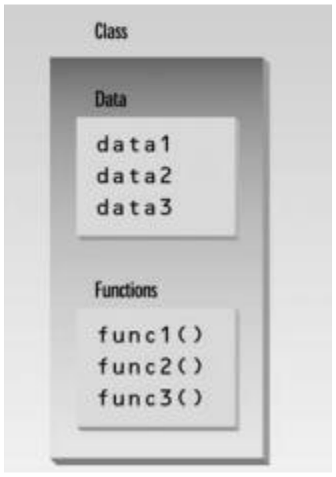


Figure 2.1 Classes contain Data and functions

Classes and Objects

Recall from fundamental of C++ that an object has the same relationship to a class that a variable has to a data type. An object is said to be an instance of a class. In SMALLOBJ, the class—whose name is smallobj—is defined in the first part of the program. Later, in main(), we define two objects—s1 and s2—that are instances of that class.

Each of the two objects is given a value, and each displays its value. Here's the output of the program:

```
Data is 1066     ←——— object s1 displayed this
Data is 1776     ←——— object s2 displayed this
```

Defining the Class:-

Here's the definition (sometimes called ***a specifier***) for the class smallobj, copied from the SMALLOBJ listing:

```
class smallobj                //define a class
   {
   private:
      int somedata;           //class data
   public:
      void setdata(int d)     //member function to set data
         { somedata = d; }
      void showdata()         //member function to display data
         { cout << "\nData is " << somedata; }
   };
```

The definition starts with the keyword class, followed by the class name—smallobj in this example. Like a structure, the body of the class is delimited by braces and terminated by a semicolon.

Note :-

Don't forget the semicolon. Remember, data constructs such as structures and classes end with a semicolon, while control constructs such as functions and loops do not.

private and public:-

The body of the class contains two unfamiliar keywords: private and public. What is their purpose?

A key feature of object-oriented programming is data hiding. This term does not refer to the activities of particularly paranoid programmers; rather it means that data is concealed within a class so that it cannot be accessed mistakenly by functions outside the class. The primary mechanism for hiding data is to put it in a class and make it private. Private data or functions can only be accessed from within the class. Public data or functions, on the other hand, are accessible from outside the class. This is shown in Figure 2.2.

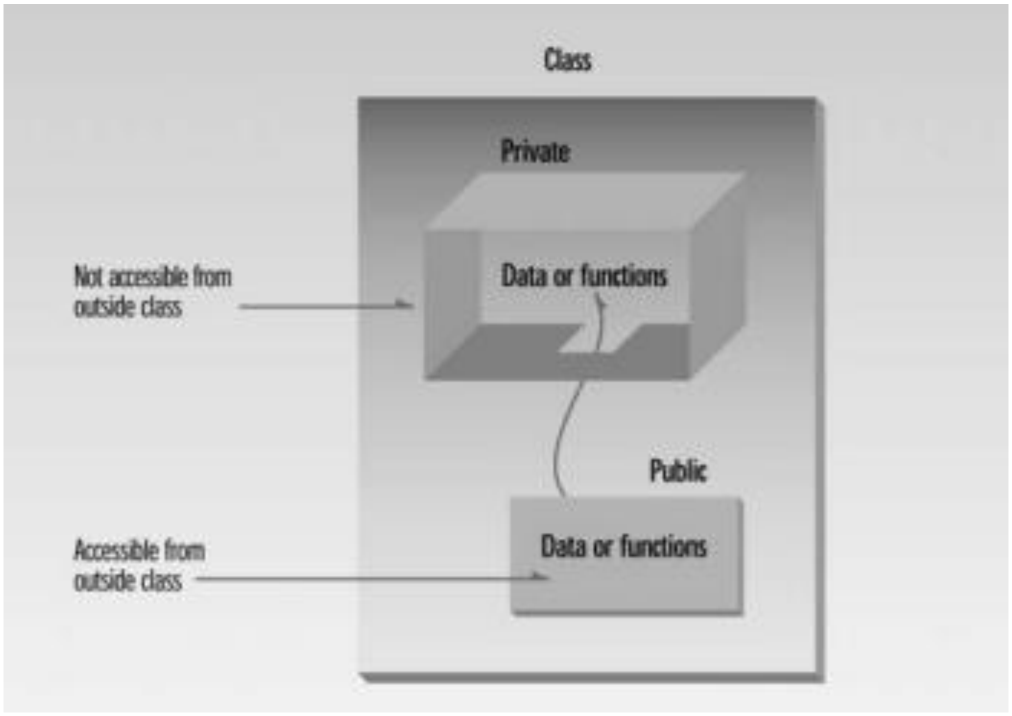

Class
Private
Data or functions
Not accessible from outside class
Public
Data or functions
Accessible from outside class

Figure 2.2 private and public

Class Data:-

The smallobj class contains one data item: somedata, which is of type int. The data items within a class are called **data members** (or sometimes member data). There can be any number of data members in a class, just as there can be any number of data items in a structure. The data member somedata follows the keyword private, so it can be accessed from within the class, but not from outside.

Member Functions:-

Member functions are functions that are included within a class. There are two member functions in smallobj: setdata() and showdata().

The function bodies of these functions have been written on the same line as the braces that delimit them. You could also use the more traditional format for these function definitions:

```
void setdata(int d)
    {
    somedata = d;
    }
```

and

```
void showdata()
   {
   cout << "\nData is " << somedata;
   }
```

Because setdata() and showdata() follow the keyword public, they can be accessed from outside the class.. Figure 2.3 shows the syntax of a class definition.

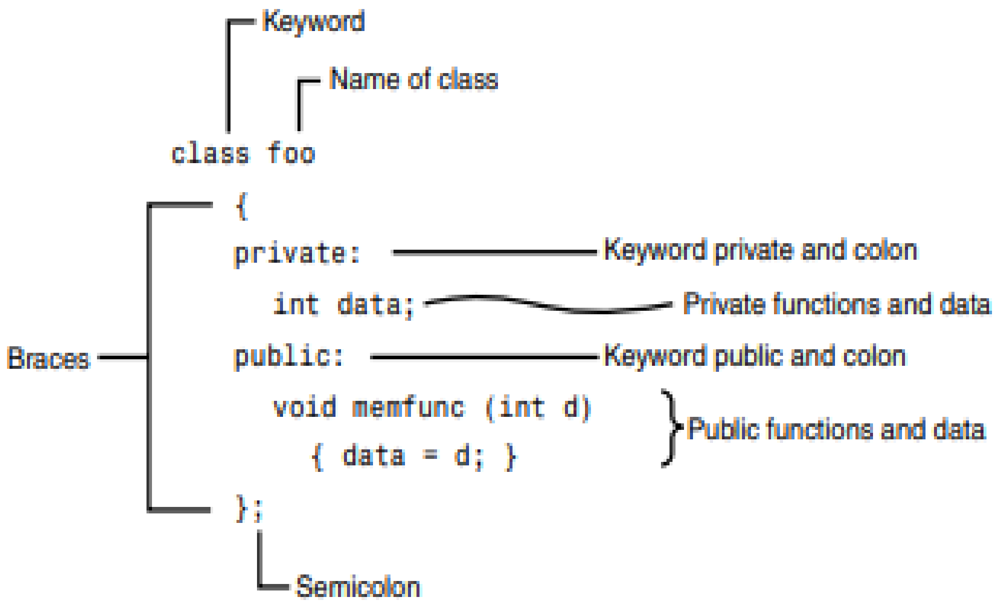


Figure 2.3 Syntax of Class Definition

Functions Are Public, Data Is Private

Usually the data within a class is private and the functions are public. This is a result of the way classes are used. The data is hidden so it will be safe from accidental manipulation, while the functions that operate on the data are public so they can be accessed from outside the class. However, **there is no rule that says data must be private and functions public;** in some circumstances you may find you'll need to use private functions and public data.

Member Functions Within Class Definition:-

The member functions in the smallobj class perform operations that are quite common in classes: setting and retrieving the data stored in the class.

The **setdata()** function accepts a value as a parameter and sets the somedata variable to this value.

The **showdata()** function displays the value stored in somedata.

Note :

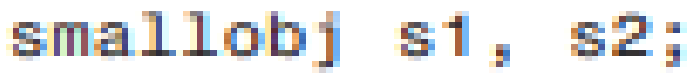
Note that the member functions setdata() and showdata() are definitions in that the actual code for the function is contained within the class definition. (The functions are not definitions in the sense that memory is set aside for the function code; this doesn't happen until an object of the class is created.) Member functions defined inside a class this way are created as inline functions by default. We'll see later that it is also possible to declare a function within a class but to define it elsewhere. Functions defined outside the class are not normally inline.

Using the Class:-

Now that the class is defined, let's see how main() makes use of it. We'll see how objects are defined, and, once defined, how their member functions are accessed.

Defining Objects:-

The first statement in main()



```
smallobj s1, s2;
```

defines two objects, s1 and s2, of class smallobj.

Remember that the definition of the class smallobj does not create any objects. It only describes how they will look when they are created, just as a structure definition describes how a structure will look but doesn't create any structure variables. It is objects that participate in program operations. Defining an object is similar to defining a variable of any data type: Space is set aside for it in memory**.**

Defining objects in this way means creating them. This is also called **instantiating** them. The term instantiating arises because an instance of the class is created. An object is an instance (that is, a specific example) of a class. Objects are sometimes called **instance variables.**

Calling Member Functions:-

The next two statements in main() call the member function setdata():

```
s1.setdata(1066);
s2.setdata(1776);
```

These statements don't look like normal function calls.

Why are the object names s1 and s2 connected to the function names with a period? This strange syntax is used to call a member function that is associated with a specific object. Because setdata() is a member function of the smallobj class, it must always be called in connection with an object of this class. It doesn't make sense to say

```
setdata(1066);
```

by itself, because a member function is always called to act on a specific object, not on the class in general. Attempting to access the class this way would be like trying to drive the blueprint of a car. Not only does this statement not make sense, but the compiler will issue an error message if you attempt it. Member functions of a class can be accessed only by an object of that class.

To use a member function, the dot operator (the period) connects the object name and the member function. The syntax is similar to the way we refer to structure members, but the parentheses signal that we're executing a member function rather than referring to a data item. (The dot operator is also called the class member access operator.)

The first call to setdata()

```
s1.setdata(1066);
```

executes the setdata() member function of the s1 object. This function sets the variable somedata in object s1 to the value 1066. The second call

```
s2.setdata(1776);
```

causes the variable somedata in s2 to be set to 1776. Now we have two objects whose somedata variables have different values, as shown in Figure 2.4.

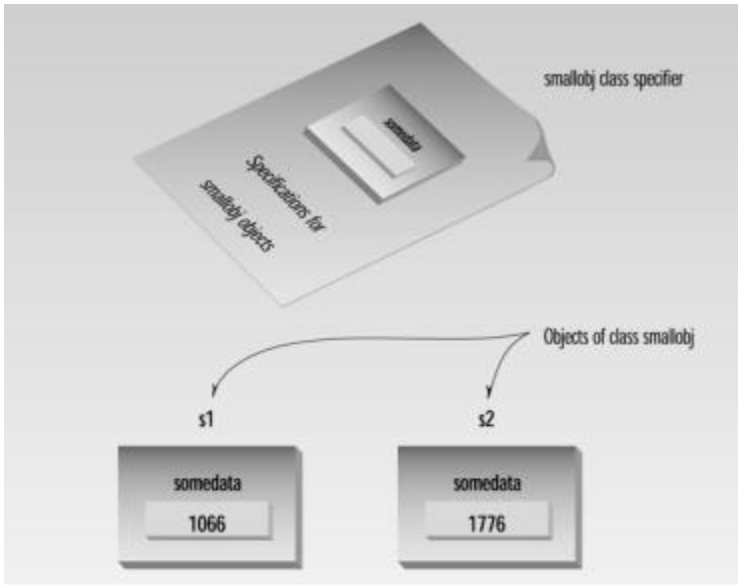

smallobj class specifier
somedata
Specifications for smallobj objects
Objects of class smallobj
s1
s2
somedata
1066
somedata
1776

Figure 2.4 Two objects of class smallobj.

Similarly, the following two calls to the showdata() function will cause the two objects to display their values:

```
s1.showdata();
s2.showdata();
```

Messages:-

Some object-oriented languages refer to calls to member functions as messages. Thus the call

```
s1.showdata();
```

can be thought of as sending a message to s1 telling it to show its data. The term message is not a formal term in C++, but it is a useful idea to keep in mind as we discuss member functions. Talking about messages emphasizes that objects are discrete entities and that we communicate with them by calling their member functions.

Example :- Object and Class in C++ Programming

```
// Program to illustrate the working of
// objects and class in C++ Programming

#include <iostream>
using namespace std;

// create a class
class Room {

   public:
    double length;
    double breadth;
    double height;

    double calculate_area() {
        return length * breadth;
    }

    double calculate_volume() {
        return length * breadth * height;
    }
};
```

```
int main() {

   // create object of Room class
   Room room1;

   // assign values to data members
   room1.length = 42.5;
   room1.breadth = 30.8;
   room1.height = 19.2;

   // calculate and display the area and volume of the room
   cout << "Area of Room =  " << room1.calculate_area() << endl;
   cout << "Volume of Room =  " << room1.calculate_volume() <<
endl;

   return 0;
}
```

Output

```
Area of Room =  1309
Volume of Room =  25132.8
```

In this program, we have used the Room class and its object room1 to calculate the area and volume of a room.

In main(), we assigned the values of length, breadth, and height with the code:

```
room1.length = 42.5;
room1.breadth = 30.8;
room1.height = 19.2;
```

We then called the functions calculate_area() and calculate_volume() to perform the necessary calculations.

Note the use of the keyword public in the program. This means the members are public and can be accessed anywhere from the program**.**

C++ Access Modifiers

One of the main features of object-oriented programming languages such as C++ is data hiding.

Data hiding refers to restricting access to data members of a class. This is to prevent other functions and classes from tampering with the class data.

However, it is also important to make some member functions and member data accessible so that the hidden data can be manipulated indirectly.

The access modifiers of C++ allows us to determine which class members are accessible to other classes and functions, and which are not.

Example:

```
class Patient {

    private:
        int patientNumber;
        string diagnosis;

    public:

      void billing() {
          // code
      }

      void makeAppointment() {
          // code
      }
};
```

Here, the variables patientNumber and diagnosis of the Patient class are hidden using the private keyword, while the member functions are made accessible using the public keyword.

Types of C++ Access Modifiers

In C++, there are 3 access modifiers:

(1) public
(2) private

(3) protected

public Access Modifier

- The public keyword is used to create public members (data and functions).
- The public members are accessible from any part of the program.

Example 1: C++ public Access Modifier

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

// define a class
class Sample {

    // public elements
   public:
    int age;

    void displayAge() {
        cout << "Age = " << age << endl;
    }
};

int main() {

    // declare a class object
    Sample obj1;

    cout << "Enter your age: ";

    // store input in age of the obj1 object
    cin >> obj1.age;

    // call class function
    obj1.displayAge();

    return 0;
}
```

Output

```
Enter your age: 20
Age = 20
```

In this program, we have created a class named Sample, which contains a public variable age and a public function displayAge().

In main(), we have created an object of the Sample class named obj1. We then access the public elements directly by using the codes obj1.age and obj1.displayAge().

Notice that the public elements are accessible from main(). This is because public elements are accessible from all parts of the program.

private Access Modifier

- **The private keyword is used to create private members (data and functions).**
- The private members can only be accessed from within the class.
- However, friend classes and friend functions can access private members.

Example : C++ private Access Specifier

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

// define a class
class Sample {

   // private elements
  private:
   int age;

   // public elements
  public:
   void displayAge(int a) {
       age = a;
       cout << "Age = " << age << endl;
   }
};

int main() {

   int ageInput;

   // declare an object
   Sample obj1;

   cout << "Enter your age: ";
   cin >> ageInput;

   // call function and pass ageInput as argument
   obj1.displayAge(ageInput);

   return 0;
}
```

Output

```
Enter your age: 20
Age = 20
```

In main(), the object obj1 cannot directly access the class variable age.

```
// error
cin >> obj1.age;
```

We can only indirectly manipulate age through the public function displayAge(), since this function initializes age with the value of the argument passed to it i.e. the function parameter int a.

protected Access Modifier

Before we learn about the protected access specifier, make sure you know about inheritance in C++.

- **The protected keyword is used to create protected members (data and function).**
- The protected members can be accessed within the class and from the derived class.

Example: C++ protected Access Specifier

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

// declare parent class
class Sample {
   // protected elements
  protected:
   int age;
};

// declare child class
class SampleChild : public Sample {

  public:
   void displayAge(int a) {
       age = a;
       cout << "Age = " << age << endl;
   }

};

int main() {
   int ageInput;

   // declare object of child class
   SampleChild child;

   cout << "Enter your age: ";
   cin >> ageInput;

   // call child class function
```

```
    // pass ageInput as argument
    child.displayAge(ageInput);

    return 0;
}
```

Output

```
Enter your age: 20
Age = 20
```

Here, SampleChild is an inherited class that is derived from Sample. The variable age is declared in Sample with the protected keyword.

This means that SampleChild can access age since Sample is its parent class.

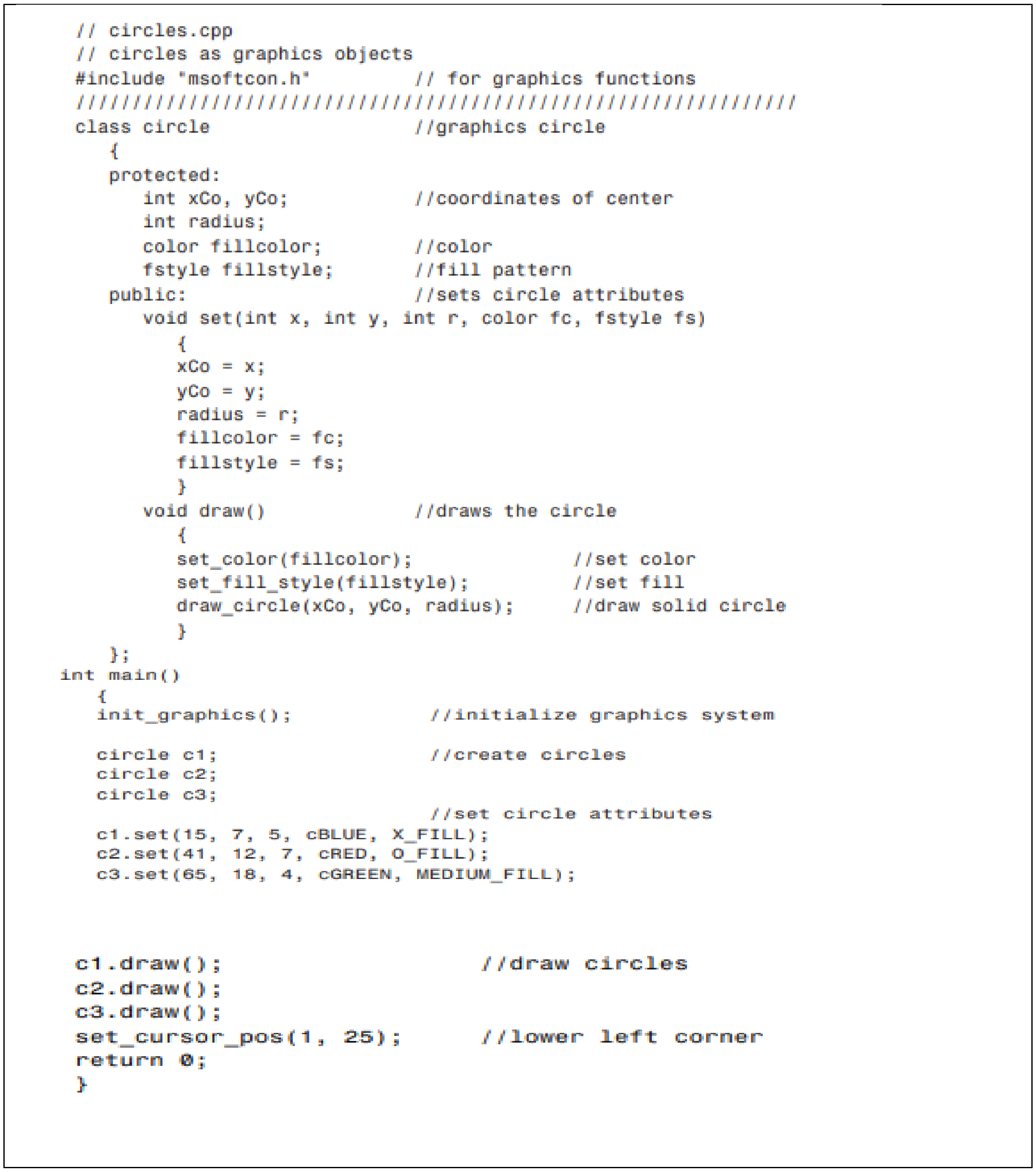
We see this as we have assigned the value of age in SampleChild even though age is declared in the Sample class.

Note: By default, class members in C++ are private, unless specified otherwise.

Example:

Circles as Objects

In our next example we'll examine an object used to represent a circle: the kind of circle displayed on your computer screen. The program creates three circles with various characteristics and displays them. Here's the listing for CIRCLES:



```
    // circles.cpp
    // circles as graphics objects
    #include "msoftcon.h"          // for graphics functions
    ////////////////////////////////////////////////////////////////
    class circle                   //graphics circle
       {
       protected:
          int xCo, yCo;            //coordinates of center
          int radius;
          color fillcolor;         //color
          fstyle fillstyle;        //fill pattern
       public:                     //sets circle attributes
          void set(int x, int y, int r, color fc, fstyle fs)
             {
             xCo = x;
             yCo = y;
             radius = r;
             fillcolor = fc;
             fillstyle = fs;
             }
          void draw()              //draws the circle
             {
             set_color(fillcolor);              //set color
             set_fill_style(fillstyle);         //set fill
             draw_circle(xCo, yCo, radius);     //draw solid circle
             }
       };
   int main()
      {
      init_graphics();             //initialize graphics system

      circle c1;                   //create circles
      circle c2;
      circle c3;
                                   //set circle attributes
      c1.set(15, 7, 5, cBLUE, X_FILL);
      c2.set(41, 12, 7, cRED, O_FILL);
      c3.set(65, 18, 4, cGREEN, MEDIUM_FILL);


    c1.draw();                     //draw circles
    c2.draw();
    c3.draw();
    set_cursor_pos(1, 25);         //lower left corner
    return 0;
    }
```

The output of this program is the same as that of the CIRCSTRC program shown in Figure 2.5. In CIRCLES, each circle is represented as a C++ object rather than as a combination of a structure variable and an unrelated circ_draw() function, as it was in CIRCSTRC. Notice in CIRCLES how everything connected with a circle—attributes and functions—is brought together in the class definition.

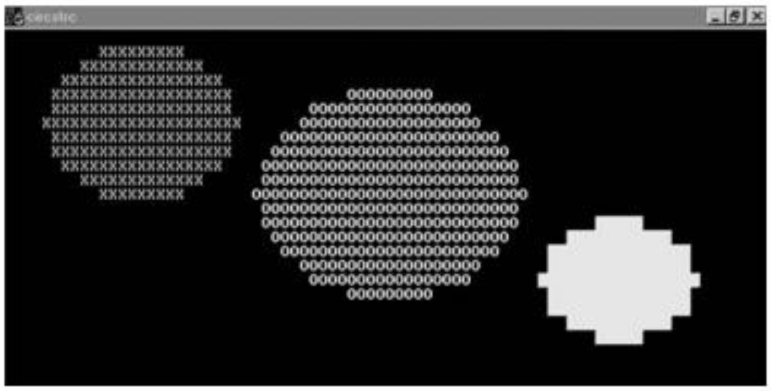


Figure 2.5 output program

Example:-The most important property of a class is that it is a type, and as such, we can declare multiple objects of it. For example, following with the previous example of class Rectangle, we could have declared the object rectb in addition to object rect:

```
//example: one class, two objects
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area () {return width*height;}
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

in this particular case, the class (type of the objects) is Rectangle, of which there are two instances (i.e., objects): rect and rectb. Each one of them has its own member variables and member functions.

Notice that the call to rect.area() does not give the same result as the call to rectb.area(). This is because each object of class Rectangle has its own variables width and height, as they -in some way- have also their own function members set_value and area that operate on the object's own member variables.

Classes allow programming using object-oriented paradigms: Data and functions are both members of the object, reducing the need to pass and carry handlers or other state variables as arguments to functions, because they are part of the object whose member is called. Notice that no arguments were passed on the calls to rect.area or rectb.area. Those member functions directly used the data members of their respective objects rect and rectb.

C++ Objects as Data Types:

Here’s another kind of entity C++ objects can represent: variables of a user-defined data type. We’ll use objects to represent distances measured in the English system, Here’s the listing for ENGLOBJ:

```
// englobj.cpp
// objects using English measurements
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
   private:
      int feet;
      float inches;
   public:
      void setdist(int ft, float in)  //set Distance to args
         { feet = ft; inches = in; }

      void getdist()              //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }

      void showdist()             //display distance
         { cout << feet << "\'-" << inches << '\"'; }


  };
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist1, dist2;         //define two lengths

   dist1.setdist(11, 6.25);       //set dist1
   dist2.getdist();               //get dist2 from user

                                  //display lengths
   cout << "\ndist1 = ";  dist1.showdist();
   cout << "\ndist2 = ";  dist2.showdist();
   cout << endl;
   return 0;
   }
```

In this program, the class Distance contains two data items, feet and inches. Here the class Distance also has three member functions: setdist(), which uses arguments to set feet and inches; getdist(), which gets values for feet and inches from the user at the keyboard; and showdist(), which displays the distance in feet-and-inches format.

The value of an object of class Distance can thus be set in either of two ways. In main(), we define two objects of class Distance: dist1 and dist2.

The first is given a value using the setdist() member function with the arguments 11 and 6.25, and the second is given a value that is supplied by the user. Here’s a sample interaction with the program:

```
Enter feet: 10
Enter inches: 4.75

dist1 = 11'-6.25"     ←———     provided by arguments
dist2 = 10'-4.75"     ←———     input by the user
```

Constructors:-

What would happen in the previous example if we called the member function area before having called set_values? An undetermined result, since the members width and height had never been assigned a value.

In order to avoid that, a class can include a special function called its *constructor*, which is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage.

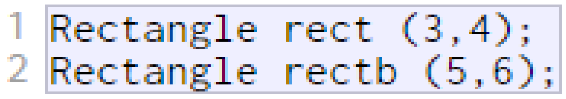
This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type; not even void.

The Rectangle class above can easily be improved by implementing a constructor:

```cpp
// example: class constructor
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle (int,int);
    int area () {return (width*height);}
};

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

But now, class Rectangle has no member function set_values, and has instead a constructor that performs a similar action: it initializes the values of width and height with the arguments passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:



```cpp
Rectangle rect (3,4);
Rectangle rectb (5,6);
```

Constructors cannot be called explicitly as if they were regular member functions. They are only executed once, when a new object of that class is created.

Notice how neither the constructor prototype declaration (within the class) nor the latter constructor definition, have return values; not even void: Constructors never return values, they simply initialize the object.

Example:

The ENGLOBJ example shows two ways that member functions can be used to give values to the data items in an object. Sometimes, however, it's convenient if an object can initialize itself when it's first created, without requiring a separate call to a member function. Automatic initialization is carried out using a special member function called a constructor. **A constructor is a member function that is executed automatically whenever an object is created**.

Example A Counter:-

As an example, we'll create a class of objects that might be useful as a general-purpose programming element. A counter is a variable that counts things. Maybe it counts file accesses, or the number of times the user presses the Enter key, or the number of customers entering a bank. Each time such an event takes place, the counter is incremented (1 is added to it). The counter can also be accessed to find the current count.

Let's assume that this counter is important in the program and must be accessed by many different functions. In procedural languages such as C, a counter would probably be implemented as a global variable. However, as we noted, global variables complicate the program's design and may be modified accidentally. This example, COUNTER, provides a counter variable that can be modified only through its member functions.

```
// counter.cpp
// object represents a counter variable
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;                 //count
   public:
      Counter() : count(0)                //constructor
         { /*empty body*/ }
      void inc_count()                    //increment count
         { count++; }
      int get_count()                     //return count
         { return count; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Counter c1, c2;                    //define and initialize

   cout << "\nc1=" << c1.get_count();  //display
   cout << "\nc2=" << c2.get_count();

   c1.inc_count();                    //increment c1
   c2.inc_count();                    //increment c2
   c2.inc_count();                    //increment c2

   cout << "\nc1=" << c1.get_count();  //display again
   cout << "\nc2=" << c2.get_count();


            cout << endl;
            return 0;
            }
```

The Counter class has one data member: count, of type unsigned int (since the count is always positive). It has three member functions: the constructor Counter(), which we'll look at in a moment; inc_count(), which adds 1 to count; and get_count(), which returns the current value of count.

Automatic Initialization:-

When an object of type Counter is first created, we want its count to be initialized to 0. After all, most counts start at 0. We could provide a set_count() function to do this and call it with an argument of 0, or we could provide a zero_count() function, which would always set count to 0.

However, such functions would need to be executed every time we created a Counter object.

```
Counter c1;        //every time we do this,
c1.zero_count();   //we must do this too
```

This is mistake prone, because the programmer may forget to initialize the object after creating it. It's more reliable and convenient, especially when there are a great many objects of a given class, to cause each object to initialize itself when it's created. In the Counter class, the constructor Counter() does this. This function is called automatically whenever a new object of type Counter is created. Thus in main() the statement

```
Counter c1, c2;
```

creates two objects of type Counter. As each is created, its constructor, Counter(), is executed. This function sets the count variable to 0. So the effect of this single statement is to not only create two objects, but also to initialize their count variables to 0.

Same Name as the Class:-

There are some unusual aspects of constructor functions. First, it is no accident that they have exactly the same name (Counter in this example) as the class of which they are members. This is one way the compiler knows they are constructors.

Second, no return type is used for constructors. Why not? Since the constructor is called automatically by the system, there's no program for it to return anything to; a return value wouldn't make sense. This is the second way the compiler knows they are constructors.

Initializer List:-

One of the most common tasks a constructor carries out is initializing data members. In the Counter class the constructor must initialize the count member to 0. You might think that this would be done in the constructor's function body, like this:

```
count()
   { count = 0; }
```

However, this is not the preferred approach (although it does work). Here's how you should initialize a data member:

```
count() : count(0)
   {  }
```

The initialization takes place following the member function declarator but before the function body. It's preceded by a colon. The value is placed in parentheses following the member data.

If multiple members must be initialized, they're separated by commas. The result is the initializer list (sometimes called by other names, such as the member-initialization list).

```
someClass() : m1(7), m2(33), m2(4) ←———    initializer list
   {  }
```

Why not initialize members in the body of the constructor? The reasons are complex, but have to do with the fact that members initialized in the initializer list are given a value before the constructor even starts to execute. This is important in some situations. For example, the initializer list is the only way to initialize const member data and references.

Actions more complicated than simple initialization must be carried out in the constructor body, as with ordinary functions.

Counter Output:-

The main() part of this program exercises the Counter class by creating two counters, c1 and c2. It causes the counters to display their initial values, which—as arranged by the constructor—are 0. It then increments c1 once and c2 twice, and again causes the counters to display themselves (non-criminal behavior in this context). Here's the output:

```
c1=0
c2=0
c1=1
c2=2
```

If this isn’t enough proof that the constructor is operating as advertised, we can rewrite the constructor to print a message when it executes.

```
Counter() : count(0)
    { cout << "I'm the constructor\n";  }
```

Now the program’s output looks like this:

```
I'm the constructor
I'm the constructor


c1=0
c2=0
c1=1
c2=2
```

As you can see, the constructor is executed twice—once for c1 and once for c2—when the statement

```
Counter c1, c2;
```

is executed in main().

Example A Graphics:-

Let’s rewrite our earlier CIRCLES example to use a constructor instead of a set() function. To handle the initialization of the five attributes of circles, this constructor will have five arguments and five items in its initialization list. Here’s the listing for CIRCTOR:

```
// circtor.cpp
// circles use constructor for initialization
#include "msoftcon.h"         // for graphics functions
////////////////////////////////////////////////////////////////
class circle                  //graphics circle
   {
   protected:
      int xCo, yCo;           //coordinates of center
      int radius;
      color fillcolor;        //color
      fstyle fillstyle;       //fill pattern
   public:
                              //constructor
      circle(int x, int y, int r, color fc, fstyle fs) :
         xCo(x), yCo(y), radius(r), fillcolor(fc), fillstyle(fs)
         {  }

      void draw()             //draws the circle
         {
         set_color(fillcolor);              //set color



         set_fill_style(fillstyle);         //set fill
         draw_circle(xCo, yCo, radius);     //draw solid circle
         }
   };
int main()
   {
   init_graphics();           //initialize graphics system
                              //create circles
   circle c1(15, 7, 5, cBLUE, X_FILL);
   circle c2(41, 12, 7, cRED, O_FILL);
   circle c3(65, 18, 4, cGREEN, MEDIUM_FILL);

   c1.draw();                 //draw circles
   c2.draw();
   c3.draw();
   set_cursor_pos(1, 25);     //lower left corner
   return 0;
   }
```

This program is similar to CIRCLES, except that set() has been replaced by the constructor. Note how this simplifies main(). Instead of two separate statements for each object, one to create it and one to set its attributes, now one statement both creates the object and sets its attributes at the same time.

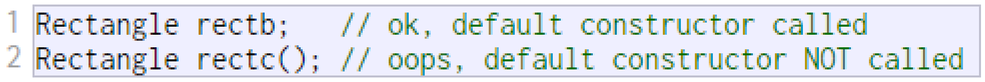
Overloading constructors

Like any other function, a constructor can also be overloaded with different versions taking different parameters: with a different number of parameters and/or parameters of different types. The compiler will automatically call the one whose parameters match the arguments:

```
// overloading class constructors
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    Rectangle ();
    Rectangle (int,int);
    int area (void) {return (width*height);}
};

Rectangle::Rectangle () {
  width = 5;
  height = 5;
}

Rectangle::Rectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  Rectangle rect (3,4);
  Rectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

In the above example, two objects of class Rectangle are constructed: rect and rectb. rect is constructed with two arguments, like in the example before.

But this example also introduces a special kind constructor: the *default constructor*. The *default constructor* is the constructor that takes no parameters, and it is special because it is called when an object is declared but is not initialized with any arguments. In the example above, the *default constructor* is called for rectb. Note how rectb is not even constructed with an empty set of parentheses - in fact, empty parentheses cannot be used to call the default constructor:

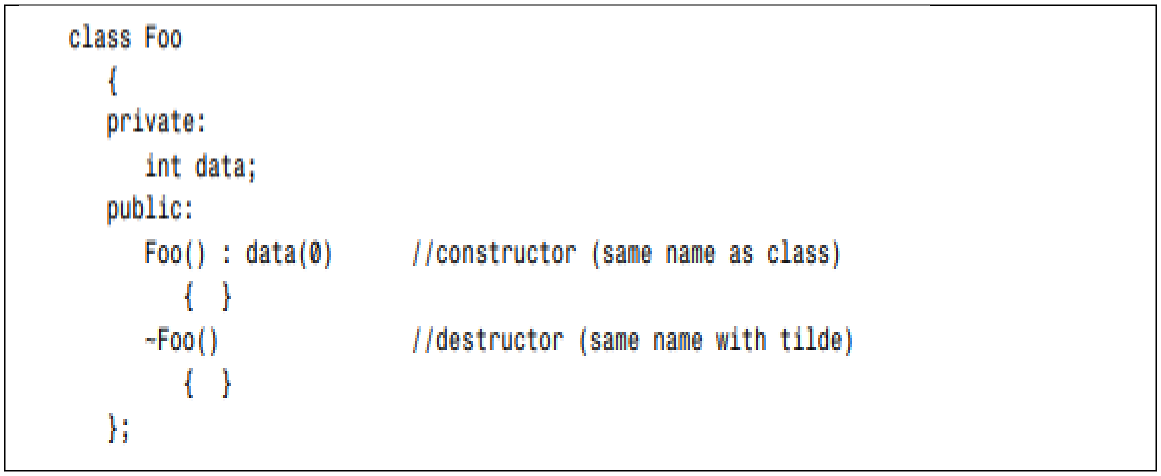


```
Rectangle rectb;   // ok, default constructor called
Rectangle rectc(); // oops, default constructor NOT called
```

This is because the empty set of parentheses would make of rectc a function declaration instead of an object declaration: It would be a function that takes no arguments and returns a value of type Rectangle.

Destructors:-

We’ve seen that a special member function—the constructor—is called automatically when an object is first created. You might guess that another function is called automatically when an object is destroyed. This is indeed the case. Such a function is called a destructor. A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde:



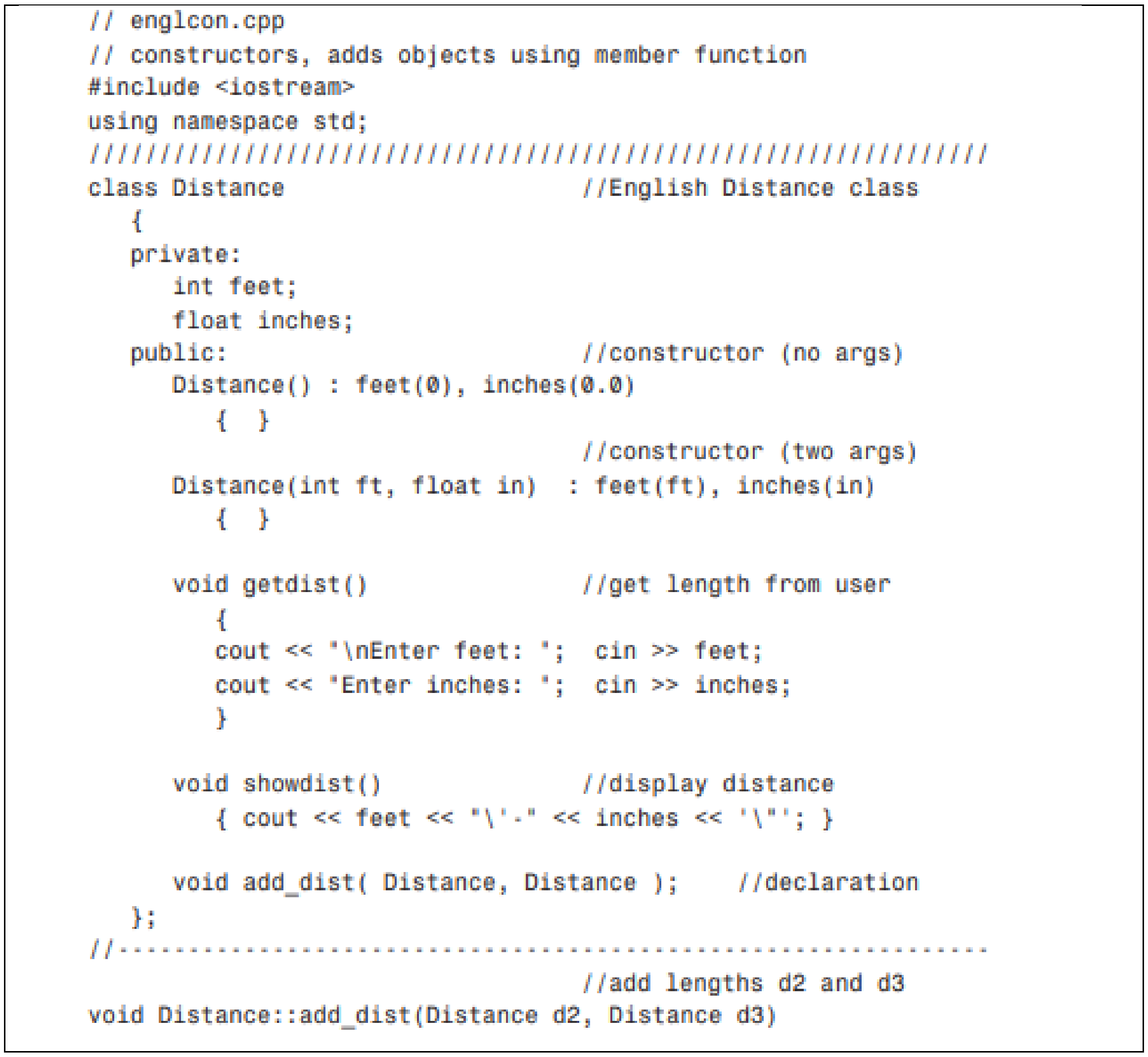
```
class Foo
   {
   private:
      int data;
   public:
      Foo() : data(0)      //constructor (same name as class)
         {  }
      ~Foo()               //destructor (same name with tilde)
         {  }
   };
```

Like constructors, destructors do not have a return value. They also take no arguments (the assumption being that there’s only one way to destroy an object).

The most common use of destructors is to deallocate memory that was allocated for the object by the constructor**.**

Objects as Function Arguments:-

Our next program adds some embellishments to the ENGLOBJ example. It also demonstrates some new aspects of classes: constructor overloading, defining member functions outside the class, and—perhaps most importantly—objects as function arguments. Here’s the listing for ENGLCON:



```
// englcon.cpp
// constructors, adds objects using member function
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                          //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                              //constructor (no args)
      Distance() : feet(0), inches(0.0)
         {  }
                                        //constructor (two args)
      Distance(int ft, float in)  : feet(ft), inches(in)
         {  }

      void getdist()                    //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }

      void showdist()                   //display distance
         { cout << feet << "\'-" << inches << '\"'; }

      void add_dist( Distance, Distance );    //declaration
   };
//--------------------------------------------------------------
                                        //add lengths d2 and d3
void Distance::add_dist(Distance d2, Distance d3)
```

```
   {
   inches = d2.inches + d3.inches; //add the inches
   feet = 0;                       //(for possible carry)
   if(inches >= 12.0)              //if total exceeds 12.0,
      {                            //then decrease inches
      inches -= 12.0;              //by 12.0 and
      feet++;                      //increase feet
      }                            //by 1
   feet += d2.feet + d3.feet;      //add the feet
   }
int main()
    {
   Distance dist1, dist3;          //define two lengths
   Distance dist2(11, 6.25);       //define and initialize dist2

   dist1.getdist();                //get dist1 from user
   dist3.add_dist(dist1, dist2);   //dist3 = dist1 + dist2

                                   //display all lengths
   cout << "\ndist1 = ";  dist1.showdist();
   cout << "\ndist2 = ";  dist2.showdist();
   cout << "\ndist3 = ";  dist3.showdist();
   cout << endl;
   return 0;
   }
```

This program starts with a distance dist2 set to an initial value and adds to it a distance dist1, whose value is supplied by the user, to obtain the sum of the distances. It then displays all three distances:

```
Enter feet: 17
Enter inches: 5.75

dist1 = 17'-5.75"
dist2 = 11'-6.25"
dist3 = 29'-0"
```

Overloaded Constructors:-

It’s convenient to be able to give variables of type Distance a value when they are first created. That is, we would like to use definitions like

```
Distance width(5, 6.25);
```

which defines an object, width, and simultaneously initializes it to a value of 5 for feet and 6.25 for inches.

To do this we write a constructor like this:

```
Distance(int ft, float in) : feet(ft), inches(in)
    {  }
```

This sets the member data feet and inches to whatever values are passed as arguments to the constructor. So far so good.

However, we also want to define variables of type Distance without initializing them, as we did in ENGLOBJ

```
Distance dist1, dist2;
```

In that program there was no constructor, but our definitions worked just fine. How could they work without a constructor? Because an implicit no-argument constructor is built into the program automatically by the compiler, and it’s this constructor that created the objects, even though we didn’t define it in the class. This no-argument constructor is called the default constructor. If it weren’t created automatically by the constructor, you wouldn’t be able to create objects of a class for which no constructor was defined.

Often we want to initialize data members in the default (no-argument) constructor as well. If we let the default constructor do it, we don’t really know what values the data members may be given. If we care what values they may be given, we need to explicitly define the constructor. In ENGLECON we show how this looks:

```
Distance() : feet(0), inches(0.0)     //default constructor
    {  }         //no function body, doesn't do anything
```

The data members are initialized to constant values, in this case the integer value 0 and the float value 0.0, for feet and inches respectively. Now we can use objects initialized with the no-argument constructor and be confident that they represent no distance (0 feet plus 0.0 inches) rather than some arbitrary value.

Since there are now two explicit constructors with the same name, Distance(), we say the constructor is overloaded. Which of the two constructors is executed when an object is created depends on how many arguments are used in the definition:

```
Distance length;            // calls first constructor
Distance width(11, 6.0);    // calls second constructor
```

Review Questions

1. What is the purpose of a class definition?

2. A ________ has the same relation to an ________ that a basic data type has to a variable

of that type.

3. In a class definition, data or functions designated private are accessible

a. to any function in the program.

b. only if you know the password.

c. to member functions of that class.

d. only to public members of the class.

4. Write a class definition that creates a class called leverage with one private data member, crowbar, of type int and one public function whose declaration is void pry().

5. True or false: Data items in a class must be private.

6. Write a statement that defines an object called lever1 of the leverage class described in question 4.

Exercises:-

(1) Create a class named 'Student' with a string variable 'name' and an integer variable 'roll_no'. Assign the value of roll_no as '2' and that of name as "John" by creating an object of the class Student.

(2) Assign and print the roll number, phone number and address of two students having names "Sam" and "John" respectively by creating two objects of the class 'Student'.

(3) Write a program to print the area and perimeter of a triangle having sides of 3, 4 and 5 units by creating a class named 'Triangle' with a function to print the area and perimeter.

(4) Write a program to print the area and perimeter of a triangle having sides of 3, 4 and 5 units by creating a class named 'Triangle' with the constructor having the three sides as its parameters.

(5) Write a program to print the area of two rectangles having sides (4,5) and (5,8) respectively by creating a class named 'Rectangle' with a function named 'Area' which returns the area. Length and breadth are passed as parameters to its constructor.

(6) Write a program to print the area of a rectangle by creating a class named 'Area' having two functions. First function named as 'setDim' takes the length and breadth of the rectangle as parameters and the second function named as 'getArea' returns the area of the rectangle. Length and breadth of the rectangle are entered through keyboard.

(7) Write a program to print the area of a rectangle by creating a class named 'Area' taking the values of its length and breadth as parameters of its constructor and having a function named 'returnArea' which returns the area of the rectangle. Length and breadth of the rectangle are entered through keyboard.

(8) Print the average of three numbers entered by the user by creating a class named 'Average' having a function to calculate and print the average without creating any object of the Average class.

(9) Print the sum, difference and product of two complex numbers by creating a class named 'Complex' with separate functions for each operation whose real and imaginary parts are entered by the user.
(10)Write a program to print the volume of a box by creating a class named 'Volume' with an initialization list to initialize its length, breadth and height. (just to make you familiar with initialization lists)

Chapter 3
Operator Overloading

- Overloading Unary Operators

- Overloading Binary Operators

Operator overloading is one of the most exciting features of object-oriented programming. It can transform complex, obscure program listings into intuitively obvious ones. For example, statements like

d3.addobjects(d1,d2);

or the similar but equally obscure

d3=d1.addobjects(d2);

can be changed to the much more readable

d3=d1+d2;

The rather forbidding term operator overloading refers to giving the normal C++ operators, such as +, *, <=, and +=, additional meanings when they are applied to user-defined data types.

Normally

a =b + c;

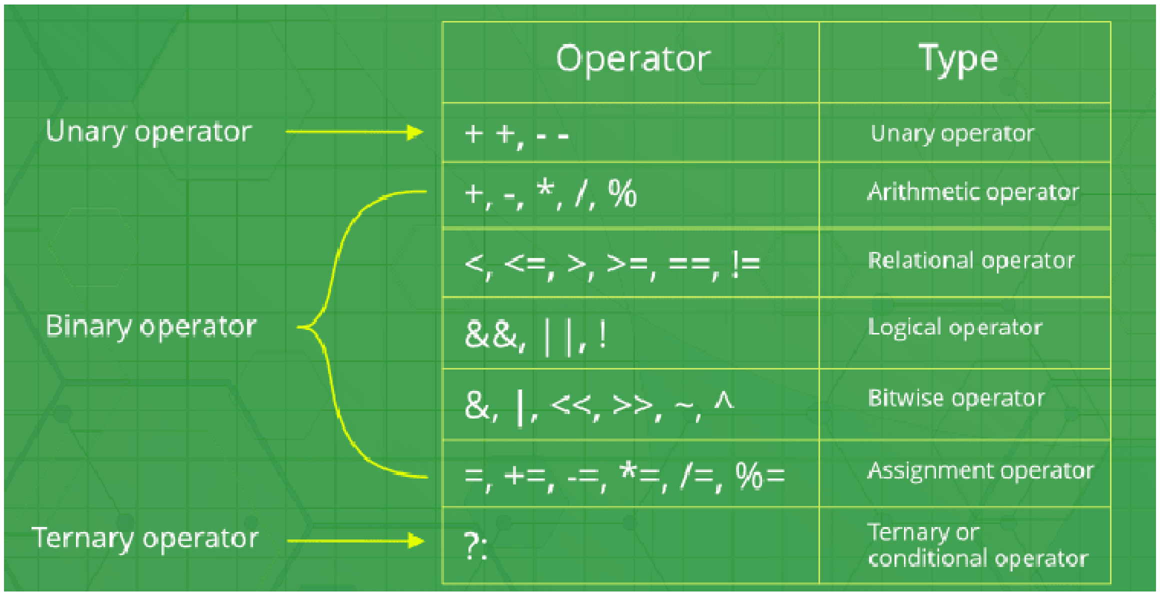
works only with basic types such as int and float, and attempting to apply it when a, b, and c are objects of a user-defined class will cause complaints from the compiler. However, using overloading, you can make this statement legal even when a, b, and c are user-defined types.

In effect, operator overloading gives you the opportunity to redefine the C++ language. If you find yourself limited by the way the C++ operators work, you can change them to do whatever you want. By using classes to create new kinds of variables, and operator overloading to create new definitions for operators, you can extend C++ to be, in many ways, a new language of your own design.

Another kind of operation**, data type conversion**, is closely connected with operator overloading.

C++ handles the conversion of simple types, such as int and float, automatically; but conversions involving user-defined types require some work on the programmer's part.

Operators in C++


Unary operator
Binary operator
Ternary operator

Operator	Type
+ +, - -	Unary operator
+, -, *, /, %	Arithmetic operator
<, <=, >, >=, ==, !=	Relational operator
&&, | |, !	Logical operator
&, |, <<, >>, ~, ^	Bitwise operator
=, +=, -=, *=, /=, %=	Assignment operator
?:	Ternary or conditional operator

Overloading Unary Operators:-

Unary operators act on only one operand. (An operand is simply a variable acted on by an operator.)

Examples of unary operators are the increment and decrement operators ++ and --, and the unary minus, as in -33.

In the COUNTER example in Chapter 2, “Objects and Classes,” we created a class Counter to keep track of a count. Objects of that class were incremented by calling a member function

c1.inc_count( );

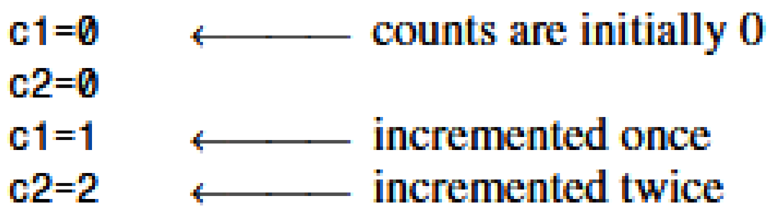
That did the job, but the listing would have been more readable if we could have used the increment operator ++ instead:

++c1;

Let’s rewrite COUNTER to make this possible. Here’s the listing for COUNTPP1:

```
// countpp1.cpp
// increment counter variable with ++ operator
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;              //count
   public:
      Counter() : count(0)             //constructor
         {  }
      unsigned int get_count()         //return count
         { return count; }
      void operator ++ ()               //increment (prefix)
         {
         ++count;
         }
   };
////////////////////////////////////////////////////////////////

int main()
   {
   Counter c1, c2;                        //define and initialize

   cout << "\nc1=" << c1.get_count();  //display
   cout << "\nc2=" << c2.get_count();

   ++c1;                    //increment c1
   ++c2;                    //increment c2
   ++c2;                    //increment c2



cout << "\nc1=" << c1.get_count();  //display again
cout << "\nc2=" << c2.get_count() << endl;
return 0;
```

In this program we create two objects of class Counter: c1 and c2. The counts in the objects are displayed; they are initially 0. Then, using the overloaded ++ operator, we increment c1 once and c2 twice, and display the resulting values. Here’s the program’s output:

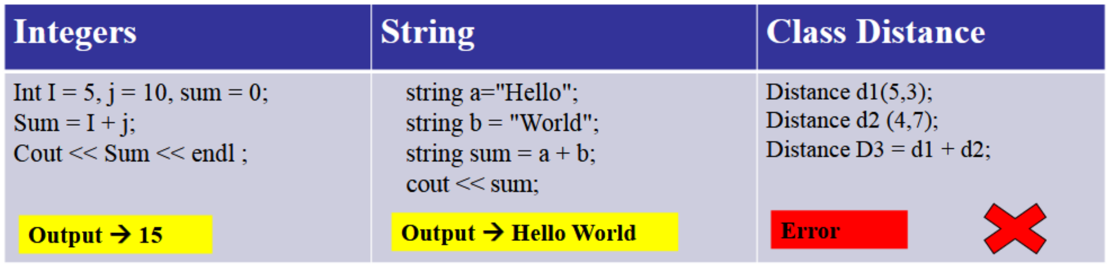


```
c1=0     ←——— counts are initially 0
c2=0
c1=1     ←——— incremented once
c2=2     ←——— incremented twice
```

The statements responsible for these operations are

```
++c1;
++c2;
++c2;
```

The ++ operator is applied once to c1 and twice to c2. We use prefix notation in this example; we’ll explore postfix late



Integers	String	Class Distance

Int I = 5, j = 10, sum = 0;
Sum = I + j;
Cout << Sum << endl ;
	string a="Hello";
	string b = "World";
	string sum = a + b;
	cout << sum;
		Distance d1(5,3);
		Distance d2 (4,7);
		Distance D3 = d1 + d2;

Output → 15	**Output → Hello World**	**Error** ❌

The operator Keyword:-

How do we teach a normal C++ operator to act on a user-defined operand? The keyword operator is used to overload the ++ operator in this declarator:

void operator ++ ()

The return type (void in this case) comes first, followed by the keyword operator, followed by the operator itself (++), and finally the argument list enclosed in parentheses (which are empty here). This declarator syntax tells the compiler to call this member function whenever the ++ operator is encountered, provided the operand (the variable operated on by the ++) is of type Counter.

We saw in Chapter 1, “Functions,” that the only way the compiler can distinguish between overloaded functions is by looking at the data types and the number of their arguments. In the same way, the only way it can distinguish between overloaded operators is by looking at the data type of their operands. If the operand is a basic type such as an int, as in

++intvar;

then the compiler will use its built-in routine to increment an int. But if the operand is a Counter variable, the compiler will know to use our user-written operator++() instead.

Operator Arguments:-

In main() the ++ operator is applied to a specific object, as in the expression ++c1. Yet operator++() takes no arguments. What does this operator increment? It increments the count data in the object of which it is a member. Since member functions can always access the particular object for which they’ve been invoked, this operator requires no arguments.

This is shown in Figure 3.1.

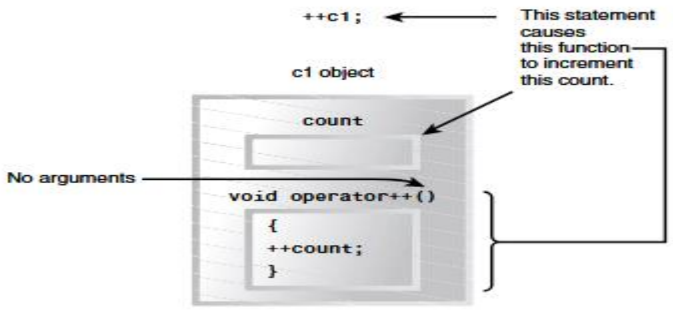

++c1;
This statement causes this function to increment this count.
c1 object
count
No arguments
void operator++()
{
++count;
}

Figure 3.1 Overloaded Unary operator : no argument

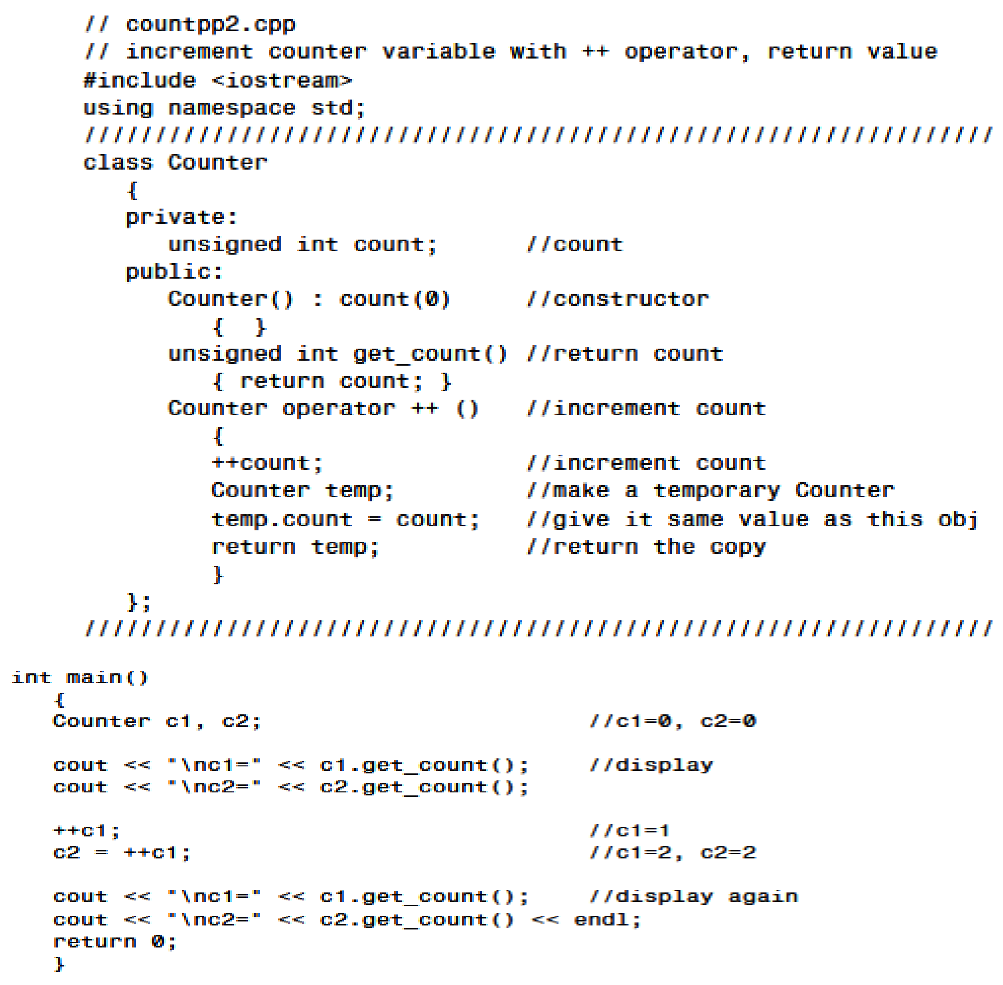
Operator Return Values:-

The operator++() function in the COUNTPP1 program has a subtle defect. You will discover it if you use a statement like this in main():

c1 = ++c2;

The compiler will complain. Why? Because we have defined the ++ operator to have a return type of void in the operator++() function, while in the assignment statement it is being asked to return a variable of type Counter. That is, the compiler is being asked to return whatever value c2 has after being operated on by the ++ operator, and assign this value to c1. So as defined in COUNTPP1, we can't use ++ to increment Counter objects in assignments; it must always stand alone with its operand. Of course the normal ++ operator, applied to basic data types such as int, would not have this problem.

To make it possible to use our homemade operator++() in assignment expressions, we must provide a way for it to return a value. The next program, COUNTPP2, does just that



```
    // countpp2.cpp
    // increment counter variable with ++ operator, return value
    #include <iostream>
    using namespace std;
    ////////////////////////////////////////////////////////////////
    class Counter
       {
       private:
          unsigned int count;      //count
       public:
          Counter() : count(0)     //constructor
             {  }
          unsigned int get_count() //return count
             { return count; }
          Counter operator ++ ()   //increment count
             {
             ++count;              //increment count
             Counter temp;         //make a temporary Counter
             temp.count = count;   //give it same value as this obj
             return temp;          //return the copy
             }
       };
    ////////////////////////////////////////////////////////////////

int main()
   {
   Counter c1, c2;                       //c1=0, c2=0

   cout << "\nc1=" << c1.get_count();    //display
   cout << "\nc2=" << c2.get_count();

   ++c1;                                 //c1=1
   c2 = ++c1;                            //c1=2, c2=2

   cout << "\nc1=" << c1.get_count();    //display again
   cout << "\nc2=" << c2.get_count() << endl;
   return 0;
   }
```

now return a value, so they can be used in other expressions, such as

```
c2 = ++c1;
```

as shown in main(), where the value returned from c1++ is assigned to c2. The output from this program is

```
c1=0
c2=0
c1=2
c2=2
```

Postfix Notation:-

So far we’ve shown the increment operator used only in its prefix form.

++c1

What about postfix, where the variable is incremented after its value is used in the expression?

c1++

To make both versions of the increment operator work, we define two overloaded ++ operators, as shown in the POSTFIX program:

```
// postfix.cpp
// overloaded ++ operator in both prefix and postfix
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter
   {
   private:
      unsigned int count;         //count
   public:
      Counter() : count(0)        //constructor  no args
         {  }
      Counter(int c) : count(c)   //constructor, one arg
         {  }
      unsigned int get_count() const //return count
         { return count; }

      Counter operator ++ ()      //increment count (prefix)
         {                        //increment count, then return
         return Counter(++count); //an unnamed temporary object
         }                        //initialized to this count

      Counter operator ++ (int)   //increment count (postfix)
         {                        //return an unnamed temporary
         return Counter(count++); //object initialized to this
         }                        //count, then increment count
   };
////////////////////////////////////////////////////////////////

      int main()
         {
         Counter c1, c2;                       //c1=0, c2=0

         cout << "\nc1=" << c1.get_count();    //display
         cout << "\nc2=" << c2.get_count();

         ++c1;                                 //c1=1
         c2 = ++c1;                            //c1=2, c2=2 (prefix)

         cout << "\nc1=" << c1.get_count();    //display
         cout << "\nc2=" << c2.get_count();

         c2 = c1++;                            //c1=3, c2=2 (postfix)




   cout << "\nc1=" << c1.get_count();    //display again
   cout << "\nc2=" << c2.get_count() << endl;
   return 0;
   }
```

Now there are two different declarators for overloading the ++ operator. The one we've seen before, for prefix notation, is

Counter operator ++ ()

The new one, for postfix notation, is Counter operator ++ (int)

The only difference is the int in the parentheses. This int isn't really an argument, and it doesn't mean integer. It's simply a signal to the compiler to create the postfix version of the operator. The designers of C++ are fond of recycling existing operators and keywords to play multiple roles, and int is the one they chose to indicate postfix. (Well, can you think of a better

syntax?) Here's the output from the program:

```
c1=0

c2=0

c1=2

c2=2

c1=3

c2=2
```

We saw the first four of these output lines in COUNTPP2 and COUNTPP3. But in the last two lines

we see the results of the statement

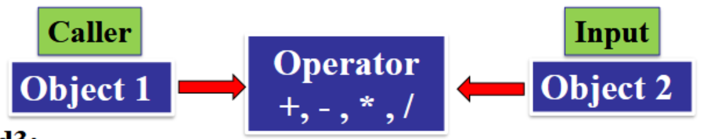
c2=c1++;

Here, c1 is incremented to 3, but c2 is assigned the value of c1 before it is incremented, so c2

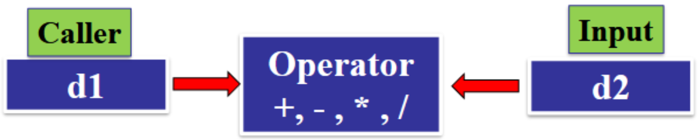
retains the value 2.

Of course, you can use this same approach with the decrement operator (--).

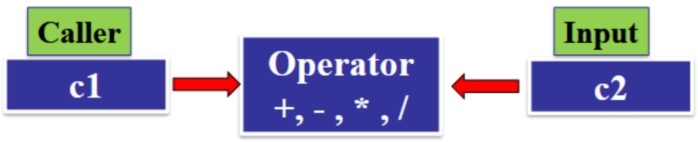
Overloading Binary Operators:-



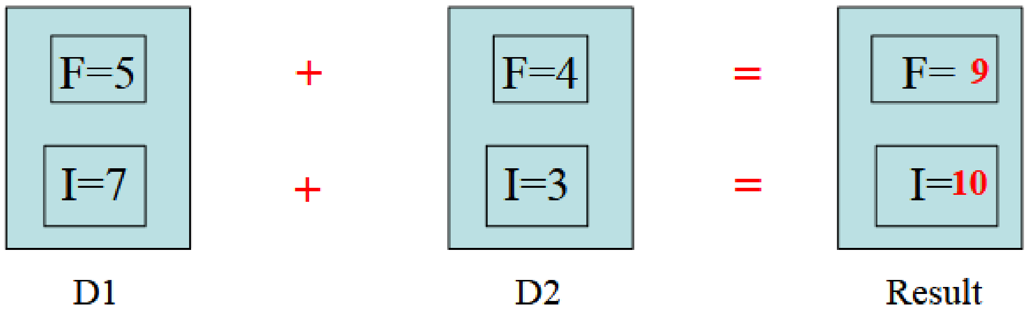
Distance d1 , d2, d3;
d3= d1 + d2



Counter c1 , c2, c3;
c3= c1 + c2



```
Distance Distance::operator + (Distance d2) const  //return sum
   {
   int f = feet + d2.feet;          //add the feet
   float i = inches + d2.inches;    //add the inches
   if(i >= 12.0)                    //if total exceeds 12.0,
      {                             //then decrease inches
      i -= 12.0;                    //by 12.0 and
      f++;                          //increase feet by 1
      }                             //return a temporary Distance
   return Distance(f,i);            //initialized to sum
   }
```



Binary operators can be overloaded just as easily as unary operators. We'll look at examples that overload arithmetic operators, comparison operators, and arithmetic assignment operators

Arithmetic Operators

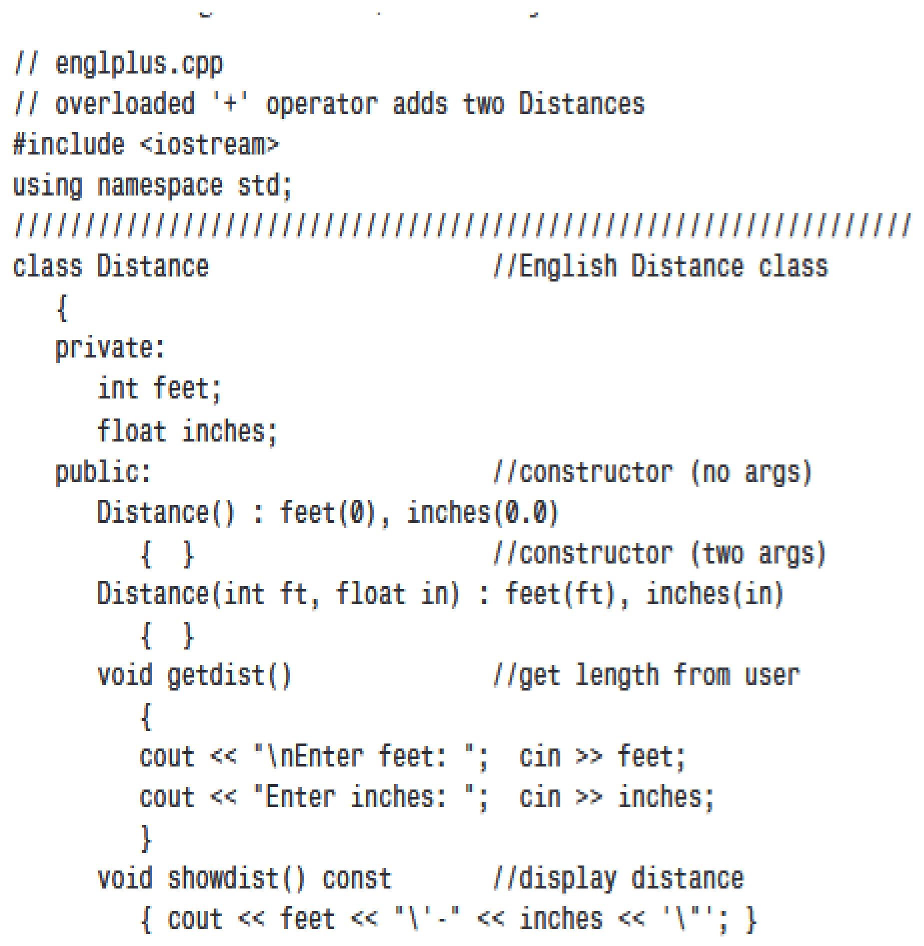
In the ENGLCON program in Chapter 2 we showed how two English Distance objects could be added using a member function add_dist():

dist3.add_dist(dist1, dist2);

By overloading the + operator we can reduce this dense-looking expression to

dist3 = dist1 + dist2;

Here's the listing for ENGLPLUS, which does just that:



```
// englplus.cpp
// overloaded '+' operator adds two Distances
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                        //constructor (no args)
      Distance() : feet(0), inches(0.0)
         {  }                     //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()              //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const       //display distance
         { cout << feet << "\'-" << inches << '\"'; }
      Distance operator + ( Distance ) const;  //add 2 distances
   };
//---------------------------------------------------------------
                              //add this distance to d2
Distance Distance::operator + (Distance d2) const  //return sum
   {
   int f = feet + d2.feet;         //add the feet
   float i = inches + d2.inches;   //add the inches
   if(i >= 12.0)                   //if total exceeds 12.0,
      {                            //then decrease inches
      i -= 12.0;                   //by 12.0 and
      f++;                         //increase feet by 1
      }                            //return a temporary Distance
   return Distance(f,i);           //initialized to sum
   }
////////////////////////////////////////////////////////////////
```

```
int main()
   {
   Distance dist1, dist3, dist4;   //define distances
   dist1.getdist();                //get dist1 from user

   Distance dist2(11, 6.25);       //define, initialize dist2

   dist3 = dist1 + dist2;          //single '+' operator

   dist4 = dist1 + dist2 + dist3;  //multiple '+' operators
                                   //display all lengths
   cout << "dist1 = ";  dist1.showdist(); cout << endl;
   cout << "dist2 = ";  dist2.showdist(); cout << endl;
   cout << "dist3 = ";  dist3.showdist(); cout << endl;
   cout << "dist4 = ";  dist4.showdist(); cout << endl;
   return 0;
   }
```

To show that the result of an addition can be used in another addition as well as in an assignment, another addition is performed in main(). We add dist1, dist2, and dist3 to obtain dist4 (which should be double the value of dist3), in the statement

dist4 = dist1 + dist2 + dist3;

Here’s the output from the program:

```
Enter feet: 10
Enter inches: 6.5

dist1 = 10'-6.5"     ←——— from user
dist2 = 11'-6.25"    ←——— initialized in program
dist3 = 22'-0.75"    ←——— dist1+dist2
dist4 = 44'-1.5"     ←——— dist1+dist2+dist3
```

In class Distance the declaration for the operator+() function looks like this:

Distance operator + (Distance);

This function has a return type of Distance, and takes one argument of type Distance.

In expressions like

dist3 = dist1 + dist2;

it's important to understand how the return value and arguments of the operator relate to the objects. When the compiler sees this expression it looks at the argument types, and finding only type Distance, it realizes it must use the Distance member function operator+(). But what does this function use as its argument—dist1 or dist2? And doesn't it need two arguments, since there are two numbers to be added?

Here's the key: The argument on the left side of the operator (dist1 in this case) is the object of which the operator is a member. The object on the right side of the operator (dist2) must be furnished as an argument to the operator. The operator returns a value, which can be assigned or used in other ways; in this case it is assigned to dist3. Figure 3.2 shows how this looks

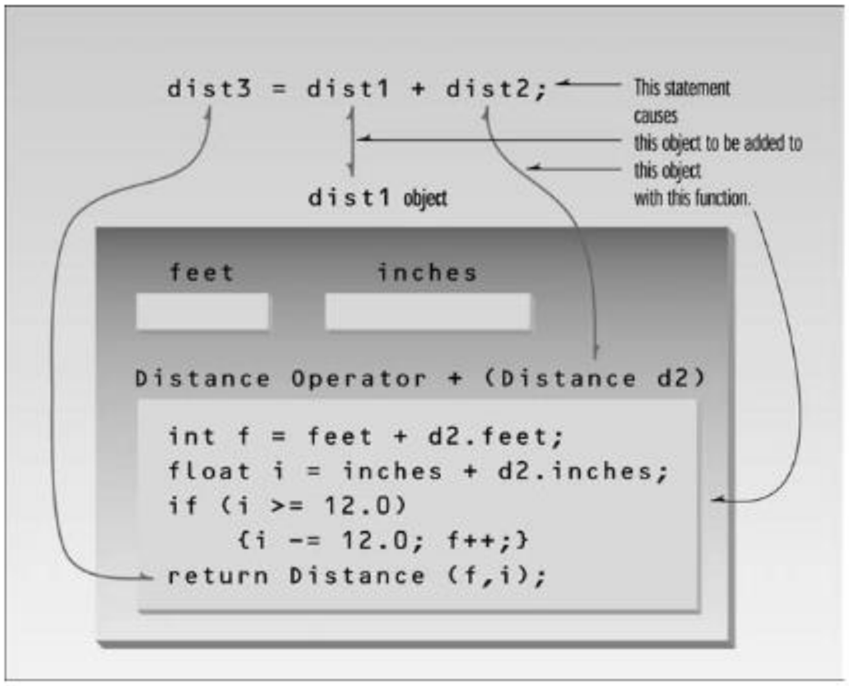

dist3 = dist1 + dist2;
This statement causes this object to be added to this object with this function.
dist1 object
feet
inches
Distance Operator + (Distance d2)
int f = feet + d2.feet;
float i = inches + d2.inches;
if (i >= 12.0)
{i -= 12.0; f++;}
return Distance (f,i);

Figure 3.2 Overloaded binary operator: one argument.

In the operator+() function, the left operand is accessed directly—since this is the object of which the operator is a member—using feet and inches. The right operand is accessed as the function's argument, as d2.feet and d2.inches.

We can generalize and say that an overloaded operator always requires one less argument than its number of operands, since one operand is the object of which the operator is a member.

That's why unary operators require no arguments.

To calculate the return value of operator+() in ENGLPLUS, we first add the feet and inches from the two operands (adjusting for a carry if necessary). The resulting values, f and i, are then used to initialize a nameless Distance object, which is returned in the statement

return Distance(f, i);

This is similar to the arrangement used in COUNTPP3, except that the constructor takes two arguments instead of one. The statement

dist3 = dist1 + dist2;

in main() then assigns the value of the nameless Distance object to dist3. Compare this intuitively obvious statement with the use of a function call to perform the same task, as in the ENGLCON example in Chapter 2.

Similar functions could be created to overload other operators in the Distance class, so you could subtract, multiply, and divide objects of this class in natural-looking ways.

Review Questions

1. Operator overloading is

a. making C++ operators work with objects.

b. giving C++ operators more than they can handle.

c. giving new meanings to existing C++ operators.

d. making new C++ operators.

2. Assuming that class X does not use any overloaded operators, write a statement that subtracts

an object of class X, x1, from another such object, x2, and places the result in x3.

3. Assuming that class X includes a routine to overload the - operator, write a statement that

would perform the same task as that specified in Question 2.

4. True or false: The >= operator can be overloaded.

5. Write a complete definition for an overloaded operator for the Counter class of the

COUNTPP1 example that, instead of incrementing the count, decrements it.

6. How many arguments are required in the definition of an overloaded unary operator?

7. Assume a class C with objects obj1, obj2, and obj3. For the statement obj3 =

obj1 - obj2 to work correctly, the overloaded - operator must

a. take two arguments.

b. return a value.

c. create a named temporary object.

d. use the object of which it is a member as an operand.

8. Write a complete definition for an overloaded ++ operator for the Distance class from

the ENGLPLUS example. It should add 1 to the feet member data, and make possible

statements like

dist1++;

9. Repeat Question 8, but allow statements like the following:

dist2 = dist1++;

10. When used in prefix form, what does the overloaded ++ operator do differently from

what it does in postfix form?

11. Here are two declarators that describe ways to add two string objects:

void add(String s1, String s2)

String operator + (String s)

Match the following from the first declarator with the appropriate selection from the second:

function name (add) matches _________.

return value (type void) matches _________.

first argument (s1) matches _________.

second argument (s2) matches _________.

object of which function is a member matches _________.

a. argument (s)

b. object of which operator is a member

c. operator (+)

d. return value (type String)

e. no match for this item

---

(12) What is a binary operator?
a) Operator that performs its action on a single operand
b) Operator that performs its action on two operand
c) Operator that performs its action on three operand
d) Operator that performs its action on any number of operand

(13) Which is the correct example of a binary operator?
a) ++
b) —
c) Dereferencing operator(*)
d) +

(14) Which is the correct example of a unary operator?
a) &
b) ==
c) —
d) /

(15) Which is called ternary operator?
a) ?:
b) &&
c) |||
d) ===

16) What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
using namespace std;
class complex
{
        int i;
        int j;
        public:
        complex(){}
        complex(int a, int b)
        {
                i = a;
                j = b;
        }

        complex operator+(complex c)
        {
                complex temp;
                temp.i = this->i + c.i;
                temp.j = this->j + c.j;
                return temp;
        }

        void show(){
                cout<<"Complex Number: "<<i<<" + i"<<j<<endl;
        }
};

int main(int argc, char const *argv[])
{
        complex c1(1,2);
        complex c2(3,4);
        complex c3 = c1 + c2;
        c3.show();
        return 0;
}
```

a) Complex Number: 4 + i6
b) Complex Number: 2 + i2
c) Error
d) Segmentation fault

Chapter 4
Inheritance

Introduction :-

Inheritance is probably the most powerful feature of object-oriented programming, after classes themselves. **Inheritance** is the process of creating new classes, called derived classes, from existing or base classes. The derived class inherits all the capabilities of the base class but can add embellishments and refinements of its own. The base class is unchanged by this process. The inheritance relationship is shown in Figure 4.1.

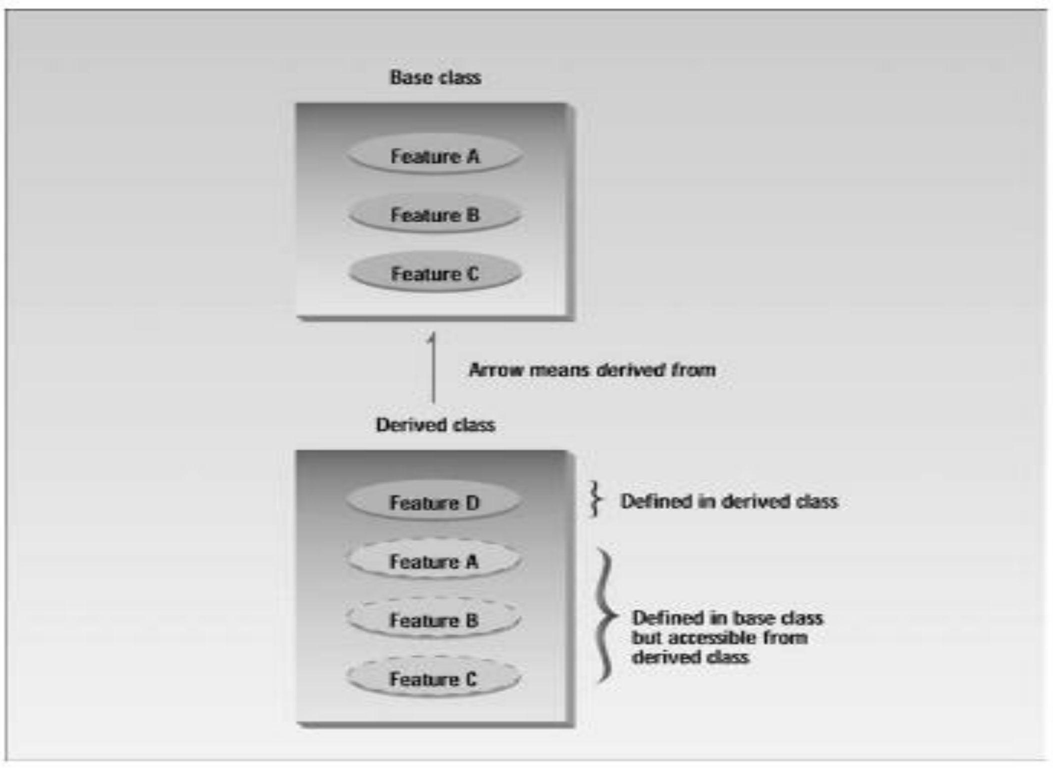


Figure 4.1 Inheritance

The arrow in Figure 4.1 goes in the opposite direction of what you might expect. If it pointed down we would label it inheritance. However, the more common approach is to point the arrow up, from the derived class to the base class, and to think of it as a “derived from” arrow.

Inheritance is an essential part of OOP. Its big payoff is that it permits code **reusability**. Once a base class is written and debugged, it need not be touched again, but, using inheritance, can nevertheless be adapted to work in different situations. Reusing existing code saves time and money and increases a program’s reliability. Inheritance can also help in the original conceptualization of a programming problem, and in the overall design of the program.

An important result of reusability is the ease of distributing class libraries. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

Definition :-

Inheritance is probably the most powerful feature of object-oriented programming, after classes themselves.

Inheritance is the process of creating new classes, called derived classes, from existing or base classes.

The derived class inherits all the capabilities of the base class but can add its own features. And the base class is unchanged by this process

Notes

Inheritance permits code reusability.

Reusing existing code saves time and money and increases a program's reliability

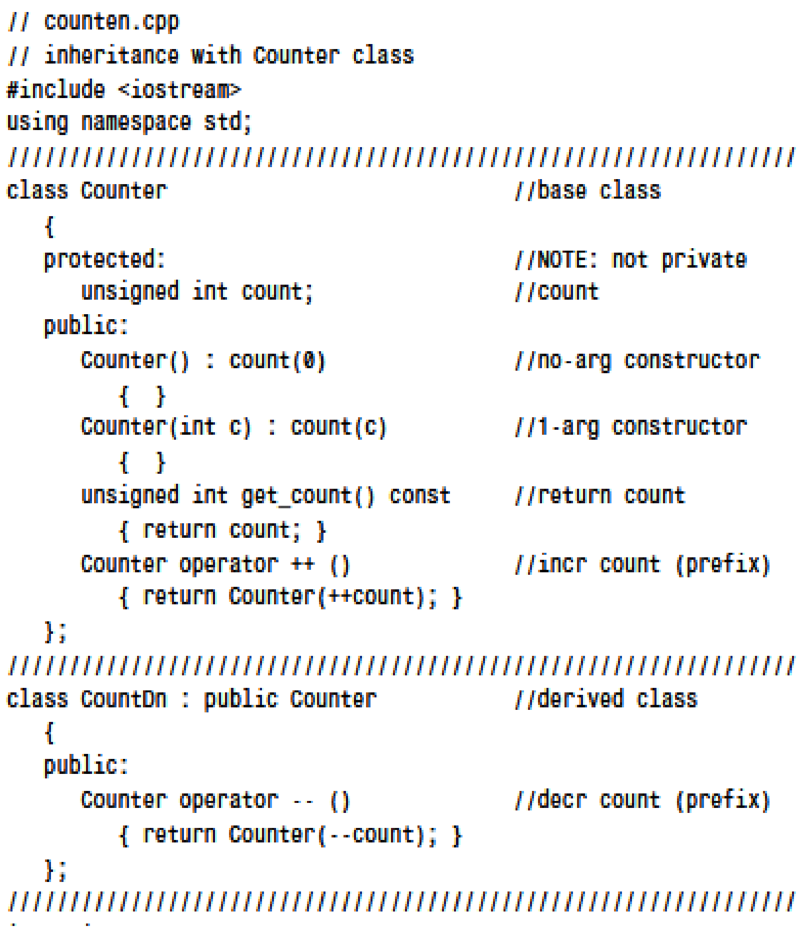
Derived Class and Base Class

Remember the COUNTPP3 example from **Chapter , "Operator Overloading"?** This program used a class Counter as a general-purpose counter variable. A count could be initialized to 0 or to a specified number with constructors, incremented with the ++ operator, and read with the get_count() operator.

Let's suppose that we have worked long and hard to make the Counter class operate just the way we want, and we're pleased with the results, except for one thing. We really need a way to decrement the count. Perhaps we're counting people entering a bank, and we want to increment the count when they come in and decrement it when they go out, so that the count represents the number of people in the bank at any moment.

We could insert a decrement routine directly into the source code of the Counter class. However, there are several reasons that we might not want to do this. First, the Counter class works very well and has undergone many hours of testing and debugging. (Of course that’s an exaggeration in this case, but it would be true in a larger and more complex class.) If we start fooling around with the source code for Counter, the testing process will need to be carried out again, and of course we may foul something up and spend hours debugging code that worked fine before we modified it.

To avoid these problems we can use inheritance to create a new class based on Counter, without modifying Counter itself. Here’s the listing for COUNTEN, which includes a new class, CountDn, that adds a decrement operator to the Counter class:



```
// counten.cpp
// inheritance with Counter class
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Counter                           //base class
   {
   protected:                           //NOTE: not private
      unsigned int count;               //count
   public:
      Counter() : count(0)              //no-arg constructor
         {  }
      Counter(int c) : count(c)         //1-arg constructor
         {  }
      unsigned int get_count() const    //return count
         { return count; }
      Counter operator ++ ()            //incr count (prefix)
         { return Counter(++count); }
   };
////////////////////////////////////////////////////////////////
class CountDn : public Counter          //derived class
   {
   public:
      Counter operator -- ()            //decr count (prefix)
         { return Counter(--count); }
   };
////////////////////////////////////////////////////////////////
```

```
int main()
   {
   CountDn c1;                         //c1 of class CountDn

   cout << "\nc1=" << c1.get_count();   //display c1

   ++c1; ++c1; ++c1;                   //increment c1, 3 times
   cout << "\nc1=" << c1.get_count();   //display it

   --c1; --c1;                         //decrement c1, twice
   cout << "\nc1=" << c1.get_count();   //display it
   cout << endl;
   return 0;
   }
```

The listing starts off with the Counter class, which (with one small exception, which we'll look at later) has not changed since its appearance in COUNTPP3. Notice that, for simplicity, we haven't modeled this program on the POSTFIX program, which incorporated the second overloaded ++ operator to provide postfix notation.

Specifying the Derived Class

Following the Counter class in the listing is the specification for a new class, CountDn. This class incorporates a new function, operator--(), which decrements the count. However—and here's the key point—the new CountDn class inherits all the features of the Counter class.

CountDn doesn't need a constructor or the get_count() or operator++() functions, because these already exist in Counter.

The first line of CountDn specifies that it is derived from Counter:-

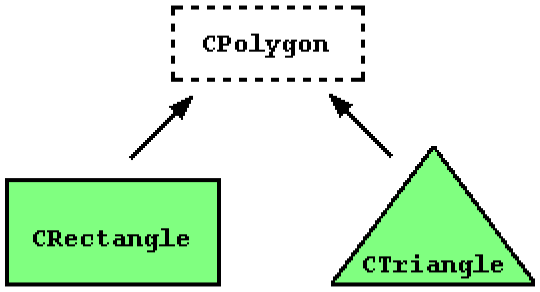
```
class CountDn : public Counter
```

Here we use a single colon (not the double colon used for the scope resolution operator),followed by the keyword public and the name of the base class Counter. This sets up the relationship between the classes. This line says that CountDn is derived from the base class Counter.

Example:

let's imagine a series of classes to describe two kinds of polygons: rectangles and triangles. These two polygons have certain common properties, such as the values needed to calculate their areas: they both can be described simply with a height and a width (or base).

This could be represented in the world of classes with a class Polygon from which we would derive the two other ones: Rectangle and Triangle:



The Polygon class would contain members that are common for both types of polygon. In our case: width and height.
And Rectangle and Triangle would be its derived classes, with specific features that are different from one type of polygon to the other.

Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive a class from it with another member called B, the derived class will contain both member A and member B.

The inheritance relationship of two classes is declared in the derived class. Derived classes definitions use the following syntax:

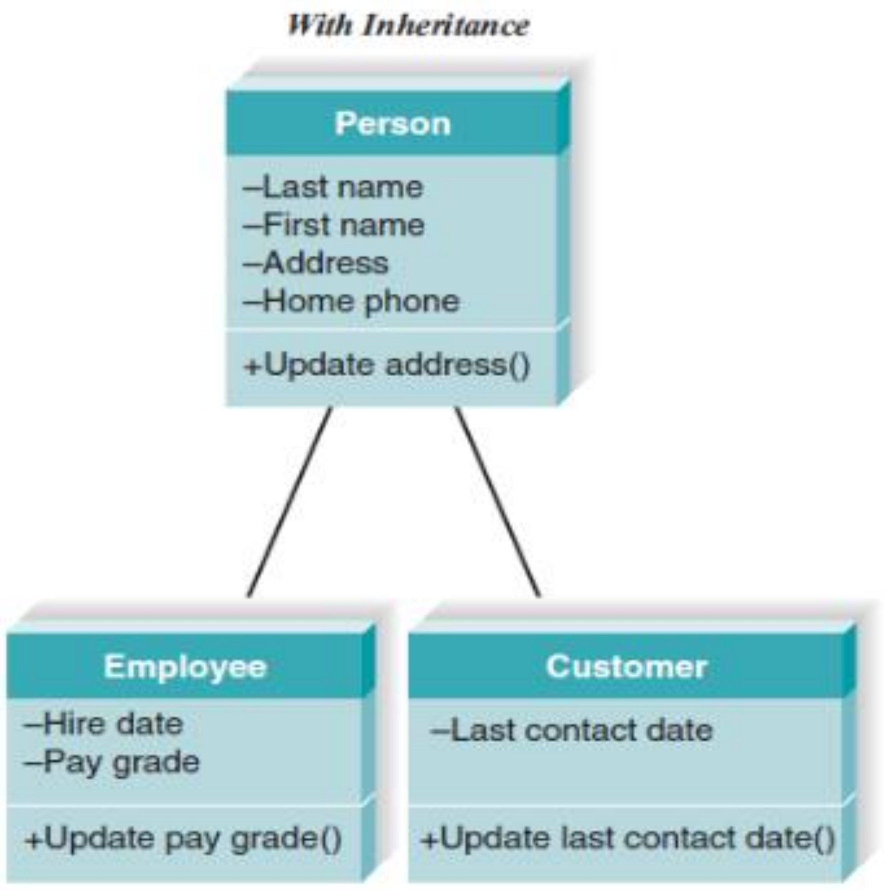
```
class derived_class_name: public base_class_name
{ /*...*/ };
```

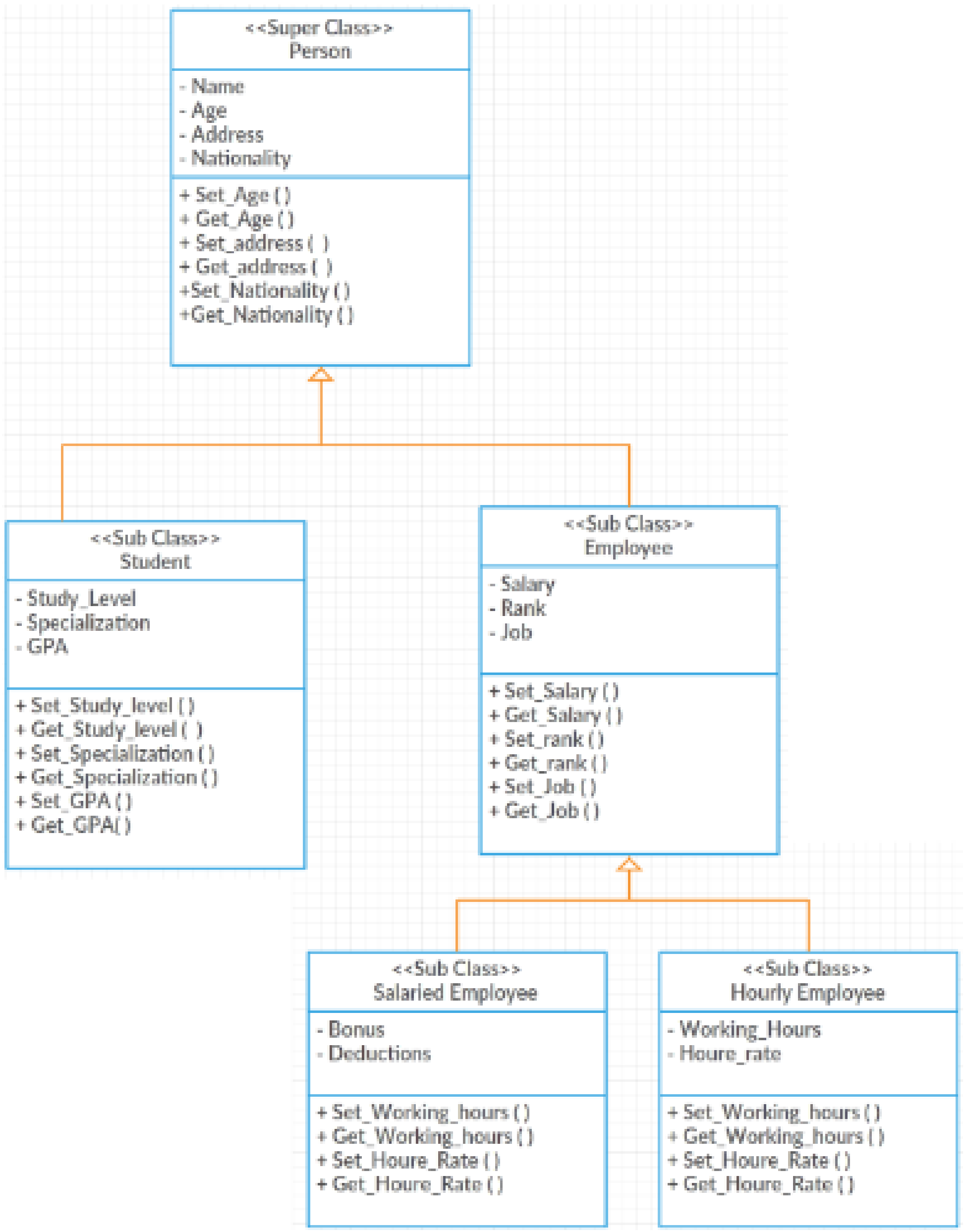
Where derived_class_name is the name of the derived class and base_class_name is the name of the class on which it is based. The public access specifier may be replaced by any one of the other access specifiers (protected or private). This access specifier limits the most accessible level for the members inherited from the base class: The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

```
// derived classes
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

The objects of the classes Rectangle and Triangle each contain members inherited from Polygon. These are: width, height and set_values.

Example(2)





The “is a” Relationship

- The relationship between a Base Class and an derived class is called an “is
  a” relationship.
  – A post graduate student “is a” Student.
  – An Employee “is a” Person.
  – Salaried Employee “is a” Employee.
  – A car “is a” vehicle
- A specialized object has:

- all of the characteristics of the general object, plus
– additional characteristics that make it special

- In object-oriented programming, inheritance is used to create an “is a” relationship among classes.

Generalization in UML Class Diagrams

In the UML, inheritance is called **generalization**, because the parent class is a more general form of the child class. Or to put it another way, the child is more specific version of the parent. The generalization in the COUNTEN program is shown in Figure 4.2

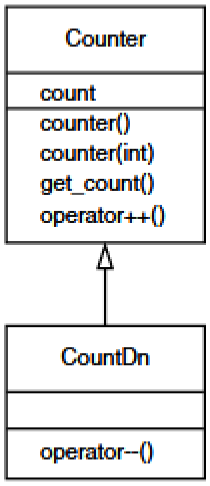


Figure 4.2 UML class diagram for COUNTEN.

In UML class diagrams, generalization is indicated by a triangular arrowhead on the line connecting the parent and child classes. Remember that the arrow means inherited from or derived from or is a more specific version of. The direction of the arrow emphasizes that the derived class refers to functions and data in the base class, while the base class has no access to the derived class.

Notice that we’ve added attributes (member data) and operations (member functions) to the classes in the diagram. The top area holds the class title, the middle area holds attributes, and the bottom area is for operations.

Accessing Base Class Members

An important topic in inheritance is knowing when a member function in the base class can be used by objects of the derived class. This is called accessibility. Let's see how the compiler handles the accessibility issue in the COUNTEN examp**le.**

4.5 Substituting Base Class Constructors

In the main() part of COUNTEN we create an object of class CountDn:

```
CountDn c1;
```

This causes c1 to be created as an object of class CountDn and initialized to 0. But wait—how is this possible? There is no constructor in the CountDn class specifier, so what entity carries out the initialization? It turns out that—at least under certain circumstances—if you don't specify a constructor, the derived class will use an appropriate constructor from the base class. In COUNTEN there's no constructor in CountDn, so the compiler uses the no-argument constructor from Count.

This flexibility on the part of the compiler—using one function because another isn't available—appears regularly in inheritance situations. Generally, the substitution is what you want, but sometimes it can be unnerving.

Substituting Base Class Member Functions

The object c1 of the CountDn class also uses the operator++() and get_count() functions from the Counter class. The first is used to increment c1:

```
++c1;
```

The second is used to display the count in **c1:**

```
cout << "\nc1=" << c1.get_count();
```

Again the compiler, not finding these functions in the class of which c1 is a member, uses member functions from the base class.

Output of COUNTEN:-

In main() we increment c1 three times, print out the resulting value, decrement c1 twice, and finally print out its value again. Here's the output**:-**

```
c1=0     <———— after initialization
c1=3     <———— after ++c1, ++c1, ++c1
c1=1     <———— after --c1, --c1
```

The ++ operator, the constructors, the get_count() function in the Counter class, and the -- operator in the CountDn class all work with objects of type CountDn**.**

4.7 The protected Access Specifier

We have increased the functionality of a class without modifying it. Well, almost without modifying it. Let's look at the single change we made to the Counter cl**ass.**

The data in the classes we've looked at so far, including count in the Counter class in the earlier COUNTPP3 program, have used the private access specifie**r**

In the Counter class in COUNTEN, count is given a new specifier: protected. What does this do?

Let's first review what we know about the access specifiers private and public. A member function of a class can always access class members, whether they are public or private. But an object declared externally can only invoke (using the dot operator, for example) public members of the class. It's not allowed to use private members. For instance, suppose an object objA is an instance of class A, and function funcA() is a member function of A. Then in main() (or any other function that is not a member of A) the statemen**t.**

```
objA.funcA();
```

will not be legal unless funcA() is public. The object objA cannot invoke private members of class A. Private members are, well, private. This is shown in Figure 5.3.
This is all we need to know if we don't use inheritance. With inheritance, however, there is a whole raft of additional possibilities. The question that

concerns us at the moment is, can member functions of the derived class access members of the base class? In other words, can operator--() in CountDn access count in Counter? The answer is that member functions can access members of the base class if the members are public, or if they are protected. They can't access private members.

We don't want to make count public, since that would allow it to be accessed by any function anywhere in the program and eliminate the advantages of data hiding. A protected member, on the other hand, can be accessed by member functions in its own class or—and here's the key—in any class derived from its own class. It can't be accessed from functions outside these classes, such as main(). This is just what we want. The situation is shown in Figure4.3

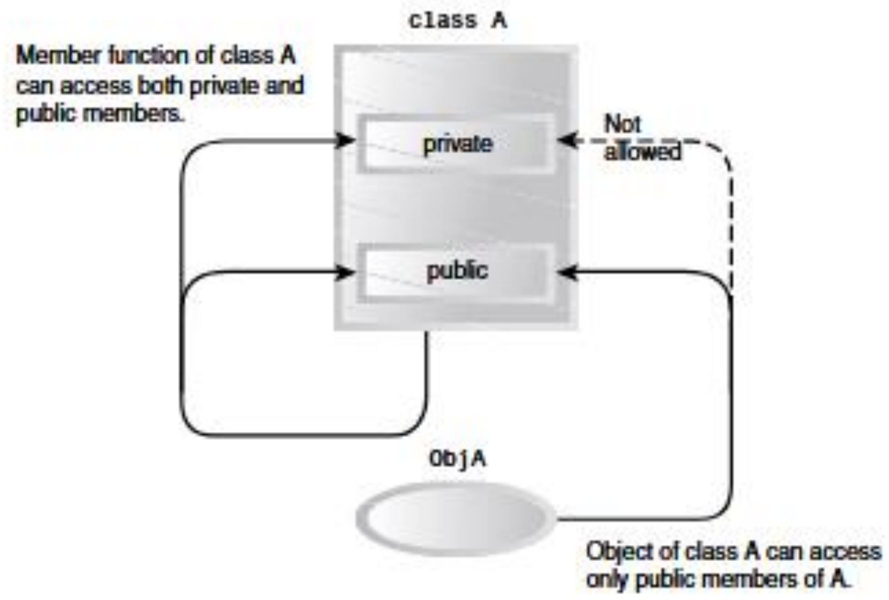

class A
Member function of class A can access both private and public members.
private
Not allowed
public
ObjA
Object of class A can access only public members of A.

Figure 4.3 Access specifiers without inheritance

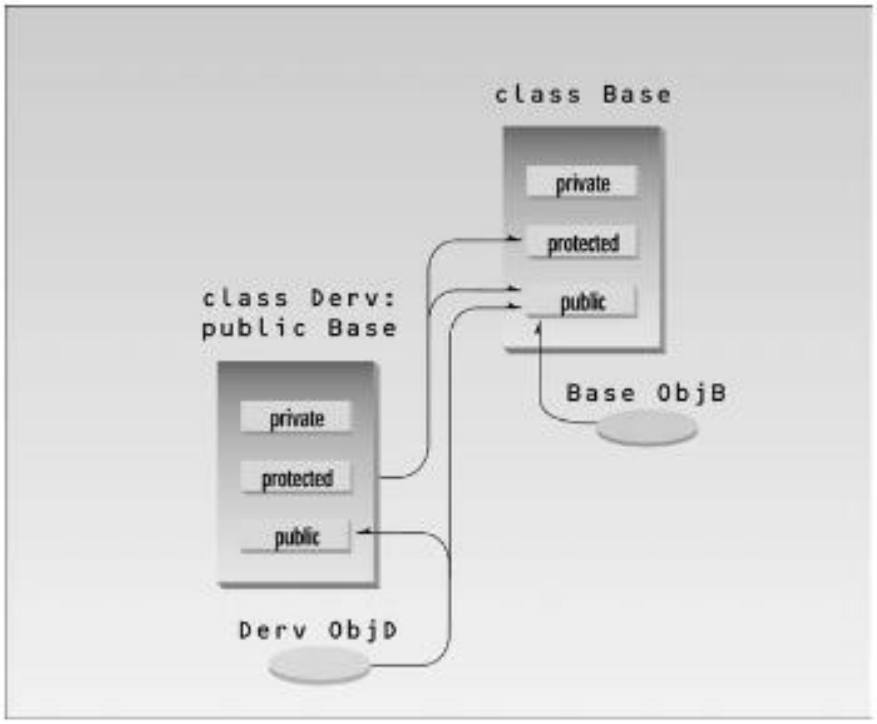


Figure 4.4 Access specifiers with inheritance.

Access Specifier	*Accessible from Own Class*	*Accessible from Derived Class*	*Accessible from Objects Outside Class*
public	yes	yes	yes
protected	yes	yes	no
private	yes	no	no

Table 4.1 Inheritance and Accessibility

Modes of Inheritance

# Public mode

If we derive a child class from a public parent class. Then the public member of the parent class becomes a public member for the child class and protected members of parents class becomes protected members of the child class

# Private mode:-

If we derive a child class from a private base class, then the public , as well as protected members, become private for the derived class.

Private members of a base class cannot be directly accessed in the derived class in any circumstance.

# protected mode

If we derive child class from a protected base class, then the public , as well as a protected member of the parent class, becomes the protected member of the child class

Order of Constructor Call with Inheritance in C++

Whether derived class's default constructor is called or parameterized is called, base class's default constructor is always called inside them**.**

To call base class's parameterized constructor inside derived class's parameterized constructor, we must mention it explicitly while declaring derived class's parameterized constructor

Function Overriding:-

- It is the redefinition of base class function in its derived class with same signature.

```
Class a
{
public:
      virtual void display(){ cout << "hello"; }
}

Class b:public a
{
public:
       void display(){ cout << "bye";};
}
```

Function Overloading:-

- It provides multiple definitions of the function by changing signature i.e changing number of parameters, change data type of parameters.
- It can be done in base as well as derived clas**s.**

Example:

```
void area(int a);
void area(int a, int b);
```

Class Hierarchies:-

n the examples so far in this chapter, inheritance has been used to add functionality to an existing class. Now let’s look at an example where inheritance is used for a different purpose: as part of the original design of a program.
Our example models a database of employees of a widget company. We’ve simplified the situation so that only three kinds of employees are represented. Managers manage, scientists perform research to develop better widgets, and laborers operate the dangerous widget-stamping presses.

The database stores a name and an employee identification number for all employees, no matter what their category. However, for managers, it also stores their titles and golf club dues. For scientists, it stores the number of scholarly articles they have published. Laborers need no additional data beyond their names and numbers.

Our example program starts with a base class employee. This class handles the employee's last name and employee number. From this class three other classes are derived: manager, scientist, and laborer. The manager and scientist classes contain additional information about these categories of employee, and member functions to handle this information, as shown in Figure 5.5

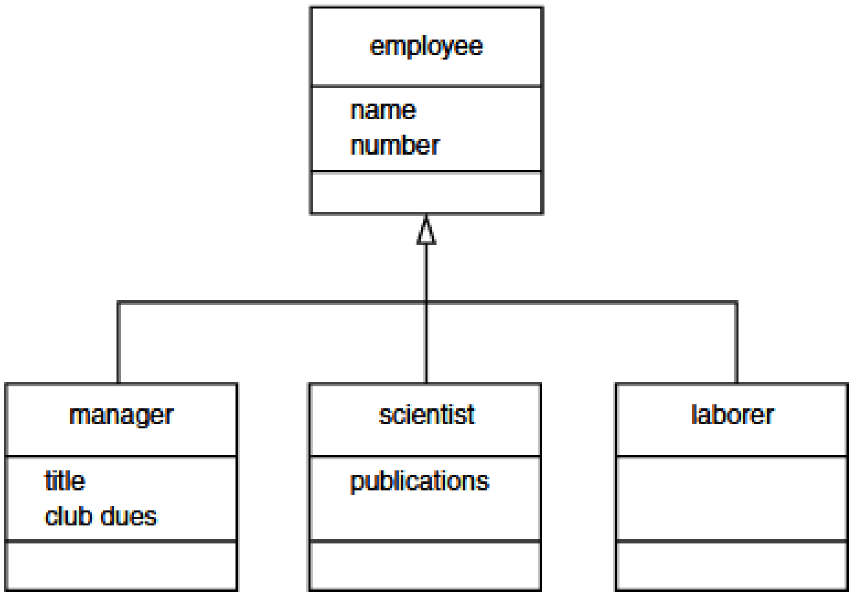


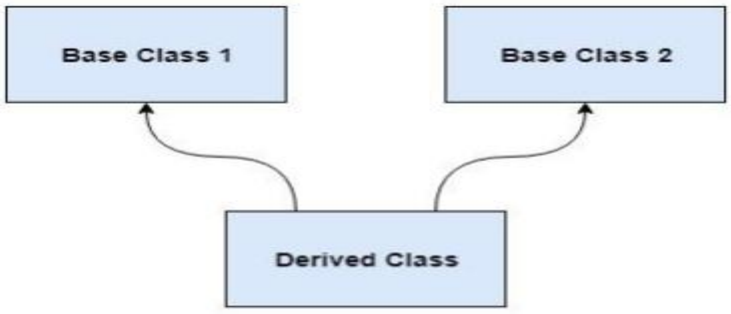
Figure4.5 UML class diagram for EMPLOY

```cpp
// employ.cpp
// models employee database using inheritance
#include <iostream>
using namespace std;
const int LEN = 80;                //maximum length of names
////////////////////////////////////////////////////////////////
class employee                     //employee class
   {
   private:
      char name[LEN];              //employee name
      unsigned long number;        //employee number
   public:
      void getdata()
         {
         cout << "\n   Enter last name: "; cin >> name;
         cout << "   Enter number: ";      cin >> number;
         }
      void putdata() const
         {
         cout << "\n   Name: " << name;
         cout << "\n   Number: " << number;
         }
   };
////////////////////////////////////////////////////////////////
class manager : public employee    //management class
   {
   private:
      char title[LEN];             //"vice-president" etc.
      double dues;                 //golf club dues
   public:
      void getdata()
         {
         employee::getdata();
         cout << "   Enter title: ";          cin >> title;
         cout << "   Enter golf club dues: "; cin >> dues;
         }
      void putdata() const
         {
         employee::putdata();
         cout << "\n   Title: " << title;
         cout << "\n   Golf club dues: " << dues;
         }
   };
////////////////////////////////////////////////////////////////
```

```
class scientist : public employee  //scientist class
   {
   private:
      int pubs;                    //number of publications
   public:
      void getdata()
         {
         employee::getdata();
         cout << "   Enter number of pubs: "; cin >> pubs;
         }
      void putdata() const
         {
         employee::putdata();
         cout << "\n   Number of publications: " << pubs;
         }
   };
////////////////////////////////////////////////////////////////
class laborer : public employee    //laborer class
   {



   };
////////////////////////////////////////////////////////////////
int main()
   {
   manager m1, m2;
   scientist s1;
   laborer l1;

   cout << endl;            //get data for several employees
   cout << "\nEnter data for manager 1";
   m1.getdata();

   cout << "\nEnter data for manager 2";
   m2.getdata();

   cout << "\nEnter data for scientist 1";
   s1.getdata();

   cout << "\nEnter data for laborer 1";
   l1.getdata();
                            //display data for several employees
   cout << "\nData on manager 1";
   m1.putdata();

   cout << "\nData on manager 2";
   m2.putdata();

   cout << "\nData on scientist 1";
   s1.putdata();

   cout << "\nData on laborer 1";
   l1.putdata();
   cout << endl;
   return 0;
   }
```

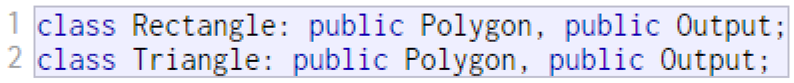
Multiple Inheritance in C++:-

- Multiple inheritance occurs when a class inherits from more than one base class. So the class can inherit features from multiple base classes in the same tim**e.**
- Unlike other object oriented programming languages, C++ allow this important features to programmers.



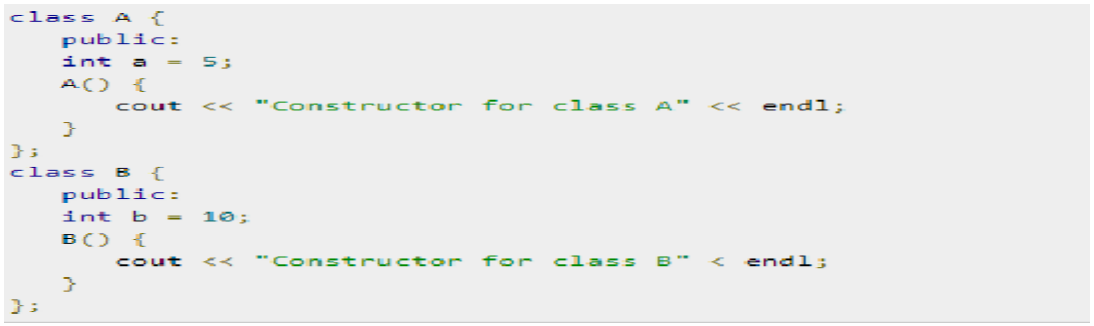
Multiple Inheritance

- For example, if the program had a specific class to print on screen called Output, and we wanted our classes Rectangle and Triangle to also inherit its members in addition to those of Polygon we could write

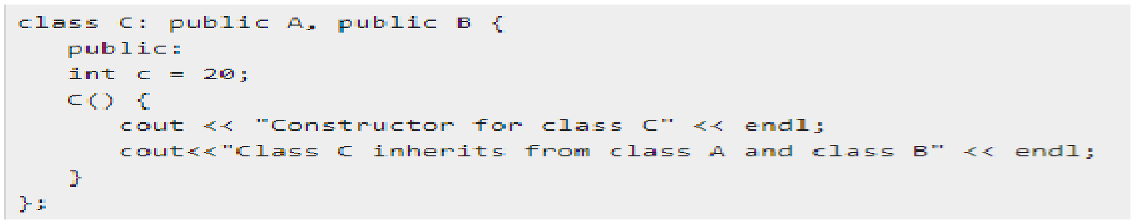


```
class Rectangle: public Polygon, public Output;
class Triangle: public Polygon, public Output;
```

Example:-



```
class A {
    public:
    int a = 5;
    A() {
        cout << "Constructor for class A" << endl;
    }
};
class B {
    public:
    int b = 10;
    B() {
        cout << "Constructor for class B" < endl;
    }
};
```



```
class C: public A, public B {
    public:
    int c = 20;
    C() {
        cout << "Constructor for class C" << endl;
        cout<<"Class C inherits from class A and class B" << endl;
    }
};
```

Example:

```
// multiple inheritance
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
};

class Output {
  public:
    static void print (int i);
};

void Output::print (int i) {
  cout << i << '\n';
}

class Rectangle: public Polygon, public Output {
  public:
    Rectangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height; }
};

class Triangle: public Polygon, public Output {
  public:
    Triangle (int a, int b) : Polygon(a,b) {}
    int area ()
      { return width*height/2; }
};

int main () {
  Rectangle rect (4,5);
  Triangle trgl (4,5);
  rect.print (rect.area());
  Triangle::print (trgl.area());
  return 0;
}
```

Review Questions

1. Inheritance is a way to

a. make general classes into more specific classes.

b. pass arguments to objects of classes.

c. add features to existing classes without rewriting them.

d. improve data hiding and encapsulation.

2. A “child” class is said to be _________ from a base class.

3. Advantages of inheritance include

a. providing class growth through natural selection.

b. facilitating class libraries.

c. avoiding the rewriting of code.

d. providing a useful conceptual framework.

4. Write the first line of the specifier for a class Bosworth that is publicly derived from a

class Alphonso.

5. True or false: Adding a derived class to a base class requires fundamental changes to the

base class.

6. To be accessed from a member function of the derived class, data or functions in the base

class must be public or _________.

7. If a base class contains a member function basefunc(), and a derived class does not contain

a function with this name, can an object of the derived class access basefunc()?

8. Assume that the classes mentioned in Question 4 and the class Alphonso contain a member

function called alfunc(). Write a statement that allows object BosworthObj of class Bosworth to access alfunc()

Exercises

(1) Create two classes named Mammals and MarineAnimals. Create another class named BlueWhale which inherits both the above classes. Now, create a function in each of these classes which prints "I am mammal", "I am a marine animal" and "I belong to both the categories: Mammals as well as Marine Animals" respectively. Now, create an object for each of the above class and try calling

A. function of Mammals by the object of Mammal

B. function of MarineAnimal by the object of MarineAnimal

C. function of BlueWhale by the object of BlueWhale

D. function of each of its parent by the object of BlueWhale

(2) We want to calculate the total marks of each student of a class in Physics,Chemistry and Mathematics and the average marks of the class. The number of students in the class are entered by the user. Create a class named Marks with data members for roll number, name and marks. Create three other classes inheriting the Marks class, namely Physics, Chemistry and Mathematics, which are used to define marks in individual subject of each student. Roll number of each student will be generated automatically.

(3) All the banks operating in India are controlled by RBI. RBI has set a well defined guideline (e.g. minimum interest rate, minimum balance allowed, maximum withdrawal limit etc) which all banks must follow. For example, suppose RBI has set minimum interest rate applicable to a saving bank account to be 4% annually; however, banks are free to use 4% interest rate or to set any rates above it.
Write a program to implement bank functionality in the above scenario. Note: Create few classes namely Customer, Account, RBI (Base Class) and few derived classes (SBI, ICICI, PNB etc). Assume and implement required member variables and functions in each class.

Chapter 5
Pointers

- Addresses and Pointers

- The Address-of Operators &

- Pointer Variables-

- Syntax Quibbles:-

- Pointers Must Have a Value

- Accessing the Variable Pointed To

Pointers are the hobgoblin of C++ (and C) programming; seldom has such a simple idea inspired so much perplexity for so many. But fear not. In this chapter we will try to demystify pointers and show practical uses for them in C++ programming.

What are pointers for?

Here are some common uses:

- Accessing array elements
- Passing arguments to a function when the function needs to modify the original argument
- Passing arrays and strings to functions
- Obtaining memory from the system
- Creating data structures such as linked lists

Pointers are an important feature of C++ (and C), while many other languages, such as Visual Basic and Java, have no pointers at all. (Java has references, which are sort of watered-down pointers.) Is this emphasis on pointers really necessary? You can do a lot without them, as their

absence from the preceding chapters demonstrates. Some operations that use pointers in C++ can be carried out in other ways. For example, array elements can be accessed with array notation rather than pointer notation (we'll see the difference soon), and a function can modify arguments passed by reference, as well as those passed by pointers.

Addresses and Pointers:-

The ideas behind pointers are not complicated. Here's the first key concept: Every byte in the computer's memory has an address. Addresses are numbers, just as they are for houses on a street. The numbers start at 0 and go up from there—1, 2, 3, and so on. If you have 1MB of memory, the highest address is 1,048,575. (Of course you have much more.)

Your program, when it is loaded into memory, occupies a certain range of these addresses. That means that every variable and every function in your program starts at a particular address. Figure 5.1 shows how this looks.

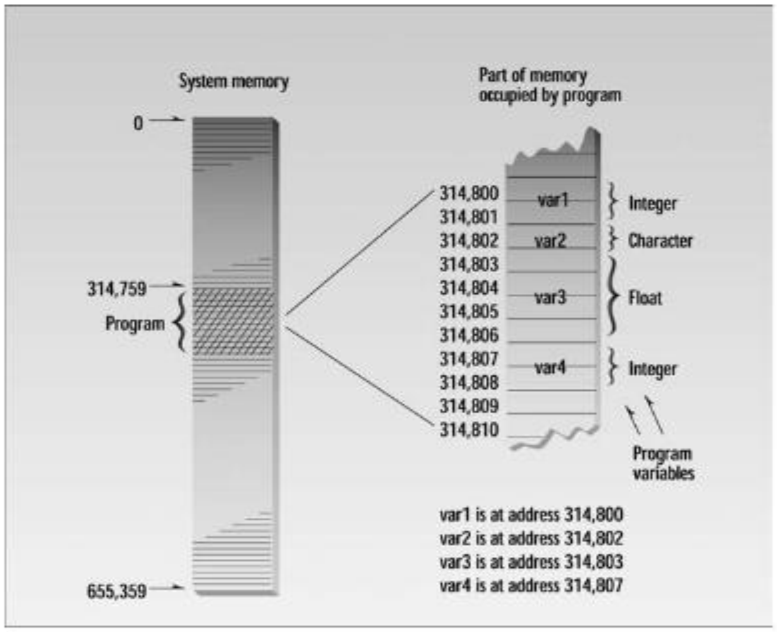


Figure 5.1 Memory addresses.

The Address-of Operator &:-

You can find the address occupied by a variable by using the address-of operator &. Here's a short program, VARADDR, that demonstrates how to do this:

```
// varaddr.cpp
// addresses of variables
#include <iostream>
using namespace std;

int main()
   {
   int var1 = 11;           //define and initialize
   int var2 = 22;           //three variables
   int var3 = 33;




cout << &var1 << endl    //print the addresses
     << &var2 << endl    //of these variables
     << &var3 << endl;
return 0;
}
```

This simple program defines three integer variables and initializes them to the values 11, 22, and 33. It then prints out the addresses of these variables.

The actual addresses occupied by the variables in a program depend on many factors, such as the computer the program is running on, the size of the operating system, and whether any other programs are currently in memory. For these reasons you probably won't get the same addresses we did when you run this program. (You may not even get the same results twice a row.) Here's the output on our machine

```
0x8f4ffff4    ←———— address of var1
0x8f4ffff2    ←———— address of var2
0x8f4ffff0    ←———— address of var3
```

Remember that the address of a variable is not at all the same as its contents. The contents of the three variables are 11, 22, and 33. Figure 5.2 shows the three variables in memory.

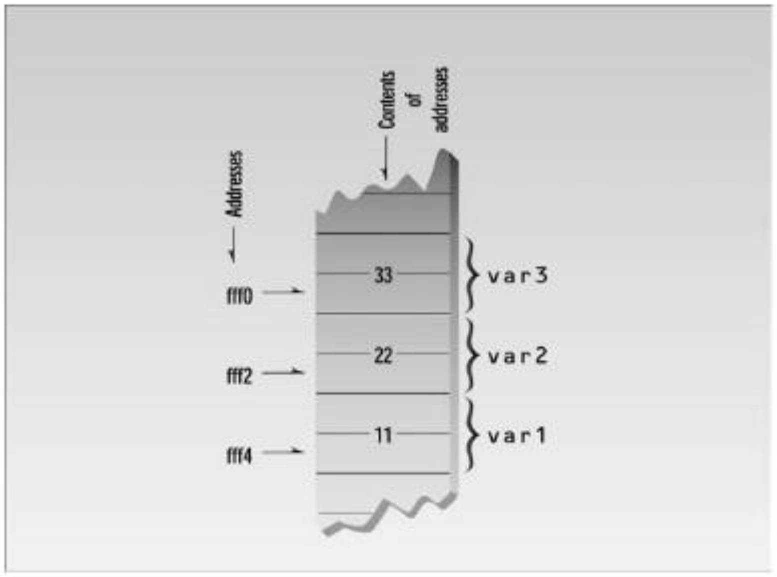


Figure 5.2 Addresses and contents of variables.

The << insertion operator interprets the addresses in hexadecimal arithmetic, as indicated by the prefix 0x before each number. This is the usual way to show memory addresses. If you aren't familiar with the hexadecimal number system, don't worry. All you really need to know is that each variable starts at a unique address. However, you might note in the output that each address differs from the next by exactly 2 bytes. That's because integers occupy 2 bytes of memory (on a 16-bit system). If we had used variables of type char, they would have adjacent addresses, since a char occupies 1 byte; and if we had used type double, the addresses would have differed by 8 bytes.

The addresses appear in descending order because local variables are stored on the stack, which grows downward in memory. If we had used global variables, they would have ascending addresses, since global variables are stored on the heap, which grows upward. Again, you don't need to worry too much about these considerations, since the compiler keeps track of the details for you.

Don't confuse the address-of operator &, which precedes a variable name in a variable declaration, with the reference operator &, which follows the type name in a function prototype or definition.

Pointer Variables:-

Addresses by themselves are rather limited. It's nice to know that we can find out where things are in memory, as we did in VARADDR, but printing out address values is not all that useful. The potential for increasing our programming power requires an additional idea: variables that hold address values. We've seen variable types that store characters, integers, floating-point numbers, and so on. Addresses are stored similarly. A variable that holds an address value is called a pointer variable, or simply a pointer.

What is the data type of pointer variables? It's not the same as the variable whose address is being stored; a pointer to int is not type int. You might think a pointer data type would be called something like pointer or ptr. However, things are slightly more complicated. The next program, PTRVAR , shows the syntax for pointer variables

```
// ptrvar.cpp
// pointers (address variables)
#include <iostream>
using namespace std;

int main()
   {
   int var1 = 11;              //two integer variables
   int var2 = 22;



   cout << &var1 << endl       //print addresses of variables
        << &var2 << endl << endl;

   int* ptr;                   //pointer to integers

   ptr = &var1;                //pointer points to var1
   cout << ptr << endl;        //print pointer value

   ptr = &var2;                //pointer points to var2
   cout << ptr << endl;        //print pointer value
   return 0;
   }
```

This program defines two integer variables, var1 and var2, and initializes them to the values 11 and 22. It then prints out their addresses.

The program next defines a pointer variable in the line int* ptr;

To the uninitiated this is a rather bizarre syntax. The asterisk means pointer to. Thus the statement defines the variable ptr as a pointer to int. This is another way of saying that this variable can hold the addresses of integer variables.

What's wrong with the idea of a general-purpose pointer type that holds pointers to any data type? If we called it type pointer we could write declarations like

pointer ptr;

The problem is that the compiler needs to know what kind of variable the pointer points to.

(We'll see why when we talk about pointers and arrays.) The syntax used in C++ allows pointers to any type to be declared

```
char* cptr;          // pointer to char
int* iptr;           // pointer to int
float* fptr;         // pointer to float
Distance* distptr;   // pointer to user-defined Distance class
```

and so on.

Syntax Quibbles:-

We should note that it is common to write pointer definitions with the asterisk closer to the variable name than to the type.

char *charptr;

It doesn't matter to the compiler, but placing the asterisk next to the type helps emphasize that the asterisk is part of the variable type (pointer to char), not part of the name itself.

If you define more than one pointer of the same type on one line, you need only insert the type-pointed-to once, but you need to place an asterisk before each variable name.

char* ptr1, * ptr2, * ptr3; // three variables of type char*

Or you can use the asterisk-next-to-the-name approach.

char *ptr1, *ptr2, *ptr3; // three variables of type char*

Pointers Must Have a Value:-

An address like 0x8f4ffff4 can be thought of as a pointer constant. A pointer like ptr can be thought of as a pointer variable. Just as the integer variable var1 can be assigned the constant value 11, so can the pointer variable ptr be assigned the constant value 0x8f4ffff4.

When we first define a variable, it holds no value (unless we initialize it at the same time). It may hold a garbage value, but this has no meaning. In the case of pointers, a garbage value is the address of something in memory, but probably not of something that we want. So before a pointer is used, a specific address must be placed in it. In the PTRVAR program, ptr is first assigned the address of var1 in the line:

```
ptr = &var1;      ←——— put address of var1 in ptr
```

Following this, the program prints out the value contained in ptr, which should be the same address printed for &var1. The same pointer variable ptr is then assigned the address of var2, and this value is printed out. Figure 5.3 shows the operation of the PTRVAR program. Here’s the output of PTRVAR

```
0x8f51fff4    ←——— address of var1
0x8f51fff2    ←——— address of var2

0x8f51fff4    ←——— ptr set to address of var1
0x8f51fff2    ←——— ptr set to address of var2
```

To summarize: A pointer can hold the address of any variable of the correct type; it’s a receptacle awaiting an address. However, it must be given some value, or it will point to an address we don’t want it to point to, such as into our program code or the operating system. Rogue pointer values can result in system crashes and are difficult to debug, since the compiler gives no warning. The moral: Make sure you give every pointer variable a valid address value before using it

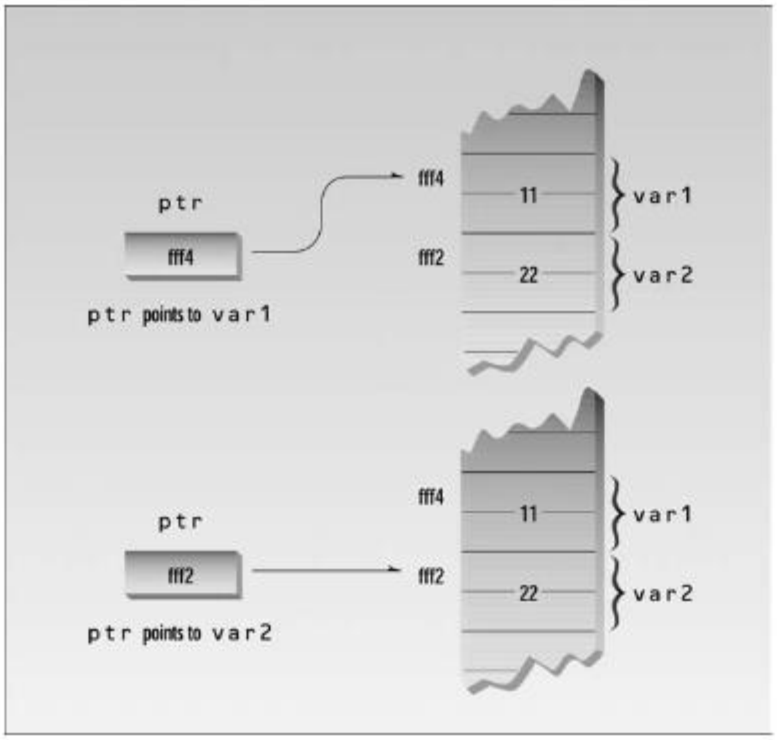


Figure 5.3 Changing values in ptr

Accessing the Variable Pointed To:-

Suppose that we don’t know the name of a variable but we do know its address. Can we access the contents of the variable? (It may seem like mismanagement to lose track of variable names, but we’ll soon see that there are many variables whose names we don’t know.)

There is a special syntax to access the value of a variable using its address instead of its name.

Here’s an example program, PTRACC, that shows how it’s done:

```
// ptracc.cpp
// accessing the variable pointed to
#include <iostream>
using namespace std;

int main()
   {
   int var1 = 11;              //two integer variables
   int var2 = 22;
```

```
int* ptr;                  //pointer to integers

ptr = &var1;               //pointer points to var1
cout << *ptr << endl;      //print contents of pointer (11)

ptr = &var2;               //pointer points to var2
cout << *ptr << endl;      //print contents of pointer (22)
return 0;
}
```

This program is very similar to PTRVAR , except that instead of printing the address values in ptr, we print the integer value stored at the address that's stored in ptr. Here's the output:

11

22

The expression that accesses the variables var1 and var2 is *ptr, which occurs in each of the two cout statements.

When an asterisk is used in front of a variable name, as it is in the *ptr expression, it is called the dereference operator (or sometimes the indirection operator). It means the value of the variable pointed to by. Thus the expression *ptr represents the value of the variable pointed to by ptr. When ptr is set to the address of var1, the expression *ptr has the value 11, since var1 is 11. When ptr is changed to the address of var2, the expression *ptr acquires the value 22, since var2 is 22. Another name for the dereference operator is the contents of operator, which is another way to say the same thing. Figure 5.4 shows how this looks.

You can use a pointer not only to display a variable's value, but also to perform any operation you would perform on the variable directly. Here's a program, PTRTO, that uses a pointer to assign a value to a variable, and then to assign that value to another variable

```
// ptrto.cpp
// other access using pointers
#include <iostream>
using namespace std;

int main()
   {
   int var1, var2;          //two integer variables
   int* ptr;                //pointer to integers

   ptr = &var1;             //set pointer to address of var1
   *ptr = 37;               //same as var1=37
   var2 = *ptr;             //same as var2=var1

   cout << var2 << endl;    //verify var2 is 37
   return 0;
   }
```

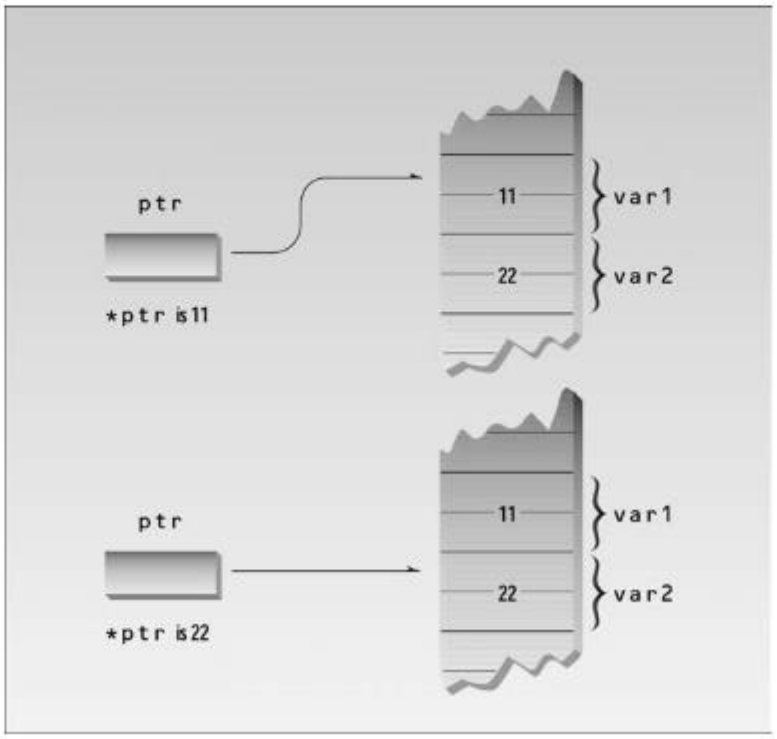


Figure 5.4 Access via pointer.

Remember that the asterisk used as the dereference operator has a different meaning than the asterisk used to declare pointer variables. The dereference operator precedes the variable and means value of the variable pointed to by. The asterisk used in a declaration means pointer to

```
int* ptr;    //declaration: pointer to int
*ptr = 37;   //indirection: value of variable pointed to by ptr
```

Using the dereference operator to access the value stored in an address is called indirect addressing, or sometimes dereferencing, the pointer

Here's a capsule summary of what we've learned so far:

```
int v;         //defines variable v of type int
int* p;        //defines p as a pointer to int
p = &v;        //assigns address of variable v to pointer p
v = 3;         //assigns 3 to v
*p = 3;        //also assigns 3 to v
```

The last two statements show the difference between normal or direct addressing, where we refer to a variable by name, and pointer or indirect addressing, where we refer to the same variable using its address.

In the example programs we've shown so far in this chapter, there's really no advantage to using the pointer expression to access variables, since we can access them directly. The value of pointers becomes evident when you can't access a variable directly.

Chapter 6
Polymorphism

- Pointer to Base Class

- Virtual members

- Abstract Base Classes

What is Polymorphism ?

- Polymorphism is an object-oriented programming concept that refers to the ability of a variable, function or object to take on multiple forms.
- with polymorphism, class objects belonging to the same hierarchical tree (inherited from a common parent class) may have functions with the same name, but with different behaviors.

Pointer to Base Class

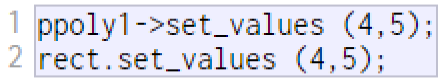
One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. *Polymorphism* is the art of taking advantage of this simple but powerful and versatile feature.

The example about the rectangle and triangle classes can be rewritten using pointers taking this feature into account:

```
// pointers to base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

Function main declares two pointers to Polygon (named ppoly1 and ppoly2). These are assigned the addresses of rect and trgl, respectively, which are objects of type Rectangle and Triangle. Such assignments are valid, since both Rectangle and Triangle are classes derived from Polygon.

Dereferencing ppoly1 and ppoly2 (with ppoly1-> and ppoly2->) is valid and allows us to access the members of their pointed objects. For example, the following two statements would be equivalent in the previous example:



```
ppoly1->set_values (4,5);
rect.set_values (4,5);
```

But because the type of both ppoly1 and ppoly2 is pointer to Polygon (and not pointer to Rectangle nor pointer to Triangle), only the members inherited from Polygon can be accessed, and not those of the derived classes Rectangle and Triangle. That is why the program above accesses the area members of both objects using rect and trgl directly, instead of the pointers; the pointers to the base class cannot access the area members.

Member area could have been accessed with the pointers to Polygon if area were a member of Polygon instead of a member of its derived classes, but the problem is that Rectangle and Triangle implement different versions of area, therefore there is not a single common version that could be implemented in the base class.

Virtual members

A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with
the virtual keyword:

```
// virtual members
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return 0; }
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

In this example, all three classes (Polygon, Rectangle and Triangle) have the same members: width, height, and functions set_values and area.

The member function area has been declared as virtual in the base class because it is later redefined in each of the derived classes. Non-virtual members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if virtual is removed from the declaration of area in the example above, all three calls to area would return zero, because in all cases, the version of the base class would have been called instead.

Therefore, essentially, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class that is pointing to an object of the derived class, as in the above example.

A class that declares or inherits a virtual function is called a *polymorphic class*.

Note that despite of the virtuality of one of its members, Polygon was a regular class, of which even an object was instantiated (poly), with its own definition of member area that always returns 0.

Abstract Base Classes

Abstract base classes are something very similar to the Polygon class in the previous example. They are classes that can only be used as base classes, and thus are allowed to have virtual member functions without definition (known as pure virtual functions). The syntax is to replace their definition by =0 (an equal sign and a zero):

An abstract base Polygon class could look like this:

```
// abstract class CPolygon
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

Notice that area has no definition; this has been replaced by =0, which makes it a *pure virtual function*. Classes that contain at least one *pure virtual function* are known as *abstract base classes*.

Abstract base classes cannot be used to instantiate objects. Therefore, this last abstract base class version of Polygon could not be used to declare objects like:

```
Polygon mypolygon;   // not working if Polygon is abstract base class
```

But an *abstract base class* is not totally useless. It can be used to create pointers to it, and take advantage of all its polymorphic abilities. For example, the following pointer declarations would be valid:

```
Polygon * ppoly1;
Polygon * ppoly2;
```

And can actually be dereferenced when pointing to objects of derived (non-abstract) classes. Here is the entire example:

```
/ / abstract base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
};
```

```
class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  return 0;
}
```

In this example, objects of different but related types are referred to using a unique type of pointer (Polygon*) and the proper member function is called every time, just because they are virtual. This can be really useful in some circumstances. For example, it is even possible for a member of the abstract base class Polygon to use the special pointer this to access the proper virtual members, even though Polygon itself has no implementation for this function:

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area() =0;
    void printarea()
      { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};

int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  return 0;
}
```

Virtual members and abstract classes grant C++ polymorphic characteristics, most useful for object-oriented projects. Of course, the examples above are very simple use cases, but these features can be applied to arrays of objects or dynamically allocated objects.

Example: (dynamic memory, constructor initializers and polymorphism:)

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    virtual int area (void) =0;
    void printarea()
      { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    Triangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height/2; }
};

int main () {
  Polygon * ppoly1 = new Rectangle (4,5);
  Polygon * ppoly2 = new Triangle (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

Notice that the ppoly pointers:

```
Polygon * ppoly1 = new Rectangle (4,5);
Polygon * ppoly2 = new Triangle (4,5);
```

are declared being of type "pointer to Polygon", but the objects allocated have been declared having the derived class type directly (Rectangle and Triangle).

Chapter 7
Virtual Functions

- **Virtual Functions**
- **Friend Function**
- **Friend Class**
- **static Function**

Now that we understand something about pointers, we can delve into more advanced C++ topics. This chapter covers a rather loosely related collection of such subjects: virtual functions, friend functions, static functions, the overloaded = operator, the overloaded copy constructor, and the this pointer. These are advanced features; they are not necessary for every C++ program, especially very short ones. However, they are widely used, and are essential for most full-size programs. Virtual functions in particular are essential for polymorphism, one of the cornerstones of object-oriented programming.

Virtual Functions

Virtual means existing in appearance but not in reality. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. Why are virtual functions needed? Suppose you have a number of objects of different classes but you want to put them all in an array and perform a particular operation on them using the same function call. For example, suppose a graphics program includes several different shapes: a triangle, a ball, a square, and so on, as in the MULTSHAP program in Chapter 4, “Inheritance.” Each of these classes has a member function draw() that causes the object to

be drawn on the screen.

Now suppose you plan to make a picture by grouping a number of these elements together, and you want to draw the picture in a convenient way. One approach is to create an array that holds pointers to all the different objects in the picture. The array might be defined like this:

```
shape* ptrarr[100];  // array of 100 pointers to shapes
```

If you insert pointers to all the shapes into this array, you can then draw an entire picture using a simple loop

```
for(int j=0; j<N; j++)
   ptrarr[j]->draw();
```

This is an amazing capability: Completely different functions are executed by the same function call. If the pointer in ptrarr points to a ball, the function that draws a ball is called; if it points to a triangle, the triangle-drawing function is called. This is called polymorphism, which means different forms. The functions have the same appearance, the draw() expression, but different actual functions are called, depending on the contents of ptrarr[j].

Polymorphism is one of the key features of object-oriented programming, after classes and inheritance.

For the polymorphic approach to work, several conditions must be met. First, all the different classes of shapes, such as balls and triangles, must be descended from a single base class (called shape in MULTSHAP). Second, the draw() function must be declared to be virtual in the base class.

Normal Member Functions Accessed with Pointers:-

Our first example shows what happens when a base class and derived classes all have functions with the same name, and you access these functions using pointers but without using virtual functions. Here's the listing for NOTVIRT

```
// notvirt.cpp
// normal functions accessed from pointer
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base                       //base class
   {
   public:
      void show()                //normal function
         { cout << "Base\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv1 : public Base        //derived class 1
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv2 : public Base        //derived class 2
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Derv1 dv1;           //object of derived class 1
   Derv2 dv2;           //object of derived class 2
   Base* ptr;           //pointer to base class

   ptr = &dv1;          //put address of dv1 in pointer
   ptr->show();         //execute show()
```

```
ptr = &dv2;          //put address of dv2 in pointer
ptr->show();         //execute show()
return 0;
}
```

The Derv1 and Derv2 classes are derived from class Base. Each of these three classes has a member function show(). In main() we create objects of class Derv1 and Derv2, and a pointer to class Base. Then we put the address of a derived class object in the base class pointer in the line

```
ptr = &dv1;  // derived class address in base class pointer
```

But wait—how can we get away with this? Doesn’t the compiler complain that we’re assigning an address of one type (Derv1) to a pointer of another (Base)? On the contrary, the compiler is perfectly happy, because type checking has been relaxed in this situation, for reasons that will become apparent soon. The rule is that pointers to objects of a derived class are type compatible with pointers to objects of the base class.

Now the question is, when you execute the line

```
ptr->show();
```

what function is called? Is it Base::show() or Derv1::show()? Again, in the last two lines of NOTVIRT we put the address of an object of class Derv2 in the pointer, and again execute

```
ptr->show();
```

Which of the show() functions is called here? The output from the program answers these questions:

```
Base
Base
```

As you can see, the function in the base class is always executed. The compiler ignores the contents of the pointer ptr and chooses the member function that matches the type of the pointer, as shown in Figure 7.1.

Sometimes this is what we want, but it doesn’t solve the problem posed at the beginning of this section: accessing objects of different classes using the same statement.

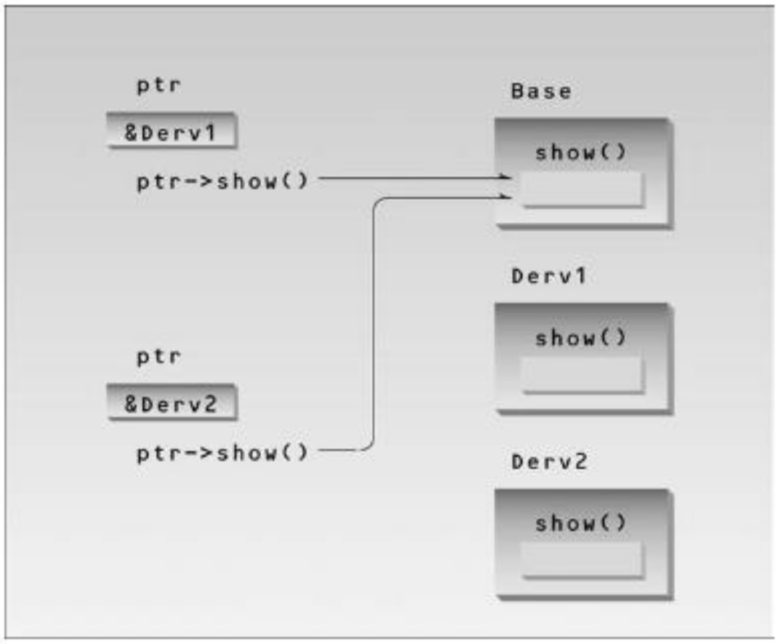


Figure 7.1 Nonvirtual pointer access.

Virtual Member Functions Accessed with Pointers:-

Let's make a single change in our program: We'll place the keyword virtual in front of the declarator for the show() function in the base class. Here's the listing for the resulting program, VIRT:

```
// virt.cpp
// virtual functions accessed from pointer
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base                          //base class
   {
   public:
      virtual void show()           //virtual function
         { cout << "Base\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv1 : public Base           //derived class 1
   {
```

```
   public:
      void show()
         { cout << "Derv1\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv2 : public Base          //derived class 2
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Derv1 dv1;             //object of derived class 1
   Derv2 dv2;             //object of derived class 2
   Base* ptr;             //pointer to base class

   ptr = &dv1;            //put address of dv1 in pointer
   ptr->show();           //execute show()

   ptr = &dv2;            //put address of dv2 in pointer
   ptr->show();           //execute show()
   return 0;
   }
```

The output of this program is

Derv1

Derv2

Now, as you can see, the member functions of the derived classes, not the base class, are executed. We change the contents of ptr from the address of

Derv1 to that of Derv2, and the particular instance of show() that is executed also changes. So the same function call.

```
ptr->show();
```

executes different functions, depending on the contents of ptr. The rule is that the compiler selects the function based on the contents of the pointer ptr, not on the type of the pointer, as in NOTVIRT. This is shown in Figure 7.2

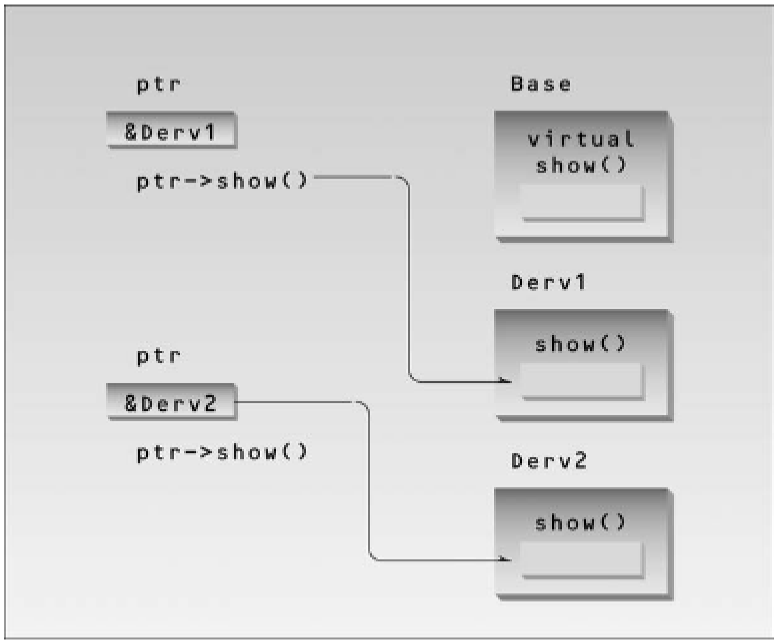


Figure 7.2 Virtual pointer access

Friend Functions

The concepts of encapsulation and data hiding dictate that nonmember functions should not be able to access an object's private or protected data. The policy is, if you're not a member, you can't get in. However, there are situations where such rigid discrimination leads to considerable inconvenience.

Friends as Bridges:

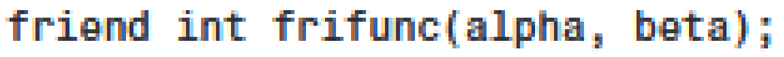
Imagine that you want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private data. In this situation there's nothing like a friend function. Here's a simple example, FRIEND, that shows how friend functions can act as a bridge between two classes

```
// friend.cpp
// friend functions
```

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class beta;              //needed for frifunc declaration

class alpha
   {
   private:
      int data;
   public:
      alpha() : data(3) {  }            //no-arg constructor
      friend int frifunc(alpha, beta);  //friend function
   };
////////////////////////////////////////////////////////////////
class beta
   {
   private:
      int data;
   public:
      beta() : data(7) {  }             //no-arg constructor
      friend int frifunc(alpha, beta);  //friend function
   };
////////////////////////////////////////////////////////////////

int frifunc(alpha a, beta b)            //function definition
   {
   return( a.data + b.data );
   }
//--------------------------------------------------------------
int main()
   {
   alpha aa;
   beta bb;

   cout << frifunc(aa, bb) << endl;     //call the function
   return 0;
   }
```

In this program, the two classes are alpha and beta. The constructors in these classes initialize their single data items to fixed values (3 in alpha and 7 in beta).

We want the function frifunc() to have access to both of these private data members, so we make it a friend function. It’s declared with the friend keyword in both classes:



```
friend int frifunc(alpha, beta);
```

This declaration can be placed anywhere in the class; it doesn’t matter whether it goes in the public or the private section

An object of each class is passed as an argument to the function frifunc(), and it accesses the private data member of both classes through these arguments. The function doesn’t do much:

It adds the data items and returns the sum. The main() program calls this function and prints the result.

A minor point: Remember that a class can’t be referred to until it has been declared. Class beta is referred to in the declaration of the function frifunc() in class alpha, so beta must be declared before alpha. Hence the declaration

class beta; at the beginning of the program.

Friend Class

The member functions of a class can all be made friends at the same time when you make the entire class a friend. The program FRICLASS shows how this looks

```
// friclass.cpp
// friend classes
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class alpha
   {
   private:
      int data1;
   public:
      alpha() : data1(99) {  }   //constructor
      friend class beta;         //beta is a friend class
   };
////////////////////////////////////////////////////////////////
class beta
   {                             //all member functions can



   public:                        //access private alpha data
      void func1(alpha a)  { cout << "\ndata1=" << a.data1; }
      void func2(alpha a)  { cout << "\ndata1=" << a.data1; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   alpha a;
   beta b;

   b.func1(a);
   b.func2(a);
   cout << endl;
   return 0;
   }
```

In class alpha the entire class beta is proclaimed a friend. Now all the member functions of beta can access the private data of alpha (in this program, the single data item data1).

Note that in the friend declaration we specify that beta is a class using the class keyword:

friend class beta;

We could have also declared beta to be a class before the alpha class specifier, as in previous examples

class beta;

and then, within alpha, referred to beta without the class keyword:

friend beta;

Static Functions

In the STATIC example in Chapter 6, “Objects and Classes,” we introduced static data members. As you may recall, a static data member is not duplicated for each object; rather a single data item is shared by all objects of a class. The STATIC example showed a class that kept track of how many objects of itself there were. Let’s extend this concept by showing how functions as well as data may be static. Besides showing static functions, our example will model a class that provides an ID number for each of its objects. This allows you to query an object to find out which object it is—a capability that is sometimes useful in debugging a program, among other situations. The program also casts some light on the operation of destructors.

Here’s the listing for STATFUNC:

```
// statfunc.cpp
// static functions and ID numbers for objects
#include <iostream>
using namespace std;



      ////////////////////////////////////////////////////////////////
      class gamma
         {
         private:
            static int total;        //total objects of this class
                                     //   (declaration only)
            int id;                  //ID number of this object
         public:
            gamma()                  //no-argument constructor
               {
               total++;              //add another object
               id = total;           //id equals current total
               }
            ~gamma()                 //destructor
               {
               total--;
               cout << "Destroying ID number " << id  << endl;
               }
            static void showtotal()  //static function
               {
               cout << "Total is " << total << endl;
               }
            void showid()            //non-static function
               {
               cout << "ID number is " << id << endl;
               }
         };
```

```
//--------------------------------------------------------------
int gamma::total = 0;           //definition of total
////////////////////////////////////////////////////////////////
int main()
   {
   gamma g1;
   gamma::showtotal();

   gamma g2, g3;
   gamma::showtotal();

   g1.showid();
   g2.showid();
   g3.showid();
   cout << "----------end of program----------\n";
   return 0;
   }
```

Accessing static Functions:

In this program there is a static data member, total, in the class gamma. This data keeps track of how many objects of the class there are. It is incremented by the constructor and decremented by the destructor.

Suppose we want to access total from outside the class. We construct a function, showtotal(), that prints the total's value. But how do we access this function?

When a data member is declared static, there is only one such data value for the entire class, no matter how many objects of the class are created. In fact, there may be no such objects at all, but we still want to be able to learn this fact. We could create a dummy object to use in calling a member function, as in

gamma dummyObj; // make an object so we can call function

dummyObj.showtotal(); // call function

But this is rather inelegant. We shouldn't need to refer to a specific object when we're doing something that relates to the entire class. It's more reasonable to use the name of the class itself with the scope-resolution operator.

gamma::showtotal(); // more reasonable

However, this won't work if showtotal() is a normal member function; an object and the dot member-access operator are required in such cases. To access showtotal() using only the class name, we must declare it to be a static member function. This is what we do in STATFUNC, in the declarator

```
static void showtotal()
```

Now the function can be accessed using only the class name. Here's the output:

```
Total is 1

Total is 3

ID number is 1

ID number is 2

ID number is 3

----------end of program--------

Destroying ID number 3

Destroying ID number 2

Destroying ID number 1
```

We define one object, g1, and then print out the value of total, which is 1. Then we define two more objects, g2 and g3, and again print out the total, which is now 3.

References

1. Object Oriented Programming in C++ by Robert Lafore Techmedia Publication.

2. The complete reference C – by Herbert shieldt Tata McGraw Hill Publication.

3. Object Oriented Programming in C++ Saurav Sahay Oxford University Press.

4. Object Oriented Programming in C++ R Rajaram New Age International Publishers 2nd .

5. OOPS C++ Big C++ Cay Horstmann Wiley Publication

6. https://www.tutorialspoint.com/cplusplus/cpp_files_streams.htm

7. https://www.programiz.com/cpp-programming