2nd Class

Object Oriented Programming

البرمجة الشيئية

أستاذة المادة: م. إخلاص فالح

Programming

- A digital computer is a useful tool for solving a great variety of **problems.**
- **A** solution to a problem is called an **algorithm; it describes the sequence of steps to** be performed for the problem to be solved.
- To make an algorithm intelligible to a computer, it needs to be expressed in a language understood by a computer (**machine language).**
- **Programs expressed in** the machine language are said to be **executable.**
- A further abstraction of machine language is the **assembly language.**
- **High-level languages** such as C++ provide a much more convenient notation for implementing algorithms.

Compiler

- A program written in a high-level language is translated to assembly language by a translator called a **compiler.**
- **The assembly** code produced by the compiler is then assembled to produce an executable program.

C++ Compiler

- Compiling a C++ program involves a number of steps :
- First, the C++ **preprocessor goes over the program text and carries out the** instructions specified by the preprocessor directives (e.g., #include). The result is a modified program text which no longer contains any directives.
- Then, the C++ **compiler translates the program code. The compiler may be a** true C++ compiler which generates native (assembly or

machine) code, or just a translator which translates the code into C. In the latter case, the resulting C code is then passed through a C compiler to produce native object code. In either case, the outcome may be incomplete due to the program referring to library routines which are not defined as a part of the program.

- Finally, the **linker completes the object code by linking it with the object code** of any library modules that the program may have referred to. The final result is an executable file.

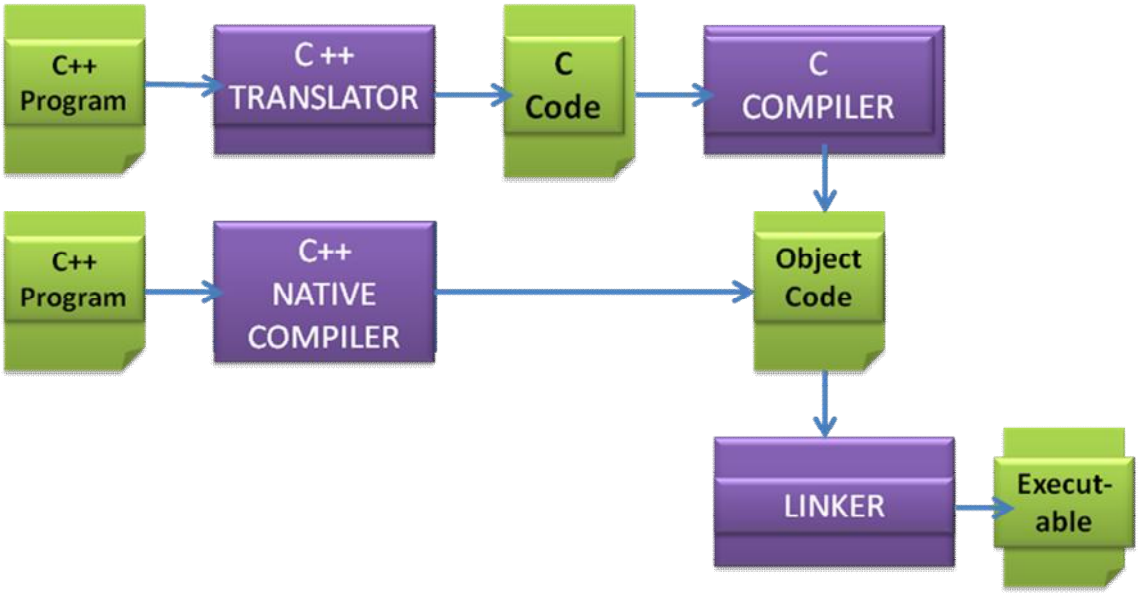


Figure1:C++ compilation

Variables

- Variable is a symbolic name for a memory location in which data can be stored and subsequently recalled.
- All variables have two important attributes:
- A **type which is established when the variable is defined (e.g., integer, real,** character). Once defined, the type of a C++ variable cannot be changed.
- A **value which can be changed by assigning a new value to the variable. The** kind of values a variable can assume depends on its type. For example, an integer variable can only take integer values (e.g., 2, 100, -12).

Simple variables

Example 1:

```
#include <iostream.h>
int main (void)
{
        int workDays;
        float workHours, payRate, weeklyPay;
        workDays = 5;
        workHours = 7.5;
        payRate = 38.55;
        weeklyPay = workDays * workHours * payRate;
        cout << "Weekly Pay = ";
        cout << weeklyPay;
        cout << '\n';
}
```

Initialization

Example 2:

```
#include <iostream.h>
int main (void)
{
        int workDays = 5;
        float workHours = 7.5;
        float payRate = 38.55;
        float weeklyPay = workDays * workHours * payRate;
        cout << "Weekly Pay = ";
        cout << weeklyPay;
        cout << '\n';
}
```

Comments

- ⦿ A comment is a piece of descriptive text which explains some aspect of a program.
- ⦿ Program comments are totally ignored by the compiler and are only intended for human readers.
- ⦿ C++ provides two types of comment delimiters:
- ❖ Anything after // (until the end of the line on which it appears) is considered a comment.
- ❖ Anything enclosed by the pair /* and */ is considered a comment.

Comments forms

Example 3:

```
#include <iostream.h>

/* This program calculates the weekly gross pay for a worker,
based on the total number of hours worked and the hourly pay rate. */

int main (void)
}
    int workDays = 5;           // Number of work days per week
    float workHours = 7.5;      // Number of work hours per day
    float payRate = 33.50;      // Hourly pay rate
    float weeklyPay;            // Gross weekly pay
    weeklyPay = workDays * workHours * payRate;
    cout << "Weekly Pay = " << weeklyPay << '\n';
{
```

Simple Type Conversion

- A value in any of the built-in types we have see so far can be converted (*type-cast)* to any of the other types.
- For example:

  (int) 3.14 // converts 3.14 to an int to give 3

  (long) 3.14 // converts 3.14 to a long to give 3L

  (double) 2 // converts 2 to a double to give 2.0

  (char) 122 // converts 122 to a char whose code is 122

  (unsigned short) 3.14 // gives 3 as an unsigned short
- In some cases, C++ also performs **implicit type conversion. This happens** when values of different types are mixed in an expression. For example:

  double d = 1; // d receives 1.0

```
int i = 10.5; // i receives 10
i = i + d; // means: i = int(double(i) + d)
```

Statements

- Simple statement
- Compound Statements
- The if Statement
- The switch Statement
- The while Statement
- The do Statement
- The for Statement
- The continue Statement
- The break Statement
- The goto Statement
- The return Statement

Simple Statements

- A simple statement is a computation terminated by a semicolon. Variable definitions and semicolon terminated expressions are examples:

```
int i; // declaration statement
++i; // this has a side-effect
double d = 10.5; // declaration statement
d + 5; // useless statement!
```

The last example represents a useless statement, because it has no side-effect (d is added to 5 and the result is just discarded).

- The simplest statement is the null statement which consists of just a semicolon:

```
;     // null statement
```

The if Statement

- t is sometimes desirable to make the execution of a statement dependent upon a condition being satisfied. The if statement provides a way of expressing this, the general form of which is:

if (*expression)*

Statement1

else

Statement2

- First *expression is evaluated. If the outcome is nonzero then statement is* executed. Otherwise, nothing happens.
- Example

```
if (count != 0)
average = sum / count;
```

The switch statement

- The switch statement provides a way of choosing between a set of alternatives, based on the value of an expression. The general form of the switch statement is:

switch (*expression) {*

case constant$_1$:

statements;

...

case constant$_n$:

statements;

default:

statements;

}

```
switch (operator) {
      case '+': result = operand1 + operand2;
      break;
      case '-': result = operand1 - operand2;
      break;
      case '*': result = operand1 * operand2;
      break;
      case '/': result = operand1 / operand2;
      break;
      default: cout << "unknown operator: " << ch <<
'\n';
      Break;
      }
```

The while Statement

The general form of the while statement is:

while (*expression)*

statement;

- First *expression (called the* ***loop condition) is evaluated. If the outcome is nonzero***
- then *statement (called the* ***loop body) is executed and the whole process is*** repeated.
- Otherwise, the loop is terminated.

The do Statement

The general form of the do statement is:

do

statement;

while (expression);

- First statement is executed and then expression is evaluated.
- If the outcome of the latter is nonzero
- Then the whole process is repeated. Otherwise, the loop is terminated.

The for Statement

The general form of the for statement is:

for (*expression1; expression2; expression3)*

statement;

- First *expression1 is evaluated. Each time round the loop, expression2 is* evaluated.
- If the outcome is nonzero then statement is executed and *expression3 is* evaluated.
- Otherwise, the loop is terminated.

The continue Statement

- The continue statement terminates the current iteration of a loop and instead jumps to the next iteration.
- It applies to the loop immediately enclosing the continue statement.
- It is an error to use the continue statement outside a loop.

```
do {
cin  >> num;
if  (num < 0) continue;
// process num here...
} while (num != 0);
```

The break Statement

- A break statement may appear inside a loop (while, do, or for) or a switch statement.
- It causes a jump out of these constructs, and hence terminates them.
- Like the continue statement, a break statement only applies to the loop or switch immediately enclosing it.
- It is an error to use the break statement outside a loop or a switch.

- For example, suppose we wish to read in a user password, but would like to allow the user a limited number of attempts:

```
for (i = 0; i < attempts; ++i)
{
cout << "Please enter your password: ";
cin >> password;
if (Verify(password)) // check password for correctness
break; // drop out of the loop
cout << "Incorrect!\n";
}
```

The goto Statement

The goto statement has the general form:

goto *label;*

- where *label is an identifier which marks the jump destination of goto. The label* should be followed by a colon and appear before a statement within the same function as the goto statement itself.
- For example

```
for (i = 0; i < attempts; ++i)
{
cout << "Please enter your password: ";
cin >> password;
if  (Verify(password)) // check password for correctness
goto out; // drop out of the loop
cout << "Incorrect!\n";
{
out:
//etc...
```

The return Statement

The return statement enables a function to return a value to its caller. It has the general form:

return *expression;*

- where *expression denotes the value returned by the function. The type of this value* should match the return type of the function.
- For a function whose return type isvoid, *expression should be empty:* return;

Overview for functions

A function is a set of statements designed to accomplish a particular task. The advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program. C++ has added many new features to functions to make them more reliable and flexible.

General Format of a Function Definition:

Functions can be define before the definition of the main() function, or they can be declared before it and define after it.

Declaring a function means listing its return type, name, and arguments. This line is called the function prototype. A function prototype tells the compiler the type of data returned by the function. It is usually defined after the preprocessing statements at the beginning of the program.

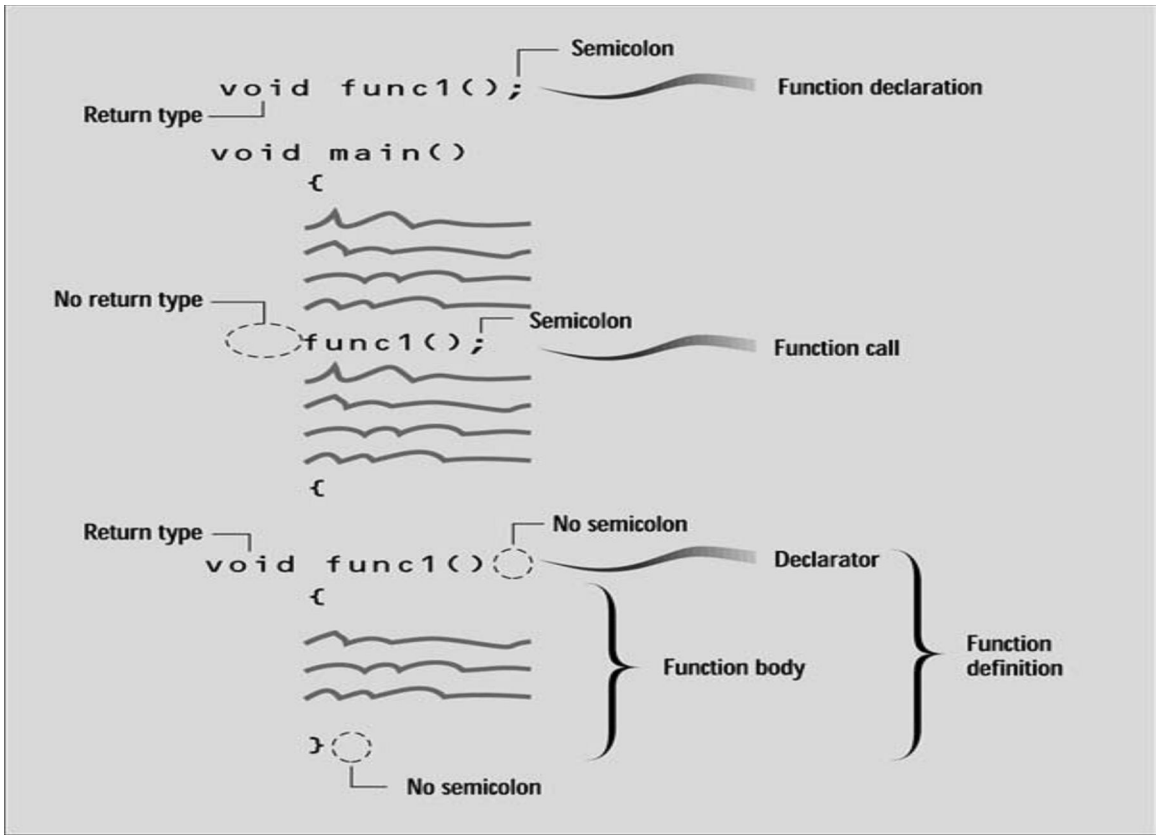


Figure 1: ***Function syntax.*****:**

Example 1:

Design a function to calculate the squared value of an integer passed from the main function. Use this function in a program to calculate the squares of integers from 1..10.

```
#include<iostream.h>

int square(int);

void main( )
{
        int x;
        for (x=1;x<=10;x++)
                cout << square(x)<<endl;
}

int square(int y)
{
        return (y*y);
}
```

Output

```
1
4
9
16
25
36
49
64
81
100
```

Local and Global Variables:

All variables define inside the block of a function are called local variables. These variables belong to the function exclusively. That is no other function has access to it.

Example 2:

Write a function to find the largest integer among three integers entered by the user in the main function.

```
#include <iostream.h>
int max(int,int,int);
void main( )
{
        int largest,x1,x2,x3;
        cout<<"Enter 3 integer numbers:";
        cin>>x1>>x2>>x3;
        largest=max(x1,x2,x3);
        cout<<largest;
}
```

```
int max(int y1, int y2, int y3)
{
        int big;
        big=y1;
        if (y2>big) big=y2;
        if (y3>big) big=y3;
        return (big);
}
```

When variables declared outside any function, it is called global variables. This type of variables can be accessed by all the functions in the program as well as the main function.

Inline Function:

C++ suppliers' programmers with the inline keyword, which can speed up programs by making very short functions execute more efficiently. Normally a function resides in a separate part of memory, and is referred to by a

running program in which it is called. Inline functions save the step of retrieving the function during execution time, at the cost of a larger compiled program.

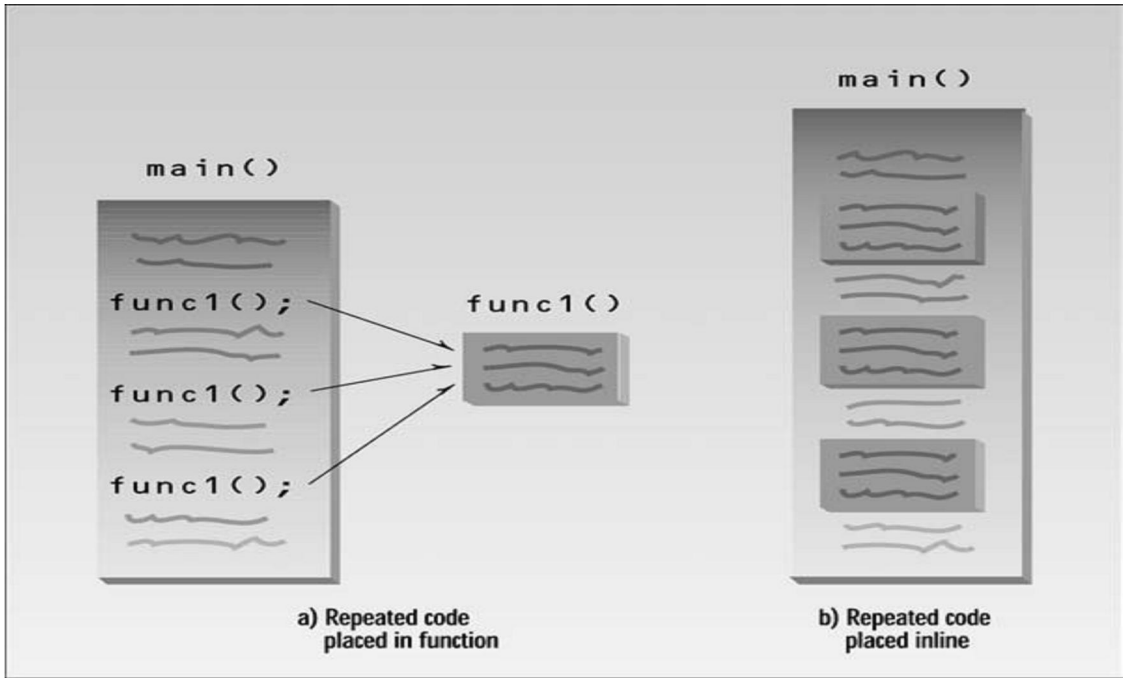


Figure: ***Functions versus inline code***

Example 3:

The following MAX is an inline function that returns the maximum of two values.

```
#include <iostream.h>

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

void main( )
{
        int x1,x2;
        cout<<"Enter 2 integer numbers:";
        cin>>x1>>x2;
        cout<<MAX(x1,x2);
}
```

Function Overloading:

Overloading refers to the use the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks.

We can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call.

Example 4:

The following program illustrates function overloading.

```
#include <iostream.h>

int       volume(int);
double volume(double,int);
long      volume(long,int,int);

void main( )
{
        cout<<volume(10)<<"\n";
        cout<<volume(2.5,8)<<"\n";
        cout<<volume(100L,75,15);
}

int volume(int s)
{
        return(s*s*s);                    // cube
}

double volume(double r,int h)       // cylinder
{
        return(3.14519*r*r*h);
}

long volume(long l,int b,int h)       // rectangular box
{
        return(l*b*h);
}
```

Passing Parameters:

There are two main methods for passing parameters to a program:

1- Passing by Value:

When parameters are passed by value, a copy of the parameters value is taken from the calling function and passed to the called function. The original variables inside the calling function, regardless of changes made by the function to it are parameters will not change. All the pervious examples used this method.

2- Passing by Reference:

When parameters are passed by reference their addresses are copied to the corresponding arguments in the called function, instead of copying their values. Thus pointers are usually used in function arguments list to receive passed references.

This method is more efficient and provides higher execution speed than the call by value method, but call by value is more direct and easy to use.

Example 5:

The following program illustrates passing parameter by reference.

```
#include <iostream.h>
void swap(int *a,int *b);
void main( )
{
        int x=10;
        int y=15;
        cout<<"x before swapping is:"<<x<<"\n";
        cout<<"y before swapping is:"<<y<<"\n";
```

```
        swap(&x,&y);
        cout<<"x after swapping is:"<<x<<"\n";
        cout<<"y after swapping is:"<<y<<"\n";
}

void swap(int *a,int *b)
{
        int c;
        c=*a;
        *a=*b;
        *b=c;
}
```

Example 6:

Write a C++ program using functions to print the contents of an integer array of 10 elements.

```
#include <iostream.h>

int const size=10;

void pary(int y[]);

void main( )
{
        int s[size]={2,4,6,8,10,12,14,16,18,20};
        pary(s);
}

void pary(int x[])
{
        int i;
        for (i=0;i<size;i++)
            cout<<x[i]<<endl;
}
```

Example 7:

Write a function that counts uppercase letter in a string entered by the user in the main program. Assume the maximum string length is 100.

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

int const size=100;

int upperCount(char[]);

void main( )
{
        char str[size];
        cout<<"Enter Your String: ";
        cin.getline(str,size,'\n');
        cout<<upperCount(str);
}

int upperCount(char x[])
{
        int count=0;
        for (int i=0;i<strlen(x);i++)
           if (x[i]>='A'&&x[i]<='Z')     count++;
        return (count);
}
```

Example 8:

Write a function that counts the number of words in a string entered by the user in the main program. Assume the maximum string length is 100.

Hint: *the number of words in a sentence can be found by counting the number of spaces.*

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

int const size=100;

int wordCount(char[]);

void main( )
{
        char str[size];
        cout<<"Enter a Sentenc: ";
        cin.getline(str,size,'\n');
        cout<<wordCount(str);
}

int wordCount(char x[])
{
        int count=1;
```

```
        for (int i=0;i<strlen(x);i++)
                if (x[i]==' ')    count++;
        return (count);
}
```

Example 9:

Design a function that takes the third element of an integer array defined in the main program. The function must multiply the element by 2 and return the result to the main program.

```
#include<iostream.h>
int elementedit(int);

void main( )
{
        int a[4]={2,5,3,7};
        int k;
        k=elementedit(a[2]);
        cout<<"k after change is: "<<k;
}

int elementedit(int m)
{
        m=m*2;
        return(m);
}
```

Arrays

An array consists of a set of objects (called its elements), all of which are of the same type and are arranged contiguously in memory. In general, only the array itself has a symbolic name, not its elements. Each element is identified by an index which denotes the position of the element in the array. The number of elements in an array is called its dimension. The dimension of an array is fixed and predetermined; it cannot be changed during program execution.

Arrays are suitable for representing composite data which consist of many similar, individual items. Examples include: a list of names, a table of world cities and their current temperatures, or the monthly transactions for a bank account.

An array variable is defined by specifying its dimension and the type of its elements. For example, an array representing 10 height measurements (each being an integer quantity) may be defined as:

```
int heights[10];
```

The individual elements of the array are accessed by indexing the array. The first array element always has the index 0. Therefore, heights[0] and heights[9] denote, respectively, the first and last element of heights. Each of heights elements can be treated as an integer variable. So, for example, to set the third element to 177, we may write:

```
heights[2] = 177;
```

Attempting to access a nonexistent array element (e.g., heights[-1] or heights[10]) leads to a serious runtime error (called ‘index out of bounds’ error).

Processing of an array usually involves a loop which goes through the array element by element. Listing 5.13 illustrates this using a function which takes an array of integers and returns the average of its elements.

The following example illustrates using a function which takes an array of integers and returns the average of its elements.

const int size = 3;
double Average (int nums[size])
{
double average = 0;
for (i = 0; i < size; ++i)
average += nums[i];
return average/size;
}

Array Initialization

An array may have an initializer. Braces are used to specify a list of comma-separated initial values for array elements. For example,

```
int nums[3] = {5, 10, 15};
```

initializes the three elements of nums to 5, 10, and 15, respectively. When the number of values in the initializer is less than the number of elements, the remaining elements are initialized to zero:

```
int nums[3] = {5, 10};
```

When a complete initializer is used, the array dimension becomes redundant, because the number of elements is implicit in the initializer. The first definition of nums can therefore be equivalently written as:

```
int nums[] = {5, 10, 15}; // no dimension needed
```

Strings

A C++ string is simply an array of characters. For example,

```
char str[ ] = "HELLO";
```

defines str to be an array of six characters: five letters and a null character. The terminating null character is inserted by the compiler. By contrast,

```
char str[ ] = {'H', 'E', 'L', 'L', 'O'};
```

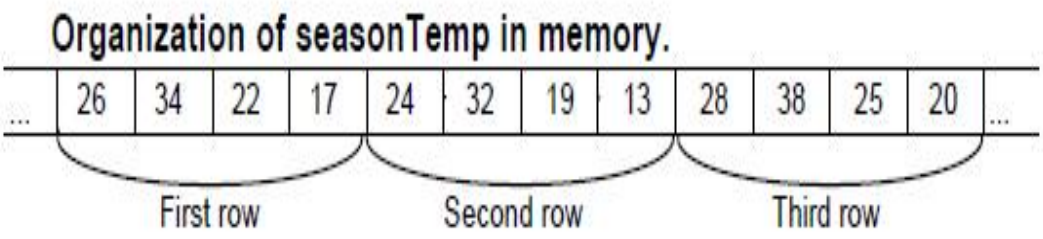
defines str to be an array of five characters.

Multidimensional Arrays

An array may have more than one dimension (i.e., two, three, or higher). The organization of the array in memory is still the same (a contiguous sequence of elements), but the programmer's perceived organization of the elements is different, for Example

```
int seasonTemp[3][4];
```

The organization of this array in memory is as 12 consecutive integer elements. The programmer, however, can imagine it as three rows of four integer entries each.



Multidimensional Arrays Initialization

The array may be initialized using a nested initializer:

```
int   seasonTemp[3][4] = {
{26, 34, 22, 17},
{24, 32, 19, 13},
{28, 38, 25, 10}
};
```

Because this is mapped to a one-dimensional array of 12 elements in memory, it is equivalent to:

```
int seasonTemp[3][4] = {26, 34, 22, 17, 24, 32, 19, 13, 28, 38, 25, 10};
```

Multidimensional Arrays Processing

Processing a multidimensional array is similar to a one-dimensional array, but uses nested loops instead of a single loop.:

Example 1

```
const int rows = 3;
const int columns = 4;
int seasonTemp[rows][columns] = {
{26, 34, 22, 17},
{24, 32, 19, 13},
{28, 38, 25, 10}
};
  int HighestTemp (int temp[rows][columns])
{  int highest = 0;
for (i = 0; i < rows; ++i)
for (j = 0; j < columns; ++j)
```

```
if (temp[i][j] > highest)
highest = temp[i][j];
return highest;
}
```

Pointers and References

- A pointer is simply the address of a variable in memory. Generally, variables can be accessed in two ways: directly by their symbolic name, or indirectly through a pointer. The act of getting to a variable via a pointer to it, is called dereferencing the pointer. Pointer variables are defined to point to variables of a specific type so that when the pointer is dereferencing, a typed variable is obtained.
- Pointers are useful for creating dynamic variables during program execution.
- Unlike normal (global and local) variables which are allocated storage on the runtime stack, a dynamic variable is allocated memory from a different storage area called the heap.
- Dynamic variables do not obey the normal scope rules. Their scope is explicitly controlled by the programmer.
- A pointer variable is defined to 'point to' data of a specific type. For example:

  int *ptr1; // pointer to an int

char *ptr2; // pointer to a char

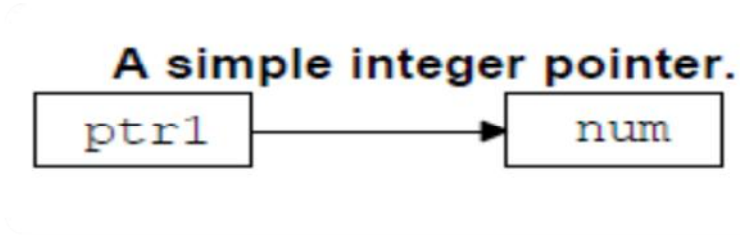
- The value of a pointer variable is the address to which it points. For example, given the definitions

int num;

- we can write:

ptr1 = #

- The symbol & is the address operator; it takes a variable as argument and returns the memory address of that variable. The effect of the above assignment is that the address of num is assigned to ptr1. Therefore, we say that ptr1 points to num.



Pointer Values

- Given that ptr1 points to num, the expression

***ptr1**

- Dereferences ptr1 to get to what it points to, and is therefore equivalent to num. The symbol * is the dereference operator; it takes a pointer as argument and returns the contents of the location to which it points.

- In general, the type of a pointer must match the type of the data it is set to point to.
- Regardless of its type, a pointer may be assigned the value 0 (called the null pointer). The null pointer is used for initializing pointers, and for marking the end of pointer-based data structures (e.g., linked lists).

Pointer Arithmetic

- In C++ one can add an integer quantity to or subtract an integer quantity from a pointer. This is frequently used by programmers and is called pointer arithmetic. Pointer arithmetic is not the same as integer arithmetic, because the outcome depends on the size of the object pointed to. For example, suppose that an int is represented by 4 bytes. Now, given

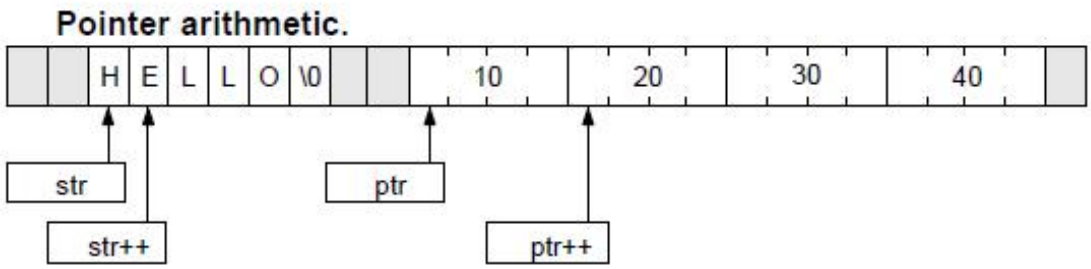
```
char   *str = "HELLO";

int   nums[] = {10, 20, 30, 40};

int   *ptr = &nums[0]; // pointer to first element
```

str++ advances str by one char (i.e., one byte) so that it points to the second character of "HELLO", whereas ptr++ advances ptr by one int (i.e., four bytes) of that it points to the second element of nums.

Pointer Arithmetic – cont.



It follows, therefore, that the elements of "HELLO" can be referred to as *str, *(str + 1), *(str + 2), etc. Similarly, the elements of nums can be referred to as *ptr, *(ptr + 1), *(ptr + 2), and *(ptr + 3).

- Pointer arithmetic is very handy when processing the elements of an array.

Example 2 The following example shows a string copying function similar to strcpy.

```
void CopyString (char *dest, char *src)

{

while (*dest++ = *src++);

}
```

Example 3:- **The following example shows how to pass a string argument and return a string to the main program:**

```
#include <string.h>
#include <iostream>
using namespace std;
char* stf(char *str3);
int main()
{
char *str1 = "ABC";
const int MAX = 10; //size of str2 buffer
char str2[MAX]; //empty string
char *st;
strcpy(str2, str1); //copy str1 to str2
cout << str2 << endl; //display str2
st=stf(str1);
cout << str1 << endl; //display str1
cout << st << endl; //display str1
return 0;
}
char* stf(char *str3)
{
    char* pointer;
    char *str1 ="XYZ";
    str3=str1;
    pointer=&str3[0];
    cout << str3 << endl;
    return pointer;
}
```

```
Output:

ABC
XYZ
ABC
XYZ
```

References

- A reference provides an alternative symbolic name (alias) for an object. Accessing an object through a reference is exactly the same as accessing it through its original name. References offer the power of pointers and the convenience of direct access to objects. They are used to support the call-by-reference style of function parameters, especially when large objects are being passed to functions.

Example 4:-

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


void main()
{
int z=1;
int &x = z;
++x;
int y = x + z;
cout <<y;
    return ;
}
```

Example 5:- The following example shows the difference of passing argument by value, pointer and reference

```
#include<iostream.h>
void Swap1 (int x, int y);
void Swap2 (int *x, int *y);
void Swap3 (int &x, int &y);
int main (void)
{ int i = 10, j = 20;
Swap1(i, j);
cout << "return from swap1=";
 cout << i << ", " << j << '\n';
Swap2(&i, &j);
cout<< "return from swap2=  ";
cout << i << ", " << j << '\n';
Swap3(i, j);
cout<<"return from swap3=  ";
cout << i << ", " << j << '\n';
     return 0; }
```

```
// pass-by-value (objects)
void Swap1 (int x, int y)
{ int temp = x;
x = y;
y = temp; }

// pass-by-value (pointers)
void Swap2 (int *x, int *y)
{ int temp = *x;
*x = *y;
*y = temp; }

// pass-by-reference
void Swap3 (int &x, int &y)
{ int temp = x;
x = y;
y = temp; }
```

Typedefs

Typedef is a syntactic facility for introducing symbolic names for data types. Just as a reference defines an alias for an object, a typedef defines an alias for a type. Its main use is to simplify otherwise complicated type declarations as an aid to improved readability. Here are a few examples:

```
typedef   char *String;

Typedef   char Name[12];

typedef   unsigned int uint;
```

The effect of these definitions is that String becomes an alias for char*, Name becomes an alias for an array of 12 chars, and uint becomes an alias for unsigned int. Therefore:

```
String   str; // is the same as: char *str;
```

```
Name   name; // is the same as: char name[12];
uint   n; // is the same as: unsigned int n;
```

Structures

A structure is a collection of simple variables. The variables in a structure can be of different types: Some can be int, some can be float, and so on. (This is unlike the array, which we'll meet later, in which all the variables must be the same type.) The data items in a structure are called the *members* of the structure.

However, for C++ programmers, structures are one of the two important building blocks in the understanding of objects and classes. In fact, the syntax of a structure is almost identical to that of a class. A structure (as typically used) is a collection of data, while a class is a collection of both data and functions. So by learning about structures we'll be paving the way for an understanding of classes and objects. Structures in C++ (and C) serve a similar purpose to *records* in some other languages such as Pascal.

A Simple Structure

Let's start off with a structure that contains three variables: two integers and a floating-point number. This structure represents an item in a widget company's parts inventory. The structure is a kind of blueprint specifying what information is necessary for a single part. The company makes several kinds of widgets, so the widget model number is the first member of the structure. The number of the part itself is the next member, and the final member is the part's cost.

The program PARTS defines the structure part, defines a structure variable of that type called part1, assigns values to its members, and then displays these values.

Example1 :

```
#include <iostream>

//////////////////////////////////////////////////////////////////
struct part                    //declare a structure
   {
   int modelnumber;            //ID number of widget
   int partnumber;             //ID number of widget part
   float cost;                 //cost of part
   };
//////////////////////////////////////////////////////////////////
int main()
   {
   part part1;                 //define a structure variable

   part1.modelnumber = 6244;   //give values to structure members
   part1.partnumber = 373;
   part1.cost = 217.55F;
                               //display structure members
   cout << "Model "     << part1.modelnumber;
   cout << ", part "    << part1.partnumber;
   cout << ", costs $" << part1.cost << endl;
   return 0;
   }
```

The program's output looks like this:

Model 6244, part 373, costs $217.55

The PARTS program has three main aspects: defining the structure, defining a structure variable, and accessing the members of the structure. Let's look at each of these.

Defining the Structure

The structure definition tells how the structure is organized: It specifies what members the structure will have.

Here it is:

```
struct part
{
int  modelnumber;
int  partnumber;
```

```
float  cost;
};
```

Syntax of the Structure Definition

The keyword struct introduces the structure definition. Next comes the *structure name* or *tag*, which is part. The declarations of the structure members—modelnumber, partnumber, and cost—are enclosed in braces. A semicolon follows the closing brace, terminating the entire structure. Note that this use of the semicolon for structures is unlike the usage for a block of code. As we've seen, blocks of code, which are used in loops, decisions, and functions, are also delimited by braces. However, they don't use a semicolon following the final brace. Figure1 shows the syntax of the structure declaration.

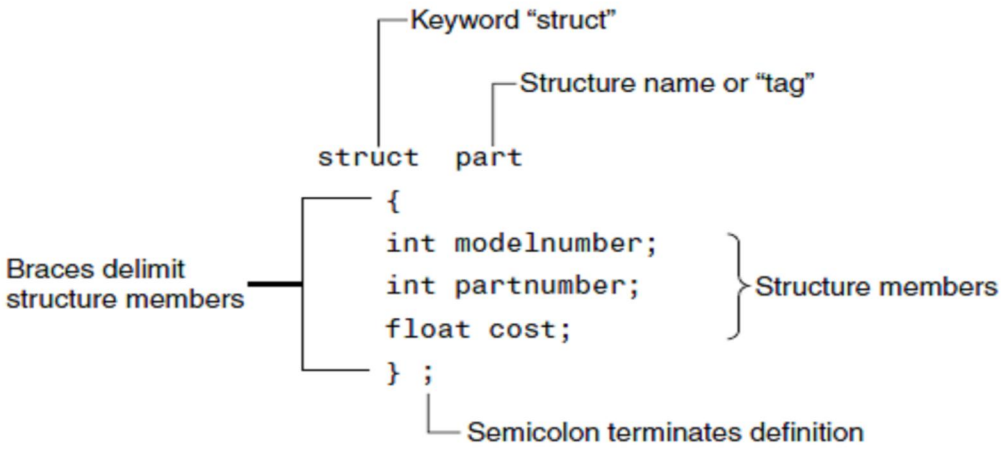


Figure 1: Syntax of the structure definition.

Defining a Structure Variable

The first statement in main() part part1; defines a variable, called part1, of type structure part. This definition reserves space in memory for part1. How much space? Enough to hold all the members of part1 namely modelnumber,

partnumber, and cost. In this case there will be 4 bytes for each of the two ints (assuming a 32-bit system), and 4 bytes for the float. Figure 2 shows how part1 looks in memory. (The figure shows 2-byte integers.) This will become more clear as we go along, but notice that the format for defining a structure variable is the same as that for defining a basic built-in data type such as int: part part1; int var1; This similarity is not accidental. One of the aims of C++ is to make the syntax and the operation of user-defined data types as similar as possible to that of built-in data types.

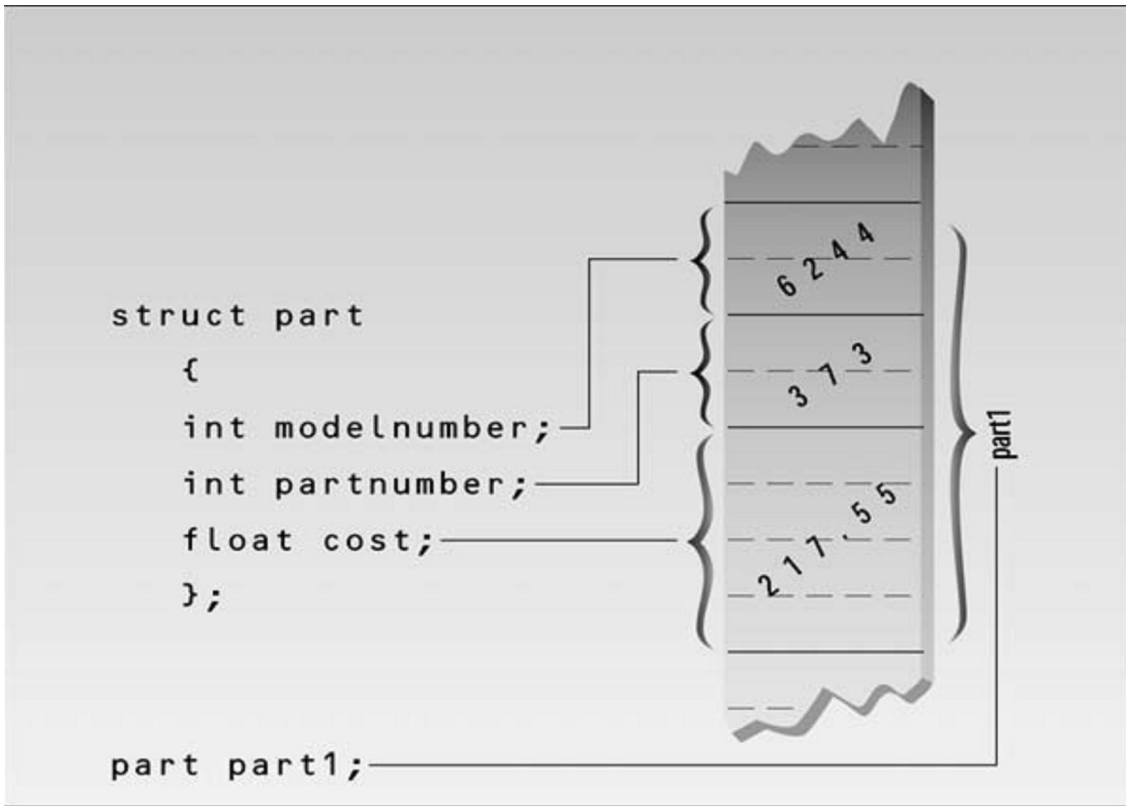


Figure 2: Structure members in memory.

Initializing Structure Members

The next example shows how structure members can be initialized when the structure variable is defined. It also demonstrates that you can have more than one variable of a given structure type.

Example 2 :

```
#include <iostream>
using namespace std;
//////////////////////////////////////////////////////////////
struct part                    //specify a structure
   {
   int modelnumber;            //ID number of widget
   int partnumber;             //ID number of widget part
   float cost;                 //cost of part
   };
//////////////////////////////////////////////////////////////
int main()
   {                           //initialize variable
   part part1 = { 6244, 373, 217.55F };
   part part2;                 //define variable
                               //display first variable
   cout << "Model "    << part1.modelnumber;
   cout << ", part "   << part1.partnumber;
   cout << ", costs $" << part1.cost << endl;

   part2 = part1;              //assign first variable to second
                               //display second variable
   cout << "Model "    << part2.modelnumber;
   cout << ", part "   << part2.partnumber;
   cout << ", costs $" << part2.cost << endl;
   return 0;
   }
```

This program defines two variables of type part: part1 and part2. It initializes part1, prints out the values of its members, assigns part1 to part2, and prints out its members.

Here's the output:

Model 6244, part 373, costs $217.55

Model 6244, part 373, costs $217.55

Structures as Arguments

The following example features a function that uses an argument of type structure named Distance:

Example 3

```
#include <iostream>
using namespace std;
struct Distance //English distance
{
int feet;
float inches;
};
Void disp( Distance ); //declaration
int main()
{
Distance d1, d2; //define two lengths
cout << "Enter feet: "; cin >> d1.feet;
cout << "Enter inches: "; cin >> d1.inches;
cout << "\nEnter feet: "; cin >> d2.feet;
cout << "Enter inches: "; cin >> d2.inches;
cout << "\nd1 = ";
disp(d1); //display length 1
cout << "\nd2 = ";
disp(d2); //display length 2
cout << endl;
return 0;
}
Void disp( Distance dd )      //parameter dd of type Distance
{
cout << dd.feet << "\'-" << dd.inches << "\"";
}
```

```
Enter feet: 6
Enter inches: 4
Enter feet: 5
Enter inches: 4.25
```

```
d1 = 6'-4"
d2 = 5'-4.25"
```

Returning Structure Variables

Example 4

```
#include <iostream>
using namespace std;
struct Distance //English distance
{
int feet;
float inches;
};
Distance add (Distance, Distance); //declarations
Void disp(Distance);
int main()
{
Distance d1, d2, d3; //define three lengths
cout << "\nEnter feet: "; cin >> d1.feet;
cout << "Enter inches: "; cin >> d1.inches;
cout << "\nEnter feet: "; cin >> d2.feet;
cout << "Enter inches: "; cin >> d2.inches;
d3 = add (d1, d2); //d3 is sum of d1 and d2
cout << endl;
disp(d1); cout << " + "; //display all lengths
disp(d2); cout << " = ";
disp(d3); cout << endl;
return 0;
}
Distance add ( Distance dd1, Distance dd2 )
{
Distance dd3; //define a new structure for sum
```

```
dd3.inches = dd1.inches + dd2.inches; //add the inches
dd3.feet = 0; //(for possible carry)
if(dd3.inches >= 12.0) //if inches >= 12.0,
{ //then decrease inches
dd3.inches -= 12.0; //by 12.0 and
dd3.feet++; //increase feet
} //by 1
dd3.feet += dd1.feet + dd2.feet; //add the feet
return dd3; //return structure
}
Void disp( Distance dd )
{
cout << dd.feet << "\'-" << dd.inches << "\"";
}
```

```
Enter feet: 4
Enter inches: 5.5
Enter feet: 5
Enter inches: 6.5
4’ 5.5” + 5’ 6.5”= 10’
```

Overview of OOP:

When working with computers, it is very important to be fashionable!

In the **1960s**, the new fashion was what was called *high-level languages* (H.L.L.) such as ***FORTRAN*** and ***COBOL***, in which the programmer did not have to understand the ***machine instructions***.

In the **1970s**, people realized that there were better ways to program than with a jumble of GOTO statements, and the *structured programming languages* such as ***PASCAL*** were invented.

In the **1980s**, much time was invested in trying to get good results out of *fourth-generation languages* (4GLs), in which complicated programming structures could be coded in a few words. There were also schemes such as Analyst Workbenches, which made systems analysts into highly paid and overqualified programmers.

Bjarne Stroustrup at Bell Labs developed C++ during 198-

1985. The term C++ was first used in 1983. Prior to 1983, Stroustrup added features to C programming language and formed what he called "C with Classes". In addition to the efficiency and portability of C, C++ provides number of new features. C++ programming language is basically an extension of C programming language.

The fashion of the **1990s** is most definitely *object-oriented programming*.

Read any book on object-oriented programming, and the first things you will read about are three importance OOP features:

- Encapsulation *and Data Hiding*.
- Inheritance *and Reuse*.
- Polymorphism.

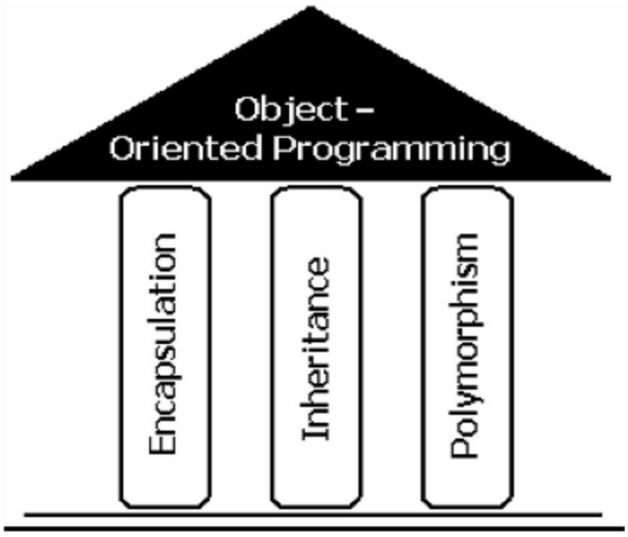


Figure 1: The three pillars of OOP.

Encapsulation and Data Hiding:

When an *engineer* needs to add a resistor to the device, he is create it, he doesn't build a new one from scratch. He walks over to a bin of resistors, examines the colored bands that indicate the properties, and picks the one he needs. The resistor is a "black box" as far as the engineer is concerned, he doesn't much care how its work as long as it conforms to his specifications, he doesn't need to look inside the box to use it in his design. When the engineer uses the resistor, he need not know anything about the internal state of the resistor. All the properties of the resistor are encapsulated in the resistor object; they are not spread out through the circuitry. It is not necessary to understand how the resistor works in order to use it effectively. Its data is hidden inside the resistor's casing.

Just as you can use a refrigerator without knowing how the compressor works, you can use a well-designed object without knowing about its internal data members.

The property of being a self-contained unit is called ***encapsulation***. With encapsulation, we can accomplish data hiding. ***Data hiding*** is the highly valued characteristic that an object can be used without the user knowing or caring how it works internally.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes.

Once created, class acts as a fully encapsulated entity, it is used as a whole unit. The actual inner workings of the class should be hidden. Users of a well-defined class do not need to know how the class works; they just need to know how to use it.

Inheritance and Reuse:

When the engineers at Acme Motors want to build a new car, they have two choices: They can start from scratch, or they can modify an existing model.

Perhaps their ***Star model*** is nearly perfect, but the engineers don't like to add a turbocharger and a six-speed transmission. The chief engineer would prefer not to start from the ground up, but rather to say, "Let's build another Star, but let's add these additional capabilities. We'll call the new model a ***Quasar model***". A Quasar model is a kind of Star model, but one with new features. C++ supports the idea of ***reuse*** through ***inheritance***. A new type, which is an extension of an existing type, can be declared. This new subclass is said to derive from the existing type and is sometimes called a derived type. The Quasar model is derived from the Star model and thus inherits all its qualities, but can add to them as needed.

Polymorphism:

The new Quasar model might respond differently than a Star model does when you press down on the accelerator. The Quasar model might engage fuel injection and a turbocharger, while the Star model would simply let gasoline into its carburetor. A user, however, does not have to know about these differences. He can just "floor it," and the right thing will happen, depending on which car he's driving. C++ supports the idea that different objects do "the right thing" through what is called function ***polymorphism*** and class polymorphism.

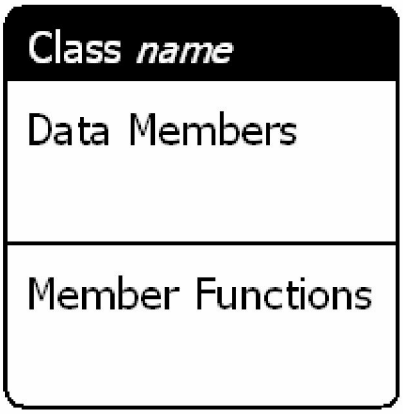
Poly means many, and morph means form. Polymorphism refers to the same name taking *many forms*.

Class Definition:

Class is a keyword, whose functionality is similar to that of the ***struct*** keyword, but with the possibility of including functions as members, instead of only data.

Classes are collections of variables and functions that operate on those variables. The variables in a class definition are called data members, and the functions are called member functions.

Note: Class is a specification for number of objects.



A **class definition** consists of two parts: header and body. The class **header** specifies the class **name** and its **base classes**. The class **body** defines the class **members**. Two types of members are supported:

- **Data members** have the syntax of variable definitions and specify the representation of class objects.
- **Member functions** have the syntax of function prototypes and specify the class operations, also called the class **interface**.

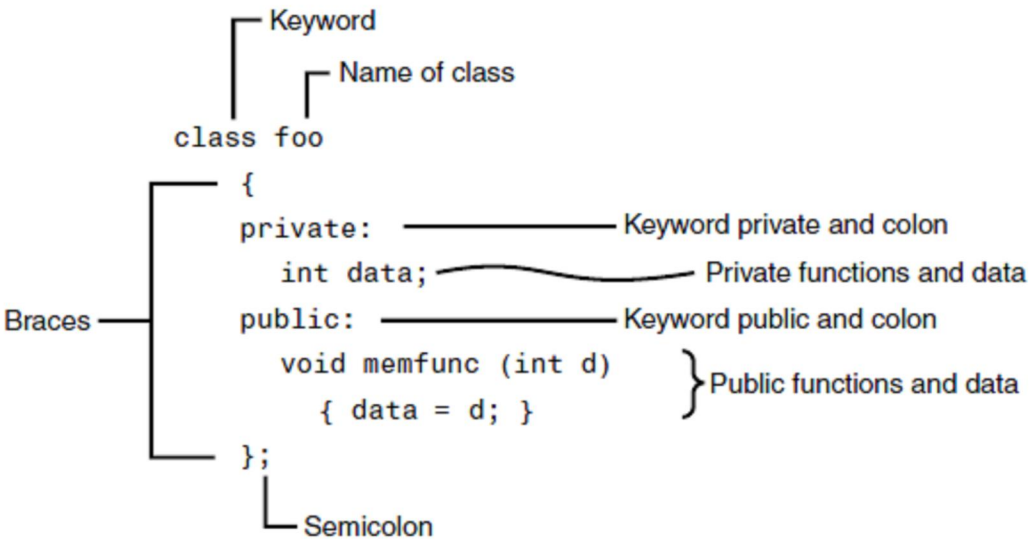


Figure2: Syntax of a class definition

We will **encapsulate** the data member which found in an object with member functions, and as you extend a class you will **"overload"** and **"override"** the functions of the *base class*.

When the permanently associate all related functions to the data of the class. This process is known as **encapsulation**.

Classes can be used to produce more specialized versions through a concept called **Inheritance**.

We can inherit features of the "**base class**", when we create a new "**derived class**". That is, we will create general object classes and then make more specific classes from them, deriving the particular from the general.

With such concept; new classes can be built on top of existing ones and extending their base classes; therefore allowing software designers to ***reuse code*** easily, which result in reducing the ***development time***.

For example, if we look at the class of a ***rectangle*** then we can think of data members to be the ***width*** and ***length***. Now if we are to define a new class of a ***square***, then we will have similar data members with the special case of having the ***width*** = ***length***, a member function to calculate the area of a square is also needed. Instead of defining the class of square from scratch, we can think of the square as the special case of a rectangle, hence, making use of the class of a rectangle by inheriting its behavior and redefining the area function to work for the class of squares.

New classes can ***modify the behavior of their base classes*** by redefining member functions to define new behaviors with different number and/or types of parameters, a capability known as **polymorphism**.

With such concept, the correct function is called at runtime based on the type and number of parameters passed.

For example, the operation "+" in the following expression:

A = B + C , thought of as an Integer addition if both B and C are integers. If both B and C are double however, the operation will be thought of as a Double addition. This is polymorphism in its simple way.

General form of class declaration:

```
class class-name
{
        public:
          public-data-members;
          public-functions;

        private:
          private-data-members;
          private -functions;

        protected:
          protected-data-members;
          protected -functions;
};
```

Class members fall under one of three different access permission categories:

- **Public** members are accessible by all class users.
- **Private** members are only accessible by the class members.
- ·**Protected** members are only accessible by the class members and the members of a derived class.

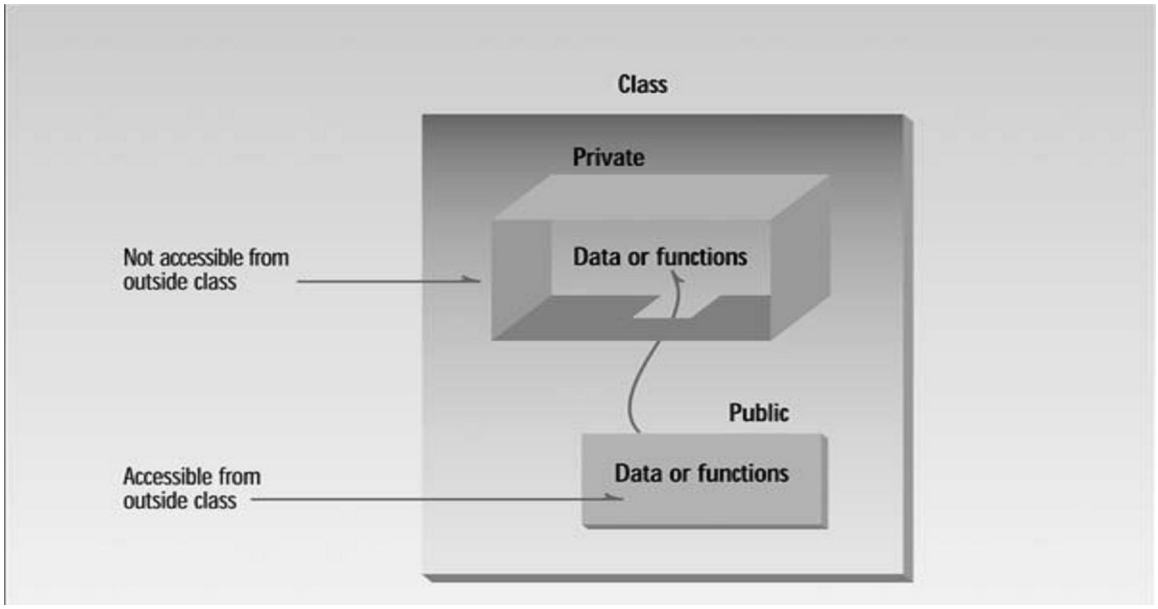


Figure 3: Public and Private Definition

Example 1:

Write a C++ program to modeling the Rectangle class.

```
# include <iostream.h>

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


int main( )
{
  Rectangle my_rectangle;

  my_rectangle.length = 6;
  my_rectangle.width = 7;
  cout<< my_rectangle.area( );

  return 0;
}
```

Example 2:A Simple Class

```
#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;
}
```

Example 3: write an oo program to define the coordinate of point and change the values of point.

```
#include<iostream.h>
class point {
          int xval , yval;
public:
   void setpt(int x , int y)
{
         xval=x;
         yval=y;
}
   void offsetpt(int x , int y)
{
         xval+=x;
         yval+=y;
                  cout<<xval<<yval;

}
};
void main()
{
 point pt;
pt.setpt(10,20);
pt.offsetpt(2,2);
}
```

Class Constructors and Destructors:

A **class constructor** is a function that is executed automatically whenever a new instance of a given class is declared.

The main purpose of a class constructor is to perform any initializations related to the class instances via passing of some parameter values as initial values and allocate proper memory locations for that object.

Note1: A class constructor must have the same name as that of the associated class.

Note2: A class constructor has not return type not even void.

Note3: A class constructor can be overloaded

Example 1: Write an oo program to represent simple constructor.

```
#include<iostream.h>
class point {
     int xval,yval;
public:
    point(int x,int y)    //constructor
{
    xval=x;
    yval=y;
}
    void offsetpt(int x,int y)
{
    xval+=x;
    yval+=y;
        cout<<xval<<yval;

}
};

void main()
{
 point pt(10,20);
pt.offsetpt(2,2);
}
```

Example 2: Write an oo program to represent a rectangle constructor.

```
# include <iostream.h>

class Rectangle
{
    int length , width;
    public:
     Rectangle(int x, int y)
{
    length = x;
    width = y;
}

int area( )
{
    return (length *width);
}

};


void main( )
{
    Rectangle rect1(6,7);
    cout<< rect1.area( ) << endl;
}
```

A class may have more than one constructor. To avoid ambiguity, however, each of these must have a unique signature.

Example 3: Write an oo program to represent multiple constructors.

```
# include <iostream.h>
class Rectangle
{
   int length , width;
   public:
    Rectangle( )                    //constructor1
    {
        length = 7;
        width = 9;
    }

    Rectangle(int x, int y)         //constructor2
{
   length = x;
   width = y;
}
    int area( )
    {   return (length *width);
    }

};

void main( )
{
   Rectangle rect1(6,7);
   Rectangle rect2;

   cout<< rect1.area( ) << endl;
   cout<< rect2.area( ) << endl;
}
```

Example 4

```
#include <iostream>
class Counter
{
private:
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;
};
```

Output:

C1=0

C2=0

C1=1

C2=2

Destructor

Just as a constructor is used to initialize an object when it is created, a destructor is used to clean up the object just before it is destroyed. A destructor always has the same name as the class itself, but is preceded with a ~ symbol. Unlike constructors, a class may have at most one destructor. A destructor never takes any arguments and has no explicit return type.

Destructors are generally useful for classes which have pointer data members which point to memory blocks allocated by the class itself.

In such cases it is important to release member-allocated memory before the object is destroyed. A destructor can do just that.

Example 1: Write an oo program to represent a simple destructor.

```
# include <iostream.h>

class Rectangle
{
   int length , width;
   public:
    Rectangle(int x, int y)            //constructor
{
   length = x;
   width = y;
}
    ~ Rectangle()                     //destructor
    {
        cout<<"destructor delete data"<<endl;
    }


int area( )
{
   return (length *width);
}

};


void main( )
{
   Rectangle rect1(6,7);
   cout<< rect1.area( ) << endl;
}
```

Example2: Write an oo program to represent destructor of pointer members.

```
# include <iostream.h>

class Rectangle
{
   int *length , *width;
   public:
    Rectangle(int x, int y)
{
   length= new int;
   width= new int;
   *length = x;
   *width = y;

}

    ~Rectangle()
{
   delete length;
   delete width;
}

int area( )
{
   return (*length **width);
}

};


void main( )
{
   Rectangle rect1(6,7);
   cout<< rect1.area( ) << endl;
}
```

Note: A class destructor proceeded with a tilde (~).

Friend function

Occasionally we may need to grant a function access to the nonpublic members of a class. Such an access is obtained by declaring the function a friend of the class.

There are two possible reasons for requiring this access:

• It may be the only correct way of defining the function.

• It may be necessary if the function is to be implemented efficiently.

Example1: Write an oo program to find the summation of point using friend function.

```
#include <iostream.h>

class point
{
private:
int xval,yval;
public:
    point()
    {
        xval=0;
        yval=0;
    cout<<"xval= "<<xval<<"yval= "<<yval<<endl;

    }
point(int x,int y)
    {
        xval=x+2;
        yval=y+3;
    cout<<"xval= "<<xval<<" "<<"yval= "<<yval<<endl;

    }

 friend int sum(point p);                    //friend function

};

int sum(point p)
{
int ss = p.xval + p.yval;
return (ss);
}

void main( )
{ point p2;
    point p(2,2);
    int dd;
        dd=sum(p);
    cout<<" the summation of coordinate x & y = "<<dd<<endl;
}
```

Example2: Write an oo program to represent a friend function .

```
#include <iostream.h>

class beta;                    //needed for frifunc declaration

class alpha
{
private:
int data;
public:
alpha() : data(3) { }
friend int frifunc(alpha, beta);     //friend function
};

class beta
{
private:
int data;
public:
beta() : data(7) { }
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;
}
```

Example3:Write an oo program to find the square of distance using friend function with overloading constructors

```
#include <iostream.h>
class Distance
{
private:
int feet;
float inches;
public:
Distance()                          //constructor (no arguments)
{
    feet=0;
 inches=0.0 ;
}

Distance(int ft, float in)          //constructor (two arguments)
{
    feet=ft;
 inches=in;
}
void showdist()                              //display distance
{ cout<<feet <<" "<<inches <<" "; }
friend float square(Distance);                 //friend function
};
//-------------------------------------------------------------
float square(Distance d)                     //return square of
{                                          //this Distance
float fltfeet = d.feet + d.inches/12;       //convert to float
float feetsqrd = fltfeet * fltfeet;          //find the square
return feetsqrd;                           //return square feet
}
//////////////////////////////////////////////////////////////////
int main()
{
Distance dist(3, 6.0);                        //two-arg constructor (3'-6")
float sqft;
sqft = square(dist);                           //return square of dist
                                               //display distance and square
cout << "\nDistance = "; dist.showdist();
cout << "\nSquare = " << sqft << " square feet\n";
return 0;
}
```

Example 4:

```
#include <iostream.h>
class D3;
class point
{
     int xVal;
     int yVal;
 public:
   point(int x, int y): xVal(x), yVal(y) {}
   friend void func1(point a, D3 b);
};
class D3
{
private:
int zVal;
public:
    D3(int zz): zVal(zz) {}
  friend void func1(point a, D3 b);
};
void func1(point a, D3 b) {
cout << "\nxVal=" << a.xVal; cout << "\nyVal=" << a.yVal; cout << "\nzVal=" << b.zVal; }
int main()
{
point d1(3,2);
D3 d2(6);
func1(d1,d2);
cout << endl;
return 0;
}
```

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 extreme case of having all member functions of a class A as friends of another class B can be expressed in an *abbreviated* form:

```
class A;
class B {
//...
friend class A; // abbreviated form
};
```

Example 1:Write an oo program to represent a friend class

```
#include <iostream.h>
//////////////////////////////////////////////////////////////
class alpha
{
private:
int data1;
public:
    alpha()
    {
        data1=99;
    }                          //constructor beta is a friend class
friend class beta;
};
//////////////////////////////////////////////////////////////
class beta
{                                       //all member functions can access private alpha data
public:
void func1(alpha a) { cout<<"\n data1="<< a.data1; }
void func2(alpha a) { cout<<"\n data1= "<<a.data1; }
};
//////////////////////////////////////////////////////////////
int main()
{
alpha a;
beta b;
b.func1(a);
b.func2(a);
cout<<endl;
return 0;
}
```

Example 2:Write an oo program to represent a friend class to class point

```
#include <iostream.h>

class point
{
private:
int xval,yval;
public:
    point(int x,int y)
    {
        xval=x+2;
        yval=y+3;

    }
    friend class display;
};

class display
{
public:
    void disdata(point p)
    {
        cout<<"xval= "<<p.xval<<" "<<"yval= "<<p.yval<<endl;
    }

};


void main( )
{   point p(2,2);
    display d;
        d.disdata(p);
    cout<<endl;
}
```

Example 3:

```
#include <iostream.h>
class point
{
int xVal;
int yVal;
public:
point(int x, int y): xVal(x), yVal(y)
{}
friend class D3;
};
class D3
{
private:

int  zVal;
public:
D3(int  zz): zVal(zz)
{}
void func1(point a)
{ cout << "\nxVal=" << a.xVal; cout << "\nyVal=" << a.yVal; cout <<
"\nzVal=" << zVal; }
};

int main()
{
D3 d1(6);
point d2(3,2);
d1.func1(d2);
cout << endl;
  return 0;
}
```

Output:
3
2
6

Default Arguments

As with global functions, a member function of a class may have default arguments. The same rules apply: all default arguments should be trailing arguments, and the argument should be an expression consisting of objects defined within the scope in which the class appears.

Surprisingly, a function can be called without specifying all its arguments. This won't work on just any function: The function declaration must provide default values for those arguments that are not specified.

Example 1:Write a simple program to represent a default argument

```
// missarg.cpp
// demonstrates missing and default arguments
#include <iostream>
using namespace std;

void repchar(char='*', int=45);   //declaration with
                                  //default arguments
int main()
   {
   repchar();                     //prints 45 asterisks
   repchar('=');                  //prints 45 equal signs
   repchar('+', 30);              //prints 30 plus signs
   return 0;
   }
//-------------------------------------------------------------
// repchar()
// displays line of characters
void repchar(char ch, int n)      //defaults supplied
   {                              //   if necessary
   for(int j=0; j<n; j++)         //loops n times
      cout << ch;                 //prints ch
   cout << endl;
   }
```

In this program the function repchar() takes two arguments. It's called three times from main(). The first time it's called with no arguments, the second time with one, and the third time with two. Why do the first two calls work? Because the called function provides default arguments, which will be used if the calling program doesn't supply them. The default arguments are specified in the declaration for repchar():

void repchar(char='*', int=45); //declaration

The default argument follows an equal sign, which is placed directly after the type name. You can also use variable names, as in

void repchar(char reptChar='*', int numberReps=45);

If one argument is missing when the function is called, it is assumed to be the last argument. The repchar() function assigns the value of the single argument to the ch parameter and uses the default value 45 for the n parameter. If both arguments are missing, the function assigns the default value '*' to ch and the default value 45 to n. Thus the three calls to the function all work, even though each has a different number of arguments.

Implicit Member Argument

When a class member function is called, it receives an implicit argument which denotes the particular object (of the class) for which the function is invoked. For example, in

Point pt(10,20);

pt.OffsetPt(2,2);

pt is an implicit argument to **OffsetPt**. Within the body of the member function, one can refer to this implicit argument explicitly as **this**, which denotes a pointer to the object for which the member is invoked. Using this, OffsetPt can be rewritten as:

Point::OffsetPt (int x, int y)

{

this->xVal += x; // equivalent to: xVal += x;

this->yVal += y; // equivalent to: yVal += y;}

The **this** pointer can be used for referring to member functions in exactly the same way as it is used for data members. It is important to bear in mind, however, that **this** is defined for use within member functions of a class only.

Example 1:Write an oo program to represent the *this* pointer

```
#include<iostream.h>
class point {
      int xval,yval;
public:

     void offsetpt(int x,int y)
{
     this->xval=x+2;
     cout<<this->xval;
        this->yval=y+5;
         cout<<this->yval;
         cout<<'\n';

}
};
void main()
{
 point pt;
pt.offsetpt(2,2);
}
```

Example 2:Write an oo program to represent the *this* pointer

```
#include <iostream.h>
//////////////////////////////////////////////////////////////////
class what
{
private:
int alpha;
public:
void tester()
{
this->alpha = 11; //same as alpha = 11;
cout<<this->alpha; //same as cout << alpha;
}
};
//////////////////////////////////////////////////////////////////
int main()
{
what w;
w.tester();
cout<<endl;
return 0;
}
```

Scope Operator

When calling a member function, we usually use an abbreviated syntax. For example:

pt.OffsetPt(2,2); // abbreviated form

This is equivalent to the full form:

pt.Point::OffsetPt(2,2); // full form

The full form uses the binary **scope operator ::** to indicate that OffsetPt is a member of Point.

In some situations, using the scope operator is essential. For example, the case where the name of a class member is hidden by a local variable (e.g., member function parameter) can be overcome using the scope operator:

```
class Point {
public:
Point (int x, int y) { Point::x = x; Point::y = y; }
//...
private:
int x, y;
}
```

Here **x** and **y** in the constructor (inner scope) hide **x** and **y** in the class (outer scope). The latter are referred to explicitly as **Point::x** and **Point::y.**

Example 1:

Write a C++ program under the concept of class constructor to modeling the Rectangle class, using scope resolution operator with the member function.

Note: all data member are private.

```
# include <iostream.h>

class Rectangle
{
   int length , width;
   public:
         Rectangle(int, int);
         int area( );
};

Rectangle :: Rectangle(int x, int y)
{
  length = x;

  width = y;
}

int Rectangle::area( )
{
   return length * width;
}

int main( )
{
   Rectangle my_rectangle(6,7);

   cout<< my_rectangle.area( );

   return 0;
}
```

<u>Member Initialization List</u>

There are two ways of initializing the data members of a class.

1) The first approach involves initializing the data members using assignments in the body of a constructor. For example:

```
class Image {
public:
Image (const int w, const int h);
private:
int width;
int height;
//...
};
Image::Image (const int w, const int h)
{
width = w;
height = h;
//...
}
```

Example 1: Write an oo program to initialize the data member of a class using assignments in the body of constructor.

```
# include <iostream.h>
class Rectangle
{
    int length , width;
public:
    Rectangle(const int l, const int w);
    int area( );
};

Rectangle :: Rectangle(const int l, const int w)
{
  length = l;
  width = w;
}

int Rectangle::area( )
{
   return length * width;
}

void main( )
{
    Rectangle my_rectangle(6,7);
    cout<<'\n';
    cout<< my_rectangle.area( );
    cout<<'\n';

}
```

2) The second approach uses a member initialization list in the definition of a constructor. For example:

class Image {

public:

Image (const int w, const int h);

private:

int width;

int height;

//...

};

Image::Image (const int w, const int h) : width(w), height(h)

{

//...

}

Example 2: Write an oo program to initialize the data member of a class in the definition of constructor.

```
# include <iostream.h>
class Rectangle
{
public:
    Rectangle(const int l, const int w);
    int area( );

private:
    int length , width;

};

Rectangle :: Rectangle(const int l, const int w):length(l),width(w)
{
  cout<<"i am in rectangle constructor";
}

int Rectangle::area( )
{
   return length * width;
}

void main( )
{
    Rectangle my_rectangle(6,7);
    cout<<'\n';
    cout<< my_rectangle.area( );
    cout<<'\n';

}
```

Constant member

A class data member may define as constant. For example:

```
class Image {
const int width;
const int height;
//...};
```

However, data member constants cannot be initialized using the same syntax as for other constants:

```
class Image {
const int width = 256; // illegal initializer!
const int height = 168; // illegal initializer!
//...
};
```

The correct way to initialize a data member constant is through a member initialization list:

```
class Image {
public:
Image (const int w, const int h);
private:
const int width;
const int height;
//...
};
Image::Image (const int w, const int h) : width(w), height(h)
```

{

//...

}

As one would expect, no member function is allowed to assign to a constant data member.

Constant Function Argument

If an argument is large, passing by reference is more efficient because, behind the scenes, only an address is really passed, not the entire variable.

Suppose you want to pass an argument by reference for efficiency, but not only do you want the function not to modify it, you want a guarantee that the function *cannot* modify it.

To obtain such a guarantee, you can apply the const modifier to the variable in the function declaration.

Example 1:Write a simple program to represent a constant member

```
//constarg.cpp
//demonstrates constant function arguments

void aFunc(int& a, const int& b);  //declaration

int main()
   {
   int alpha = 7;
   int beta = 11;
   aFunc(alpha, beta);
   return 0;
   }
//-------------------------------------------------------------
void aFunc(int& a, const int& b)   //definition
   {
   a = 107;   //OK
   b = 111;   //error: can't modify constant argument
   }
```

Here we want to be sure that aFunc() can't modify the variable beta. (We don't care if it modifies alpha.) So we use the const modifier with beta in the function declaration (and definition):

void aFunc(int& alpha, const int& beta);

Constant Member Functions

We can apply const to variables of basic types such as int to keep them from being modified. In a similar way, we can apply const to objects of classes. When an object is declared as const, you can't modify it.

Example 2:Write an oo program to represent a constant member function

```
// constObj.cpp
// constant Distance objects
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
class Distance                      //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                          //2-arg constructor
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()                //user input; non-const func
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const         //display distance; const func
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////
int main()
   {
   const Distance football(300, 0);

// football.getdist();              //ERROR: getdist() not const
   cout << "football = ";
   football.showdist();             //OK
   cout << endl;
   return 0;
   }
```

Static Members

A data member of a class can be defined to be static. This ensures that there will be exactly one copy of the member, shared by all objects of the class.

Example 1: Write an oo program to represent a static members.

```
// statfunc.cpp
// static functions and ID numbers for objects
#include <iostream>

 ////////////////////////////////////////////////////////////////
 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;
    }
```

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

Example 2: Write an oo program to represent a static member.

```
# include <iostream.h>
class test{

static int count;//count is static
int code;
public:
void setcode()
{
    cout<<"i am in set code"<<endl;
    code=++count;
}
void showcode()

{   cout<<"i am in show code"<<endl;

    cout<<"object number"<<code<<"\n";
}
static void showcount()
{   cout<<"i am in showcount"<<endl;

    cout<<"count:"<<count<<"\n";
}
};
int test::count;//count defined
void main ()
{ test t1,t2;
t1.setcode ();
t2.setcode ();
test::showcount ();
test t3;
t3.setcode ();
test::showcount ();
t1.showcode();
t2.showcode();
t3.showcode();
}
```

Member Pointers

It has already been stated that a pointer is a variable which holds the memory address of another variable of any basic data type such as *int, float* or sometimes an *array*. So far, it has been shown that how a pointer variable can be declared as a member of a class.

For example, the following declaration of creating an object

```
Class sample
{
Private :
  int   x;
  float   y;
  char  s;
public:
    void    getdata();
    void    display();
};
Sample     *ptr;
```

Which *ptr* is apointer variable that holds the address of the class object sample and consists of three data members such as *int x, float y, and char s* ,and also holds member functions such as getdata() and display().

The pointer to an object of class vaiable will be accessed and processed in one of the following ways .

First way :- ***(*object name).member name=variable;***

The parentheses are essential since the member of class period(.) has a higher precedence over the indirection operator (*).

Second way:- ***object name ->member name=variable;***

The pointer to the member of a class can be expressed using dash(-) followed by the greater than(>).

Example 1: Write a simple program using (*) to represent class pointer.

```
#include <iostream.h>
class student
{
private:
     int stageno;
     int age;
     char sex;
     float height;
     float weight;
public:
     void getinfo();
     void disinfo();
};     //end of class definition
void student::getinfo()
{
cout<<"stage number:";  cin>>stageno;
     cout<<"Age:";       cin>>age;
     cout<<"Sex :";      cin>>sex;
     cout<<"Height :";   cin>>height;
     cout<<"Weight :";   cin>>weight;
}


void student::disinfo()
{
     cout<<"Stage number = ";     cout<<stageno;  cout<<"\n";
     cout<<"Age= ";               cout<<age;      cout<<"\n";
     cout<<"Sex = ";              cout<<sex;      cout<<"\n";
     cout<<"Height =";            cout<<height;   cout<<"\n";
     cout<<"Weight =";            cout<<weight;
}

void main()
{
student *ptr;
ptr=new student;
cout<<"enter the following information"<<endl;
(*ptr).getinfo ();
cout<<endl;
cout<<"contents of class "<<endl;
(*ptr).disinfo();
}
```

Example 2: Write a simple program using (->) to represent class pointer.

```
#include <iostream.h>
class student
{
private:
    int stageno;
    int age;
    char sex;
    float height;
    float weight;
public:
    void getinfo();
    void disinfo();
};    //end of class definition
void student::getinfo()
{
    cout<<" Stage number :";     cin>>stageno;    cout<<endl;
    cout<<" Age:";               cin>>age;        cout<<endl;
    cout<< "Sex :";              cin>>sex;        cout<<endl;
    cout<<"Height :";            cin>>height;     cout<<endl;
    cout<<"Weight :";            cin>>weight;
}


void student::disinfo()
{
    cout<<"Stage number = ";     cout<<stageno;       cout<<"\n";
    cout<<" Age= ";              cout<<age;           cout<<"\n";
    cout<< "Sex = ";             cout<<sex;           cout<<"\n";
    cout<<"Height =";            cout<<height;        cout<<"\n";
    cout<<"Weight =";            cout<<weight;
}

void main()
{
student *ptr;
ptr=new student;
cout<<" enter the following information"<<endl;
ptr->getinfo();
cout<<" \n contents of class "<<endl;
ptr->disinfo();
}
```

This pointer

It is well known that a pointer is a variable which hold the memory address of another variable. Using the pointer technique, one can access the data of another variable indirectly. The *This* pointer is a variable which is used to access the address of the class itself.

Example 1:Write an oo program to display the address of class using *this* pointer

```
#include <iostream.h>
class sample
{
private :
    int x;
public:
    inline void display();
};
inline void sample::display()
{
    cout<<"object address = "<<this;
    cout <<endl;
}
void main()
{
    sample obj1,obj2,obj3;
    obj1.display();
    obj2.display();
    obj3.display();
}
```

The above program create three objects,**obj1,obj2,obj3** and displays each object's address using *this* pointer. The display() member function is used to give the address of the object

Example 2:Write an oo program to display the content of class member using *this* pointer

```
#include <iostream.h>
class sample
{
private :
    int x;
public:
    inline void display();
};
inline void sample::display()
{
    this->x=20;

    cout<<this->x;
    cout <<endl;
}
void main()
{
    sample obj1;
    obj1.display();
}
```

References Members

A class data member may define as reference. For example:

```
class  Image {
int   width;
int   height;
int   &widthRef;
//...
};
```

As with data member constants, a data member reference cannot be initialized using the same syntax as for other references:

```
class Image {
int  width;
int  height;
int  &widthRef = width; // illegal!
//...
};
```

The correct way to initialize a data member reference is through a member initialization list:

```
class Image {
public:
Image (const int w, const int h);
private:
int  width;
int  height;
int &widthRef;
```

```
//...
};
Image::Image (const int w, const int h) : widthRef(width)
{
//...
}
```

This causes widthRef to be a reference for width.

Example 1:Write an oo program to represent reference member of rectangle class

```
# include <iostream.h>
class Rectangle
{
public:
    Rectangle(const int l, const int w);
    int area(int l );

public:
    int length;
    int width;
    const int &height;

};

Rectangle :: Rectangle(const int l, const int w):length(l),width(w),height(l)
{

  cout<<" reference member height= "<<height;
}

int Rectangle::area(int l)
{

   return (l*l);
}

void main( )
{
    Rectangle my_rectangle(6,7);
    cout<<'\n';
    cout<<"area= ";
    cout<< my_rectangle.area( my_rectangle.width);
    cout<<'\n';

}
```

Example 2:Write an oo program to represent reference member of point class

```
# include <iostream.h>
class point
{
public:
    point(const int l, const int w);
    int sum(int l,int w );

public:
    int x;
    int y;
    const int z;

};

point :: point(const int l, const int w):x(l),y(w),z(l)
{

  cout<<" reference member height= "<<z;
}

int point::sum(int l,int w)
{

   return (l+w);
}

void main( )
{
    point pt(6,7);
    cout<<'\n';
    cout<<"summation=  ";
    cout<< pt.sum( pt.x,pt.y);
    cout<<'\n';

}
```

Class Object Member

A data member of a class may be of a user-defined type, that is, an object of another class. For example, a Rectangle class may be defined using two

Point data members which represent the top and bottom-right corners of the rectangle:

Example 1: Write a simple program to represent class object member.

```
# include <iostream.h>
class point
{
     int xval,yval;
public:
   point(int x,int y);
 };
point::point(int x,int y)
{
    xval=x;
    yval=y;
    cout<<"xval before change= "<<xval<<endl;
    cout<<"yval before change= "<<yval<<endl;
    xval=x+5;
    yval=y+5;
    cout<<"xval after change= "<<xval<<endl;
    cout<<"yval after change= "<<yval<<endl;
}
class Rectangle
{
public:
    Rectangle(int l,int r,int b,int t);
    int volume(int l,int r,int b,int t);
private:
    point length;
    point width;
};

Rectangle :: Rectangle(int l,int r,int b,int t):length(l,r),width(b,t)
{
  cout<<"the constructor is used to initiate the value of class point";
}

int Rectangle::volume(int l,int r,int b,int t)
{
    cout<<"volume= ";
return(l+r+b+t);
}

void main()
{   Rectangle my_rectangle(3,4,2,3);
    cout<<'\n';
    cout<< my_rectangle.volume(3,4,5,3);
    cout<<'\n';
}
```

Example 2: Write a simple program to represent class object member.

```
# include <iostream.h>
class square
{   public:
     int xval;
public:
   square(int x);
};
square::square(int x)
{
    xval=x+3;
    cout<<"x val after change= "<<xval<<endl;
}
class squarearea
{
public:
    squarearea(int l);
    int area(int l);

private:
    square length;
};

squarearea :: squarearea(int l):length(l)
{
    cout<<"length before change= "<<l<<endl;
}

int squarearea::area(int l)
{
    cout<<"area= ";
return(l*l);
}

void main()

{   squarearea s(3);
square ss(3);
    cout<<'\n';
    cout<< s.area(ss.xval);
    cout<<'\n';
}
```

Arrays as Class Member Data

Defining Arrays

Like other variables in C++, an array must be defined before it can be used to store information.

And, like other definitions, an array definition specifies a variable type and a name. But it includes another feature: a size. The size specifies how many data items the array will contain.

It immediately follows the name, and is surrounded by square brackets.

Figure 1 shows the syntax of an array definition.

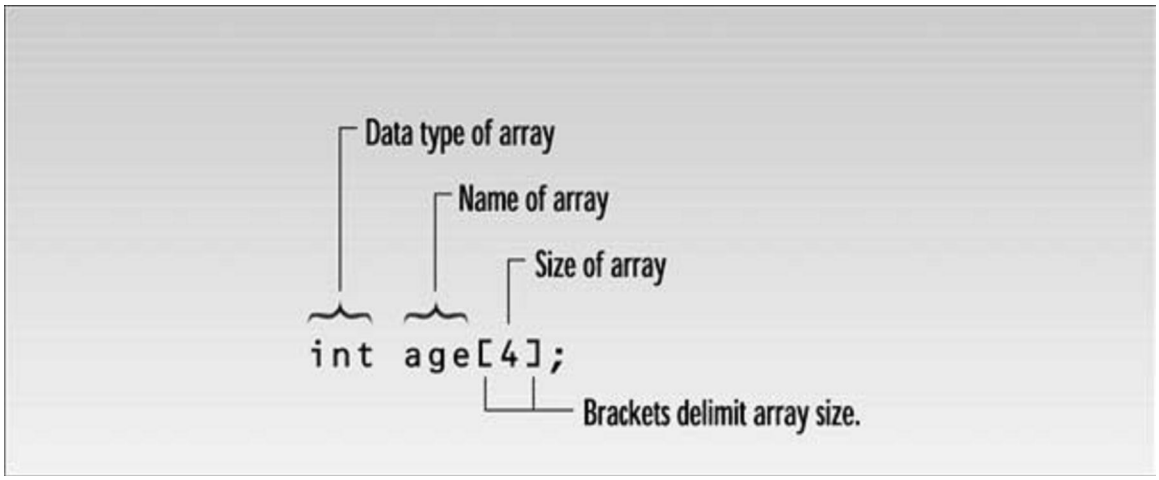


Figure1: syntax of array definition

The items in an array are called *elements* (in contrast to the items in a structure, which are called *members*). As we noted, all the elements in an array are of the same type; only the values vary. As specified in the definition, the array has exactly four elements.

Note:- the first array element is numbered 0. Thus, since there are four elements, the last one is number 3. This is a potentially confusing situation; you might think the last element in a four-element array would be number 4, but it's not.

```
#include <iostream.h>
class arr
{
private:
enum { MAX = 3 }; //constant definition
int ar[MAX]; // array of integers
public:
void get() //put number on stack
{
    int x, i;
    for (i=0;i<=MAX;i++)
    { cin >> x;
    ar[i] = x;  };
}
    void show()
    {
        int  i;
        for (i=0;i<=MAX; i++)
        cout << ar[i];
    }
    }; // end class
    int main()
    {
    arr a1;
    a1.get();
    a1.show();
    return 0;
    }
```

Defining Multidimensional Arrays

The array is defined with two size specifies, each enclosed in brackets:

```
int dimen2[5][3];
 elem = dimen2[x][y];
```

Of course there can be arrays of more than two dimensions. A three-dimensional array is an array of arrays of arrays. It is accessed with three indexes:

```
int dimen3[5][3][4];
elem = dimen3[x][y][z];
```

Strings as Class Members

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

class part
{
private:
char partname[30]; //name of widget part
int partnumber; //ID number of widget part
double cost; //cost of part
public:
void setpart(char pname[], int pn, double c)
{
strcpy(partname, pname);
partnumber = pn;
cost = c;
}
```

```
void showpart() //display data
{
cout << "\nName=" << partname;
cout << ", number=" << partnumber;
cout << ", cost=$" << cost;
}
};
int main()
{
part part1, part2;
part1.setpart("ABC", 4473, 217.55);
part2.setpart("XYZ", 9924, 419.25);
cout << "\nFirst part: "; part1.showpart();
cout << "\nSecond part: "; part2.showpart();
cout << endl;
      return 0;
}
```

Strings as Class Members – cont.

- This program defines two objects of class part and gives them values with the setpart() member function. Then it displays them with the showpart() member function. Here's the output:
- First part:
- Name= ABC, number=4473, cost=$217.55
- Second part:
- Name= XYZ, number=9924, cost=$419.25
- In the setpart() member function, we use the strcpy() string library function to copy the string from the argument pname to the class data member partname. Thus this function serves the same purpose with string variables that an assignment statement does with simple

variables. (A similar function, strncpy(), takes a third argument, which is the maximum number of characters it will copy. This can help prevent overrunning the array.)

- Besides those we've seen, there are library functions to add a string to another, compare strings, search for specific characters in strings, and perform many other actions.

Object Arrays

- An array of a user-defined type is defined and used much in the same way as an array of a built-in type. For example, a pentagon can be defined as an array of 5 points:

  Point pentagon[5];

- This definition assumes that Point has an 'argument-less' constructor (i.e., one which can be invoked without arguments). The constructor is applied to each element of the array.
- The array can also be initialized using a normal array initializer. Each entry in the initialization list would invoke the constructor with the desired arguments.

  When the initializer has less entries than the array dimension, the remaining elements are initialized by the argument-less constructor. For example,

  Point pentagon[5] = { Point(10,20), Point(10,30), Point(20,30), Point(30,20) };

- initializes the first four elements of pentagon to explicit points, and the last element is initialized to (0,0).

Arrays of Objects

- We can also create an array of objects. We'll look at a situations: an array of English distances.
- An array of a user-defined type is defined and used much in the same way as an array of a built-in type. For example, a pentagon can be defined as an array of 5 points:

  Point pentagon[5];

- This definition assumes that Point has an 'argument-less' constructor (i.e., one which can be invoked without arguments). The constructor is applied to each element of the array.
- The array can also be initialized using a normal array initializer. Each entry in the initialization list would invoke the constructor with the desired arguments. When the initializer has less entries than the array dimension, the remaining elements are initialized by the argument-less constructor. For example,

  Point pentagon[5] = { Point(10,20), Point(10,30), Point(20,30), Point(30,20)};

- initializes the first four elements of pentagon to explicit points, and the last element is initialized to (0,0).

Example 1: Write a simple program to represent array of class object point .

```
#include<iostream.h>
class point {
     int xval,yval;
public:
    void setpt(int x,int y)
{
    xval=x;
    yval=y;
}
    void offsetpt(int x,int y)
{
    xval+=x;
    yval+=y;
        cout<<xval<<yval;

}
};
void main()
{ int d,f;
 point pt[2];
 cout<<"enter the value of d & f";
 cin>>d>>f;
pt[0].setpt(d,f);
cout<<endl;
pt[1].setpt(30,40);

pt[0].offsetpt(2,2);
pt[1].offsetpt (4,6);

}
```

Example 2: Write a simple program to represent array of class object rectangle.

```
# include <iostream.h>
class Rectangle
{
public:
    int length , width;
    int area( )
    {
         return length * width;
    }
};
int main( )
{
    Rectangle my_rectangle[4];
int j=1;
    for (int i=0;i<4;i++)
    {
        cout<<"enter length ( " <<j << " ) "<<endl;
    cin>>my_rectangle[i].length ;
        cout<<"enter width ( " <<j << " ) "<<endl;

cin>>my_rectangle[i].width ;

    cout<< "area ( "<<  j <<" )" <<" ="<<my_rectangle[i].area( );
    cout<<endl;

    j=j+1;
    }    //end for loop
    return 0;
}
```

Example 3: Write a simple program to represent array of class object distance

```
#include <iostream.h>
//////////////////////////////////////////////////////////////////
class Distance                    //Distance class
{
private:
int feet;
float inches;
public:
void getdist()                     //get length from user
{
cout << "\n Enter feet: "; cin >> feet;
cout << " Enter inches: "; cin >> inches;
}
void showdist() const                 //display distance
{
    cout<<feet<<" "<<inches<<endl;
 }
};
//////////////////////////////////////////////////////////////////
int main()
{
Distance dist[100];               //array of objects distances
int n=0;                          //count the entries
char ans;                         //user response ('y' or 'n')

cout << endl;
do {                             //get distances from user
cout << "Enter distance number " << n+1;
dist[n++].getdist();              //store distance in array
cout << "Enter another (y/n)?: ";
cin >> ans;
} while( ans != 'n');           //quit if user types 'n'
for(int j=0; j<n; j++)          //display all distances
{
cout << "\nDistance number " << j+1 << " is ";
dist[j].showdist();
}
cout << endl;
return 0;
}
```

In this program the user types in as many distances as desired. After each distance is entered, the program asks if the user desires to enter another. If not, it terminates, and displays all the distances entered so far. Here's a sample interaction when the user enters three distances:

Output:-

Enter distance number 1

Enter feet: 5

Enter inches: 4

Enter another (y/n)? y

Enter distance number 2

Enter feet: 6

Enter inches: 2.5

Enter another (y/n)? y

Enter distance number 3

Enter feet: 5

Enter inches: 10.75

Enter another (y/n)? n

Distance number 1 is 5'-4"

Distance number 2 is 6'-2.5"

Distance number 3 is 5'-10.75"

Pointers to Objects

Pointers can point to objects as well as to simple data types and arrays. We've seen many examples of objects defined and given a name, in statements like

Distance dist;

where an object called dist is defined to be of the Distance class. Sometimes, however, we don't know, at the time that we write the program, how many objects we want to create. When this is the case we can use new to create objects while the program is running. As we've seen, new returns a pointer to an unnamed object.

Example 4

```
#include <iostream.h>
class Distance          //English Distance class
{
private:
int feet;
float inches;
public:
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 dist;          //define a named Distance object
```

```
dist.getdist();          //access object members
dist.showdist();         // with dot operator
Distance* distptr;       //pointer to Distance
distptr = new Distance;  //points to new Distance object
distptr->getdist();      //access object members
distptr->showdist();     // with -> operator
cout << endl;
return 0;
}
```

An Array of Pointers to Objects

A common programming construction is an array of pointers to objects. This arrangement allows easy access to a group of objects, and is more flexible than placing the objects themselves in an array.

Example 5

```
#include <iostream.h>
class person          //class of persons
{
protected:
char name[40];        //person's name
public:
void setName()        //set the name
{
cout << "Enter name: ";
cin >> name;
}
```

```
void printName() //get the name
{     cout << "\n Name is: " << name;   }
};
int main()
{
person* persPtr[100];    //array of pointers to persons
int n = 0;                        //number of persons in array
char choice;
do //put persons in array
{
persPtr[n] = new person;        //make new object
persPtr[n]->setName();          //set person's name
n++; //count new person
cout << "Enter another (y/n)? "; //enter another
cin >> choice; //person?
}
while( choice=='y' );            //quit on 'n'
for(int j=0; j<n; j++)          //print names of
{                //all persons
cout << "\nPerson number " << j+1;
persPtr[j]->printName();
}
cout << endl;
return 0;
  } //end main()
```

Operator overloading

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.

Overloading Unary Operators

Unary operators act on only one operand. Examples of unary operators are the increment and decrement operators ++ and --, and the unary minus, as in -33.

Example 1:-Write an oop program to increment the counter variable with ++ operator.

```
#include <iostream.h>
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;
}
};
////////////////////////////////////////////////////////////////////
void 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;
}
```

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.

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. 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 increment 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 1.

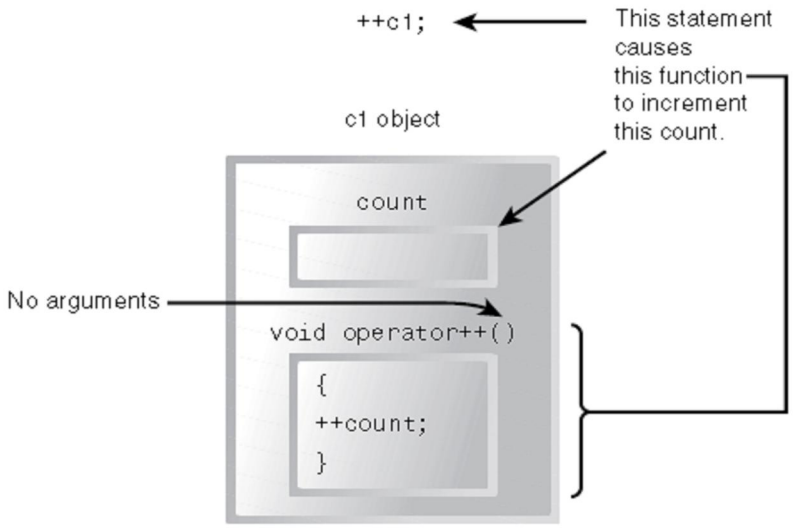


Figure 1: ***Overloaded unary operator: no arguments.***

Operator Return Values

The operator++ () function in the Example1 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.

Example 2:-Write an oop program increment the counter variable with ++ operator and return value.

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

Here the operator++() function creates a new object of type Counter, called temp, to use as a return value. It increments the count data in its own object as before, then creates the new temp object and assigns count in the new object the same value as in its own object. Finally, it returns the temp object. This has the desired effect. Expressions like ,**++c1** 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

Nameless Temporary Objects

In Example 2 we created a temporary object of type Counter, named temp, whose sole purpose was to provide a return value for the ++ operator. This required three statements.

Counter temp; // make a temporary Counter object

temp.count = count; // give it same value as this object

return temp; // return it

There are more convenient ways to return temporary objects from functions and overloaded operators. Let's examine another approach, as shown in the example3

Example 3:-Write an oop program increment the counter variable with ++ operator and unnamed temporary object.

```
#include <iostream.h>
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()   //return count
{ return count; }
Counter operator ++ ()    //increment count
{
++count;                  // increment count, then return
return Counter(count);    // an unnamed temporary object initialized to this 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
cout << "\nc1=" << c1.get_count();    //display again
cout << "\nc2=" << c2.get_count() << endl;
return 0;
}
```

In this program a single statement

return Counter(count);

This statement creates an object of type Counter.

Counter(int c) : count(c) //constructor, one arg

{ }

Once the unnamed object is initialized to the value of count, it can then be returned. The output of this program is the same as that of Example 2.

Postfix Notation

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 Example4:

Example 4:-Write an oop program increment the counter variable with ++ operator using both prefix and postfix.

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

Here's the output from the program:

c1=0

c2=0

c1=2

c2=2

c1=3

c2=2

Overloading Binary Operators

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

Example 5 shows how add two distances **dist3.add_dist(dist1, dist2);** By overloading the + operator we can reduce this dense-looking expression to **dist3 = dist1 + dist2;**

Example 5:-Write an oop program to add two distances objects.

```
#include <iostream.h>
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
};
Distance Distance::operator + (Distance d2) const //add this distance to d2 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 inches10
{ //
i -= 12.0;      //by 12.0 and
f++;            //increase feet by 1
}               //return a temporary Distance
return Distance(f,i);   //initialized to sum
}
//////////////////////////////////////////////////////////////////
void 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
cout << "dist1 = "; dist1.showdist(); cout << endl;  //display all lengths
cout << "dist2 = "; dist2.showdist(); cout << endl;
cout << "dist3 = "; dist3.showdist(); cout << endl;
cout << "dist4 = "; dist4.showdist(); cout << endl;
}
```

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
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+(). 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.

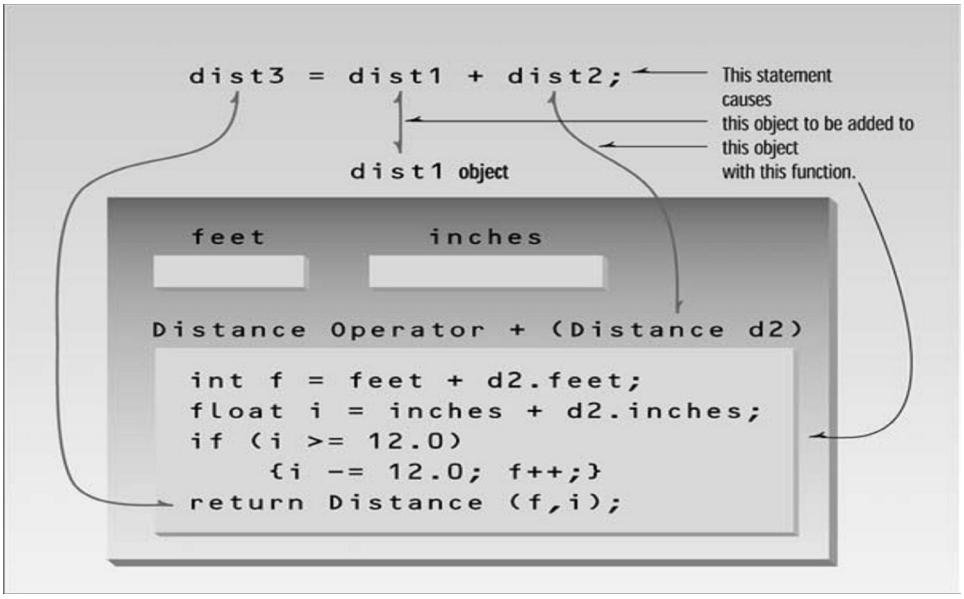


Figure 2: ***Overloaded binary operator: one argument.***

Concatenating Strings

We can overload the + operator to perform such concatenation. Here's the listing for Example 6:

Example 6:-Write an oop program to concatenate two strings.

```
#include <iostream.h>
#include <string.h> //for strcpy(), strcat() ,strlen()
class String    //user-defined string type
{
private:
enum { SZ=80 };     //size of String objects
char str[SZ];        //holds a string
public:
String()            //constructor, no args
{ strcpy(str, ""); }
String( char s[] )  //constructor, one arg
{ strcpy(str, s); }
void display() const  //display the String
{ cout << str; }
String operator + (String ss) const //add Strings
{
String temp; //make a temporary String
if( strlen(str) + strlen(ss.str) < SZ )
{
strcpy(temp.str, str); //copy this string to temp
strcat(temp.str, ss.str); //add the argument string
}
else
{ cout << "\nString overflow";}
return temp;    //return temp String
}
};
////////////////////////////////////////////////////////////////////
int main()
{
String s1 = "\nMerry Christmas! "; //uses constructor 2
String s2 = "Happy new year!"; //uses constructor 2
String s3;         //uses constructor 1
s1.display();      //display strings
s2.display();
s3.display();
s3 = s1 + s2;      //add s2 to s1,assign to s3
s3.display();      //display s3
cout << endl;
return 0;
}
```

The program first displays three strings separately. (The third is empty at this point, so nothing is printed when it displays itself.) Then the first two strings are concatenated and placed in the third, and the third string is displayed again.

Here's the output:

Merry Christmas! Happy new year! ←——— s1, s2, and s3 (empty)

Merry Christmas! Happy new year! ←——— s3 after concatenation

Multiple Overloading

We've seen different uses of the + operator: to add English distances and to concatenate strings.

You could put both these classes together in the same program, and C++ would still know how to interpret the + operator: It selects the correct function to carry out the "addition" based on the type of operand.

Comparison Operators

Let's see how to overload a different kind of C++ operator: comparison operators.

Comparing Distances

In our first example we'll overload the *less than* operator (<) in the Distance class so that we can compare two distances.

Example 7:-Write an oop program to compare two distances.

```
#include <iostream.h>
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 << '\"'; }
bool operator < (Distance) const; //compare distances
};
bool Distance::operator < (Distance d2) const //return the sum
{
float bf1 = feet + inches/12;
float bf2 = d2.feet + d2.inches/12;
return (bf1 < bf2) ? true : false;
}
////////////////////////////////////////////////////////////////////
void main()
{
Distance dist1; //define Distance dist1
dist1.getdist(); //get dist1 from user
Distance dist2(6, 2.5); //define and initialize dist2
//display distances
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
if( dist1 < dist2 ) //overloaded '<' operator
cout << "\ndist1 is less than dist2";
else
cout <<" \ndist1 is greater than (or equal to) dist2";
cout << endl;
}
```

This program compares a distance entered by the user with a distance, 6'–2.5", initialized by the program. Depending on the result, it then prints one of two possible sentences.

Here's some typical output:

Enter feet: 5

Enter inches: 11.5

dist1 = 5'-11.5"

dist2 = 6'-2.5"

dist1 is less than dist2

The approach used in the operator<() function is similar to overloading the + operator in the Example 7 except that here the operator<() function has a return type of Boolean. The return value is false or true, depending on the comparison of the two distances.

The comparison is made by converting both distances to floating-point feet, and comparing them using the normal < operator. Remember that the use of the conditional operator

return (bf1 < bf2) ? true : false;

is the same as

```
if (bf1 < bf2)
return true;
else
return false;
```

Comparing Strings

Here's another example of overloading an operator, this time the *equal to* (==) operator. We'll use it to compare two of our homemade String objects, returning true if they're the same and false if they're different.

Example 8:-Write an oop program to compare two strings.

```
#include <iostream.h>
#include <string.h>    //for strcmp()
////////////////////////////////////////////////////////////////
class String       //user-defined string type
{
private:
enum { SZ = 80 };   //size of String objects
char str[SZ];      //holds a string
public:
String()           //constructor, no args
{ strcpy(str, ""); }
String( char s[] )   //constructor, one arg
{ strcpy(str, s); }
void display() const  //display a String
{ cout << str; }
void getstr()     //read a string
{ cin.get(str, SZ); }
bool operator == (String ss) const   //check for equality
{
return ( strcmp(str, ss.str)==0 ) ? true : false;
}
};
////////////////////////////////////////////////////////////////
int main()
{
String s1 = "yes";
String s2 = "no";
String s3;
cout << "\nEnter 'yes' or 'no': ";
s3.getstr();        //get String from user
if(s3==s1)          //compare with "yes"
cout << "You typed yes\n";
else if(s3==s2)     //compare with "no"
cout << "You typed no\n";
else
cout << "You didn't follow instructions\n";
return 0;
}
```

The main() part of this program uses the == operator twice, once to see if a string input by the user is “yes” and once to see if it’s “no.”

Here's the output when the user types "yes":

Enter 'yes' or 'no': yes

You typed yes

The operator==() function uses the library function strcmp() to compare the two C-strings.

This functions return 0 if the strings are equal, a negative number if the first is less than the second, and a positive number if the first is greater than the second. Here *less than* and *greater than* are used in their lexicographical sense to indicate whether the first string appears before or after the second in an alphabetized listing.

Other comparison operators, such as < and >, could also be used to compare the lexicographical value of strings. Or, alternatively, these comparison operators could be redefined to compare string lengths. Since you're the one defining how the operators are used, you can use any definition that seems appropriate to your situation.

Arithmetic Assignment Operators

The += operator combines assignment and addition into one step. We'll use this operator to add one distance to a second, leaving the result in the first.

In this Example 9 we obtain a distance from the user and add to it a second distance, initialized to 11'–6.25" by the program.

In this program the addition is carried out in main() with the statement

dist1 += dist2; This causes the sum of dist1 and dist2 to be placed in dist1.

Example 9:-Write an oop program to add two distances using += operator.

```
#include <iostream.h>
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; }
void operator += ( Distance );
};
void Distance::operator += (Distance d2)
{
feet += d2.feet; //add the feet
inches += d2.inches; //add the inches
if(inches >= 12.0) //if total exceeds 12.0,then decrease inches
{
inches -= 12.0; //by 12.0 and
feet++;       //increase feet by 1
}
}
////////////////////////////////////////////////////////////////////
void main()
{
Distance dist1; //define dist1
dist1.getdist(); //get dist1 from user
cout << "\ndist1 = "; dist1.showdist();
Distance dist2(11, 6.25); //define, initialize dist2
cout << "\ndist2 = "; dist2.showdist();
dist1 += dist2; //dist1 = dist1 + dist2
cout << "\nAfter addition,";
cout << "\ndist1 = "; dist1.showdist();
cout << endl;
}
```

Here's a sample of interaction with the program:

Enter feet: 3

Enter inches: 5.75

dist1 = 3'-5.75"

dist2 = 11'-6.25"

After addition,

dist1 = 15'-0"

<u>The Subscript Operator ([])</u>

The subscript operator, [], which is normally used to access array elements, can be overloaded.

Example 10:-Write an oop program to create an array.

```
#include <iostream.h>
const int LIMIT = 10;
class safearay
{
private:
int arr[LIMIT];
public:
void putel(int n, int elvalue) //set value of element
{
if( n< 0 || n>=LIMIT )
{ cout << "\nIndex out of bounds";}
arr[n] = elvalue;
}
int getel(int n) const //get value of element
{
if( n< 0 || n>=LIMIT )
{ cout << "\nIndex out of bounds"; }
return arr[n];
}
};
////////////////////////////////////////////////////////////////
int main()
{
safearay sa1;
for(int j=0; j<LIMIT; j++) // insert elements
sa1.putel(j, j*10);
for(j=0; j<LIMIT; j++) // display elements
{
int temp = sa1.getel(j);
cout << "Element " << j << " is " << temp << endl;
}
return 0;
}
```

Single access() Function Returning by Reference

As it turns out, we can use the same member function both to insert data into the safe array and to read it out. The secret is to return the value from the function by reference. This means we can place the function on the left side of the equal sign, and the value on the right side will be assigned to the variable returned by the function.

Example 11:-Write an oop program to create an array and return by reference.

```
#include <iostream.h>
const int LIMIT = 100; //array size
class safearay
{
private:
int arr[LIMIT];
public:
int& access(int n) //note: return by reference
{
if( n< 0 || n>=LIMIT )
{ cout << "\nIndex out of bounds"; }
return arr[n];
}
};
//////////////////////////////////////////////////////////////////
int main()
{
safearay sa1;
for(int j=0; j<LIMIT; j++) //insert elements
sa1.access(j) = j*10; //*left* side of equal sign
for(j=0; j<LIMIT; j++) //display elements
{
int temp = sa1.access(j); //*right* side of equal sign
cout<< "Element " << j << " is " << temp << endl;
}
return 0;
}
```

The statement

sa1.access(j) = j*10; // *left* side of equal sign causes the value j*10 to be placed in arr[j], the return value of the function. It's perhaps slightly more convenient to use the same function for input and output of the safe array

than it is to use separate functions; there's one less name to remember. But there's an even better way, with no names to remember at all.

Overloaded [] Operator Returning by Reference

To access the safe array using the same subscript ([]) operator that's used for normal C++ arrays, we overload the subscript operator in the safearay class. However, since this operator is commonly used on the left side of the equal sign, this overloaded function must return by reference.

Example 12:-Write an oop program to create an array using operator [] with overload by reference.

```
#include <iostream.h>
const int LIMIT = 100; //array size
class safearay
{
private:
int arr[LIMIT];
public:
int& operator [](int n) //note: return by reference
{
if( n< 0 || n>=LIMIT )
{ cout << "\nIndex out of bounds"; }
return arr[n];
}
};
////////////////////////////////////////////////////////////
int main()
{
safearay sa1;
for(int j=0; j<LIMIT; j++) //insert elements
sa1[j] = j*10; //*left* side of equal sign
for(j=0; j<LIMIT; j++) //display elements
{
int temp = sa1[j]; //*right* side of equal sign
cout << "Element " << j << " is " << temp << endl;
}
return 0;
}
```

In this program we can use the natural subscript expressions

sa1[j] = j*10; and **temp = sa1[j];** for input and output to the safe array.

Overloadable operators.

Unary:	+	-	*	!	~	&	++	--	()	->	->*
	new	delete									
Binary:	+	-	*	/	%	&	|	^	<<	>>	
	=	+=	-=	/=	%=	&=	|=	^=	<<=	>>=	
	==	!=	<	>	<=	>=	&&	||	[]	()	,

Figure 3: Overloadable Operators

Operators that can't be overloaded are listed bellow:

Operator	Description
.	Dot operator.
.*(or ->)	Access member operator.
::	Scope resolution.
?:	Conditional operator.
sizeof.	Size of file

Figure 4: Operators can't be Overloaded

Inheritance

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 1.

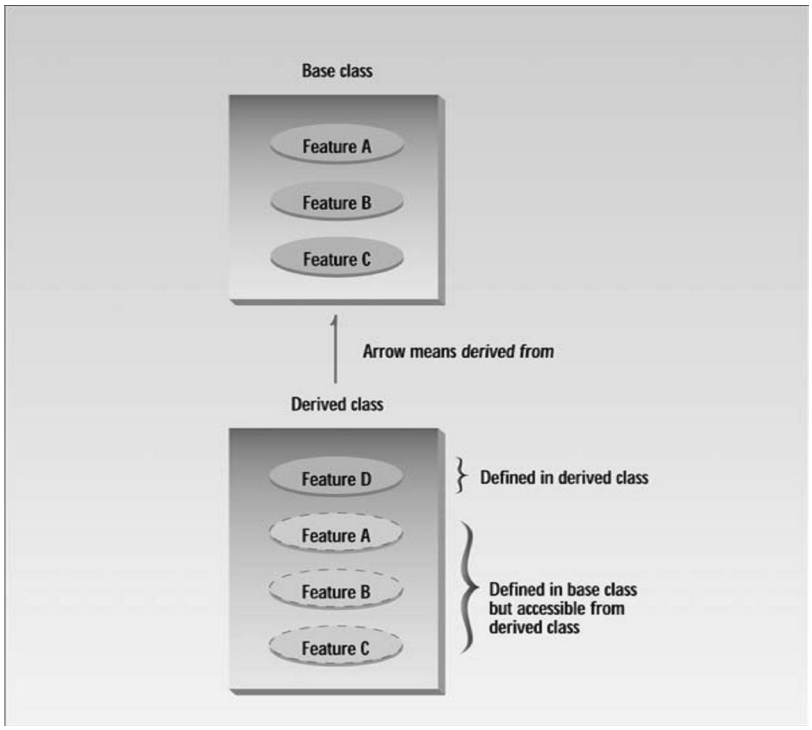


Figure 1: ***Inheritance.***

The arrow in Figure 1 goes in the opposite direction 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.

Derived Class and Base Class

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.

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. 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.

- Second reason for not modifying the Counter class: We might not have access to its source code, especially if it was distributed as part of a class library.

To avoid these problems we can use inheritance to create a new class based on Counter, without modifying counter itself. A new class, CountDn, that adds a decrement operator to the Counter class:

Example 1:-Write a program to decrement the counter variable using inheritance.

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

Output of Example1

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.

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.

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 example 1.

Substituting Base Class Constructors

In the main () part of Example1 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 example1 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.

<u>The protected Access Specifier</u>

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 class. In the Counter class in example1, count is given a new specifier: protected. What does this do?

Let's first review what we know about the access specifies 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 statement

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 2.

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(). The situation is shown in Figure 2.

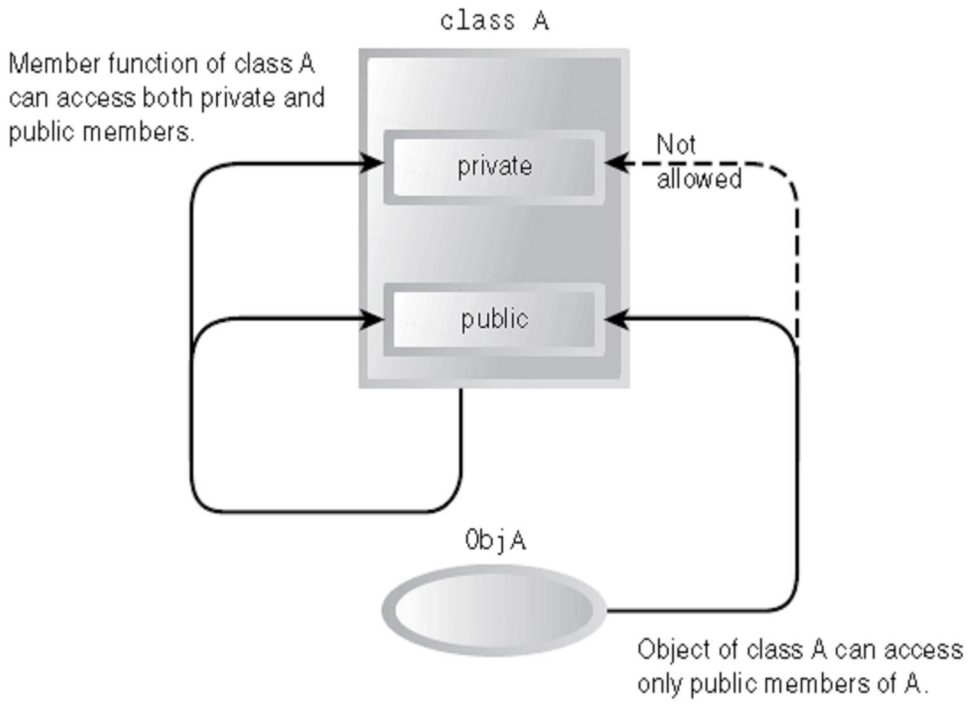


Figure 2: Access *specifiers without inheritance*

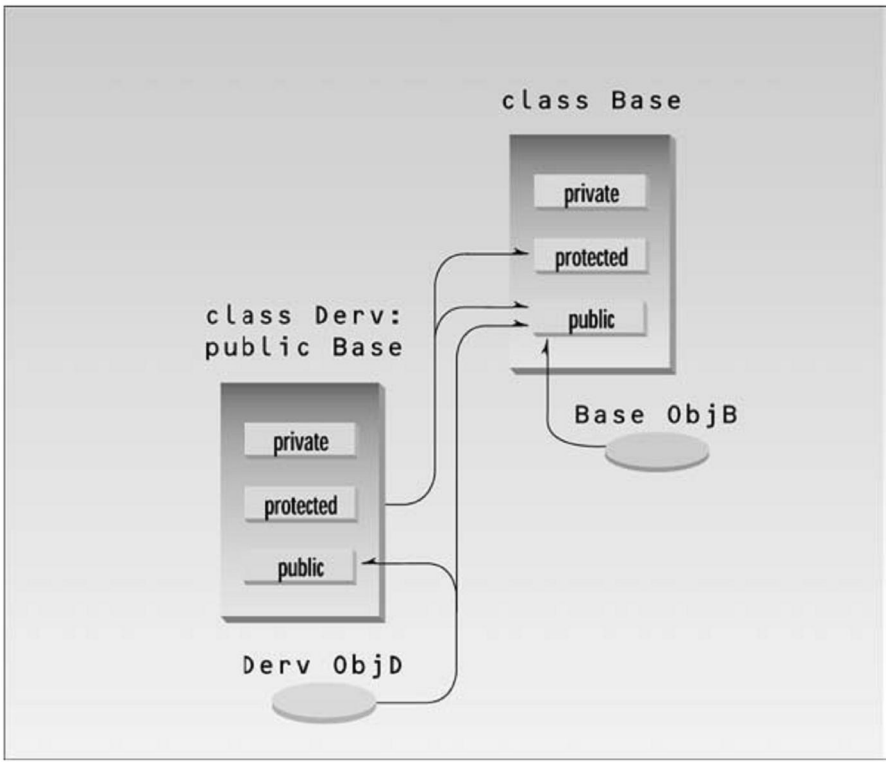


Figure 3: ***Access specifiers with inheritance.***

Table 1: Inheritance and Accessibility

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

The moral is that if you are writing a class that you suspect might be used, at any point in the future, as a base class for other classes, then any member data that the derived classes might need to access should be made protected rather than private. This ensures that the class is “inheritance ready.”

Dangers of protected

You should know that there's a disadvantage to making class members protected. Say you've written a class library, which you're distributing to the public. Any programmer who buys this library can access protected members of your classes simply by deriving other classes from them. This makes protected members considerably less secure than private members. To avoid corrupted data, it's often safer to force derived classes to access data in the base class using only public functions in the base class, just as ordinary main() programs must do. Using the protected specifier leads to simpler programming,

Base Class Unchanged

Remember that, even if other classes have been derived from it, the base class remains unchanged. In the main() part of Example1 , we could define objects of type Counter:

Counter c2; ← object of base class

Such objects would behave just as they would if CountDn didn't exist.

Note also that inheritance doesn't work in reverse. The base class and its objects don't know anything about any classes derived from the base class. In this example that means that objects of class Counter, such as c2, can't use the operator--() function in CountDn. If you want a counter that you can decrement, it must be of class CountDn, not Counter.

Derived Class Constructors

There's a potential glitch in the example1 program. What happens if we want to initialize a CountDn object to a value? Can the one-argument

constructor in Counter be used? The answer is no. As we saw in example1, the compiler will substitute a no-argument constructor from the base class, but it draws the line at more complex constructors. To make such a definition work we must write a new set of constructors for the derived class. This is shown in the example2.

Example 2:-Write an oop program to decrement the counter variable using constructor in the derived class.

```
#include <iostream.h>
class Counter
{
protected: //NOTE: not private
unsigned int count; //count
public:
Counter() : count() //constructor, no args
{ }
Counter(int c) : count(c) //constructor, one arg
{ }
unsigned int get_count() const //return count
{ return count; }
Counter operator ++ () //incr count (prefix)
{ return Counter(++count); }
};
////////////////////////////////////////////////////////////////////
class CountDn : public Counter
{
public:
CountDn() : Counter() //constructor, no args
{ }
CountDn(int c) : Counter(c) //constructor, 1 arg
{ }
CountDn operator -- () //decr count (prefix)
{ return CountDn(--count); }
};
////////////////////////////////////////////////////////////////////
int main()
{
CountDn c1; //class CountDn
CountDn c2(100);
cout << "\nc1=" << c1.get_count(); //display
cout << "\nc2=" << c2.get_count(); //display
++c1; ++c1; ++c1; //increment c1
cout << "\nc1=" << c1.get_count(); //display it
--c2; --c2; //decrement c2
cout << "\nc2=" << c2.get_count(); //display it
CountDn c3 = --c2; //create c3 from c2
cout << "\nc3=" << c3.get_count(); //display c3
cout << endl;
return 0;
}
```

This program uses two new constructors in the CountDn class. Here is the no-argument constructor:

CountDn() : Counter()

{ }

This constructor has an unfamiliar feature: the function name following the colon. This construction causes the CountDn() constructor to call the Counter() constructor in the base class. In main(), when we say

CountDn c1;

the compiler will create an object of type CountDn and then call the CountDn constructor to initialize it. This constructor will in turn call the Counter constructor, which carries out the work. The CountDn() constructor could add additional statements of its own, but in this case it doesn't need to, so the function body between the braces is empty.

The statement

CountDn c2(100);

in main() uses the one-argument constructor in CountDn. This constructor also calls the corresponding one-argument constructor in the base class:

CountDn(int c) : Counter(c) ← argument c is passed to Counter

{ }

This construction causes the argument c to be passed from CountDn() to Counter(), where it is used to initialize the object. In main(), after initializing the c1 and c2 objects, we increment one and decrement the other and then print the results.

The one-argument constructor is also used in an assignment statement.

CountDn c3 = --c2;

Overriding Member Functions

You can use member functions in a derived class that override—that is, have the same name as those in the base class. You might want to do this so that calls in your program work the same way for objects of both base and derived classes.

Example 3 "Arrays and Strings." That program modeled a stack, a simple data storage device. It allowed you to push integers onto the stack and pop them off. If you tried to push too many items onto the stack, the program might bomb, since data would be placed in memory beyond the end of the st[] array. Or if you tried to pop too many items, the results would be meaningless, since you would be reading data from memory locations outside the array.

To cure these defects we've created a new class, Stack2, derived from Stack. Objects of Stack2 behave in exactly the same way as those of Stack, except that you will be warned if you attempt to push too many items on the stack or if you try to pop an item from an empty stack.

Example 3:-Write an oo program to overload functions in base and derived stack classes.

```
#include <iostream.h>
#include <process.h> //for exit()
class stack
{
protected:          //NOTE: can't be private
enum { MAX = 3 };   //size of stack array
int st[MAX];        //stack: array of integers
int top;        //index to top of stack
public:
stack()         //constructor
{ top = -1; }
void push(int var)    //put number on stack
{ st[++top] = var; }
int pop()             //take number off stack
{ return st[top--]; }
};
////////////////////////////////////////////////////////////////
class stack2 : public stack
{
public:
void push(int var) //put number on stack
{
if(top >= MAX-1) //error if stack full
{ cout << "\nError: stack is full"; exit(1); }
stack::push(var);  //call push() in Stack class
}
int pop()        //take number off stack
{
if(top < 0)      //error if stack empty
{ cout << "\nError: stack is empty\n"; exit(1); }
return stack::pop(); //call pop() in Stack class
}
};
////////////////////////////////////////////////////////////////
```

```
void main()
{
stack2 s1;
s1.push(11);       //push some values onto stack
s1.push(22);
s1.push(33);
cout << endl << s1.pop();    //pop some values from stack
cout << endl << s1.pop();
cout << endl << s1.pop();
cout << endl << s1.pop();    //oops, popped one too many...
cout << endl;
}
```

Which Function Is Used?

The Stack2 class contains two functions, push() and pop(). These functions have the same names and the same argument and return types, as the functions in Stack. When we call these functions from main(), in statements like

s1.push(11);

how does the compiler know which of the two push() functions to use? Here's the rule: When the same function exists in both the base class and the derived class, the function in the derived class will be executed. (This is true of objects of the derived class. Objects of the base class don't know anything about the derived class and will always use the base class functions.) We say that the derived class function ***overrides*** the base class function. So in the preceding statement, since s1 is an object of class Stack2, the push() function in Stack2 will be executed, not the one in Stack.

The push() function in Stack2 checks to see whether the stack is full. If it is, it displays an error message and causes the program to exit. If it isn't, it calls the push() function in Stack. Similarly, the pop() function in Stack2 checks to see whether the stack is empty. If it is, it prints an error message and exits; otherwise, it calls the pop() function in Stack. In main() we push three items onto the stack, but we pop four. The last pop elicits an error message

The output

33

22

11

Error: stack is empty and terminates the program.

Scope Resolution with Overridden Functions

How do push() and pop() in Stack2 access push() and pop() in Stack? They use the scope resolution operator, ::, in the statements

Stack::push(var);

and

return Stack::pop();

These statements specify that the push() and pop() functions in Stack are to be called. Without the scope resolution operator, the compiler would think the push() and pop() functions in Stack2 were calling themselves, which—in this case—would lead to program failure. Using the scope resolution operator allows you to specify exactly what class the function is a member of.

Inheritance in the English Distance Class

Let's derive a new class from Distance. This class will add a single data item to our feet-and inches measurements: a sign, which can be positive or negative. When we add the sign, we'll also need to modify the member functions so they can work with signed distances.

Example 4:-Write an oo program to overload functions in base and derived distence classes.

```
#include <iostream.h>
enum posneg { pos, neg }; //for sign in DistSign
////////////////////////////////////////////////////////////////////
class Distance        //English Distance class
{
protected:          //NOTE: can't be private
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0)    //no-arg constructor
{ }
Distance(int ft, float in) : feet(ft), inches(in) //2-arg constructor)
{ }
void getdist()           //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() const //display distance
{ cout << feet << "-"<< inches ; }
};
////////////////////////////////////////////////////////////////////
class DistSign : public Distance //adds sign to Distance
{
private:
posneg sign; //sign is pos or neg
public:
//no-arg constructor
DistSign() : Distance() //call base constructor
{ sign = pos; }        //set the sign to +

//2- or 3-arg constructor
DistSign(int ft, float in, posneg sg=pos) :Distance(ft, in) //call base constructor
{ sign = sg; }       //set the sign
void getdist()       //get length from user
{
Distance::getdist();   //call base getdist()
char ch;             //get sign from user
cout << "Enter sign (+ or -): "; cin >> ch;
sign = (ch=='+'|'-') ? pos : neg;
}
void showdist() const     //display distance
{
cout << ( (sign==pos) ? "(+)" : "(-)" );     //show sign
Distance::showdist();   //feet and inche
}
};
////////////////////////////////////////////////////////////////
int main()
{
DistSign alpha;      //no-arg constructor
alpha.getdist();       //get alpha from user
DistSign beta(11, 6.25);   //2-arg constructor
DistSign gamma(100, 5.5, neg); //3-arg constructor
                //display all distances
cout << "\nalpha = "; alpha.showdist();
cout << "\nbeta = "; beta.showdist();
cout << "\ngamma = "; gamma.showdist();
cout << endl;
return 0;
}
```

Here the DistSign class adds the functionality to deal with signed numbers. The Distance class in this program is just the same as in previous programs, except that the data is protected.

Actually in this case it could be private, because none of the derived-class functions accesses it. However, it's safer to make it protected so that a derived-class function could access it if necessary.

Here's some sample output:

Enter feet: 6

Enter inches: 2.5

Enter sign (+ or -): -

alpha = (-)6'-2.5"

beta = (+)11'-6.25"

gamma = (-)100'-5.5"

The DistSign class is derived from Distance. It adds a single variable, sign, which is of type posneg. The sign variable will hold the sign of the distance. The posneg type is defined in an enum statement to have two possible values: pos and neg.

Constructors in DistSign

DistSign has two constructors, mirroring those in Distance. The first takes no arguments; the second takes either two or three arguments. The third, optional, argument in the second constructor is a sign, either pos or neg. Its default value is pos. These constructors allow us to define variables (objects) of type DistSign in several ways.

Both constructors in DistSign call the corresponding constructors in Distance to set the feet and- inches values. They then set the sign variable. The no-argument constructor always sets it to pos. The second constructor sets it to pos if no third-argument value has been provided, or to a value (pos or neg) if the argument is specified.

The arguments ft and in, passed from main() to the second constructor in DistSign, are simply forwarded to the constructor in Distance.

Member Functions in DistSign

Adding a sign to Distance has consequences for both of its member functions. The getdist() function in the DistSign class must ask the user for the sign as well as for feet-and-inches values, and the showdist() function must display the sign along with the feet and inches. These functions call the corresponding functions in Distance, in the lines

Distance::getdist();

and

Distance::showdist();

Class Hierarchies

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. 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.

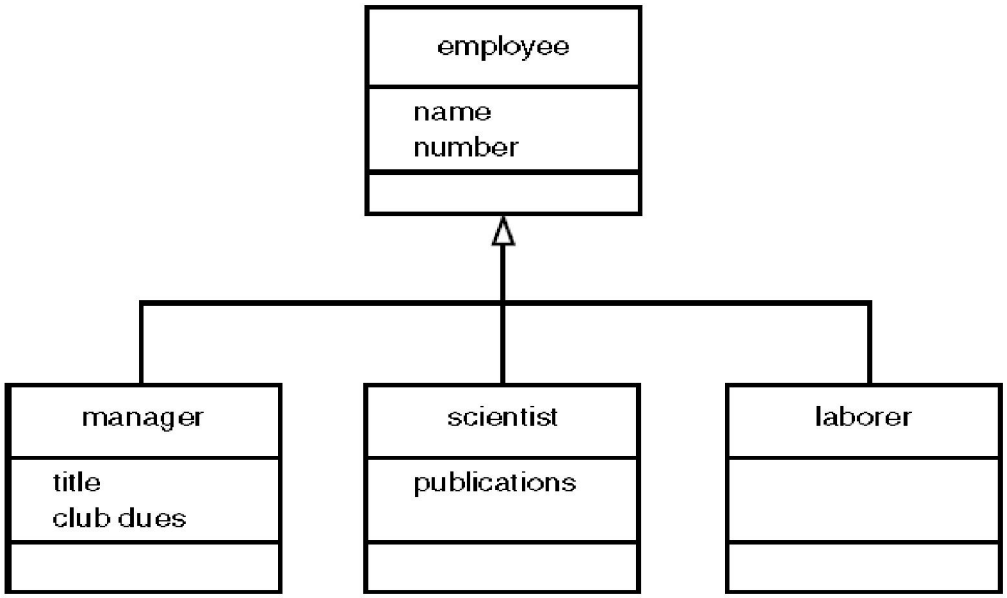


Figure 1: ***class diagram for EMPLOY.***

Example 1:-Write an oo program to model employ database using inheritance.

```
#include <iostream.h>
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
{
};
```

```
////////////////////////////////////////////////////////////////////
void 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();
}
```

The main() part of the program declares four objects of different classes: two managers, a scientist, and a laborer. (Of course many more employees of each type could be defined, but the output would become rather large.) It then calls the getdata() member functions to obtain information about each employee, and the putdata() function to display this information.

Output of program

Enter data for manager 1

Enter last name: Wainsworth

Enter number: 10

Enter title: President

Enter golf club dues: 1000000

Enter data on manager 2

Enter last name: Bradley

Enter number: 124

Enter title: Vice-President

Enter golf club dues: 500000

Enter data for scientist 1

Enter last name: Hauptman-Frenglish

Enter number: 234234

Enter number of pubs: 999

Enter data for laborer 1

Enter last name: Jones

Enter number: 6546544

The program then plays it back.

Data on manager 1

Name: Wainsworth

Number: 10

Title: President

Golf club dues: 1000000

Data on manager 2

Name: Bradley

Number: 124

Title: Vice-President

Golf club dues: 500000

Data on scientist 1

Name: Hauptman-Frenglish

Number: 234234

Number of publications: 999

Data on laborer 1

Name: Jones

Number: 6546544

"Abstract" Base Class

. It may seem that the laborer class is unnecessary, but by making it a separate class we emphasize that all classes are descended from the same source, employee. Also, if in the future we decided to modify the laborer class, we would not need to change the declaration for employee.

Constructors and Member Functions

There are no constructors in either the base or derived classes, so the compiler creates objects of the various classes automatically when it encounters definitions like manager m1, m2; using the default constructor for manager calling the default constructor for employee.

The getdata() and putdata() functions in employee accept a name and number from the user and display a name and number. Functions also called getdata() and putdata() in the manager and scientist classes use the functions in employee, and also do their own work.

In manager, the getdata() function asks the user for a title and the amount of golf club dues, and putdata() displays these values. In scientist, these functions handle the number of publications.

Access Combinations

There are so many possibilities for access that it's instructive to look at an example programthat shows what works and what doesn't. Here's the listing for Example 2:

Example 2:-

```
#include <iostream.h>
////////////////////////////////////////////////////////////////////
class A //base class
{
private:
int privdataA;    //(functions have the same access
protected:        //rules as the data shown here)
int protdataA;
public:
int pubdataA;
};
////////////////////////////////////////////////////////////////////
class B : public A   //publicly-derived class
{
public:
void funct()
{
int a;
a = privdataA;    //error: not accessible
a = protdataA;    //OK
a = pubdataA;     //OK
}
};
////////////////////////////////////////////////////////////////////
class C : private A //privately-derived class
{
public:
void funct()
{
int a;
a = privdataA;    //error: not accessible
a = protdataA;    //OK
a = pubdataA;     //OK
}
};
```

```
////////////////////////////////////////////////////////////////////
void main()
{
int a;
B objB;
a = objB.privdataA;    //error: not accessible
a = objB.protdataA;    //error: not accessible
a = objB.pubdataA;     //OK (A public to B)
C objC;
a = objC.privdataA; //error: not accessible
a = objC.protdataA; //error: not accessible
a = objC.pubdataA; //error: not accessible (A private to C)
}
```

The program specifies a base class, A, with private, protected, and public data items. Two classes, B and C, are derived from A. B is publicly derived and C is privately derived.

As we've seen before, functions in the derived classes can access protected and public data in the base class. Objects of the derived classes cannot access private or protected members of the base class.

What's new is the difference between publicly derived and privately derived classes. Objects of the publicly derived class B can access public members of the base class A, while objects of the privately derived class C cannot; they can only access the public members of their own derived class. This is shown in Figure 2.

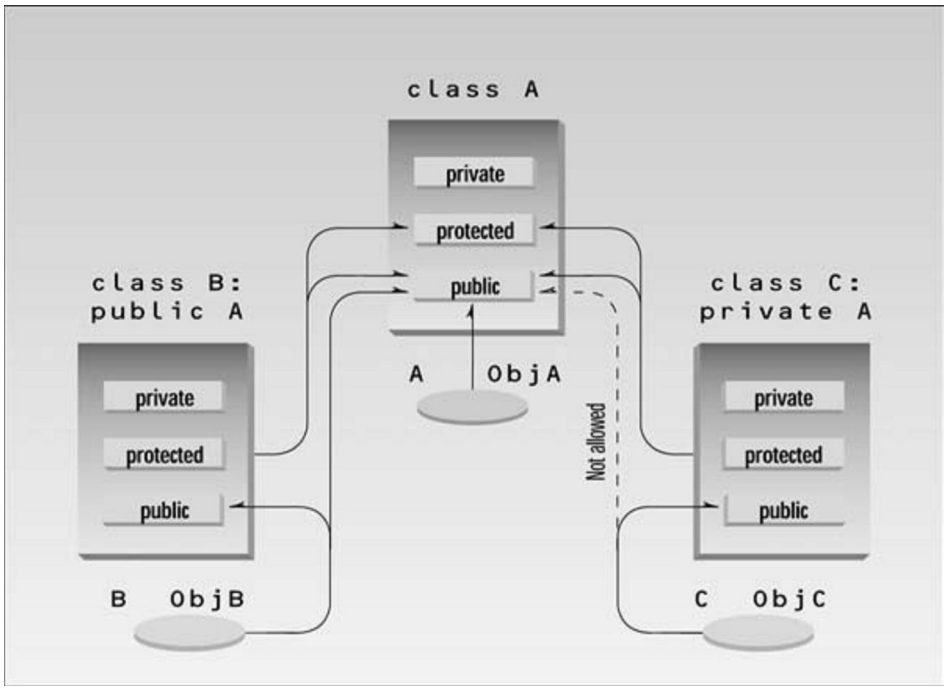


Figure2: ***Public and private derivation.***

Levels of Inheritance

Classes can be derived from classes that are themselves derived. Here's a miniprogram that shows the idea:

```
class A
{};
class B : public A
{};
class C : public B
{};
```

Here B is derived from A, and C is derived from B. The process can be extended to an arbitrary number of levels D could be derived from C, and so on. Suppose that we decided to add a special kind of laborer called a *foreman* to the EMPLOY program. Since a foreman is a kind of laborer, the foreman class is derived from the laborer class, as shown in Figure 3.

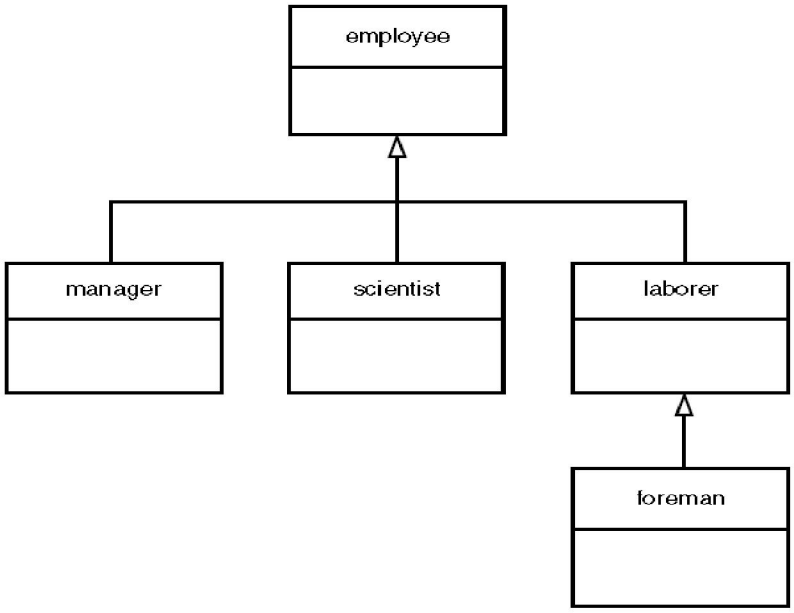


Figure 3: ***class diagram for EMPLOY2.***

Example 3:-Write an oo program to model employ database using multi levels inheritance.

```
#include <iostream.h>
const int LEN = 80; //maximum length of names
////////////////////////////////////////////////////////////////
class employee
{
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 //manager 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
{
};
////////////////////////////////////////////////////////////////
class foreman : public laborer     //foreman class
{
private:
float quotas;      //percent of quotas met successfully
public:
void getdata()
{
laborer::getdata();
cout << " Enter quotas: "; cin >> quotas;
}
void putdata() const
{
laborer::putdata();
cout << "\n Quotas: " << quotas;
}
};
////////////////////////////////////////////////////////////////
int main()
{
laborer l1;
foreman f1;
cout << endl;
cout << "\nEnter data for laborer 1";
l1.getdata();
cout << "\nEnter data for foreman 1";
f1.getdata();
cout << endl;
cout << "\nData on laborer 1";
l1.putdata();
cout << "\nData on foreman 1";
f1.putdata();
cout << endl;
return 0;
}
```

A class hierarchy results from generalizing common characteristics. The more general the class, the higher it is on the chart. Thus a laborer is more general than a foreman, who is a specialized kind of laborer, so laborer is

shown above foreman in the class hierarchy, although a foreman is probably paid more than a laborer.

Multiple Inheritances

A class can be derived from more than one base class. This is called *multiple inheritances*.

Figure 4 shows how this looks when a class C is derived from base classes A and B.

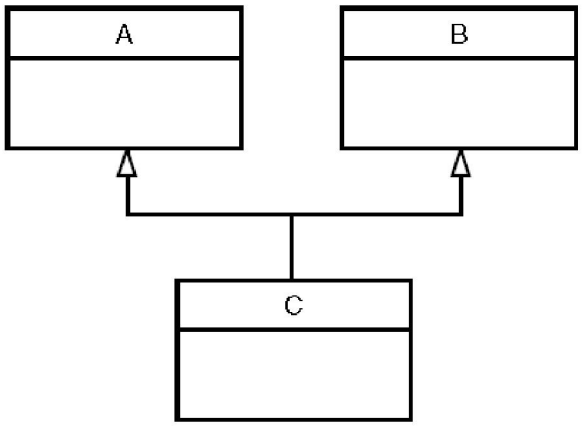


Figure 4: ***class diagram for multiple inheritances.***

The syntax for multiple inheritances is similar to that for single inheritance. In the situation shown in Figure 4, the relationship is expressed like this:

```
class A // base class A
{
};
class B // base class B
{
};
class C : public A, public B // C is derived from A and B
{
};
```

The base classes from which C is derived are listed following the colon in C's specification; they are separated by commas.

Member Functions in Multiple Inheritance

Suppose that we need to record the educational experience of some of the employees in the EMPLOY program. We've already developed a class called student that models students with different educational backgrounds. We decide that instead of modifying the employee class to incorporate educational data, we will add this data by multiple inheritances from the student class. The student class stores the name of the school or university last attended and the highest degree received. Both these data items are stored as strings. Two member functions, getedu() and putedu(), ask the user for this information and display it. Educational information is not relevant to every class of employee. We don't need to record the educational experience of laborers; it's only relevant for managers and scientists. We therefore modify manager and scientist so that they inherit from both the employee and student classes, as shown in Figure 5.

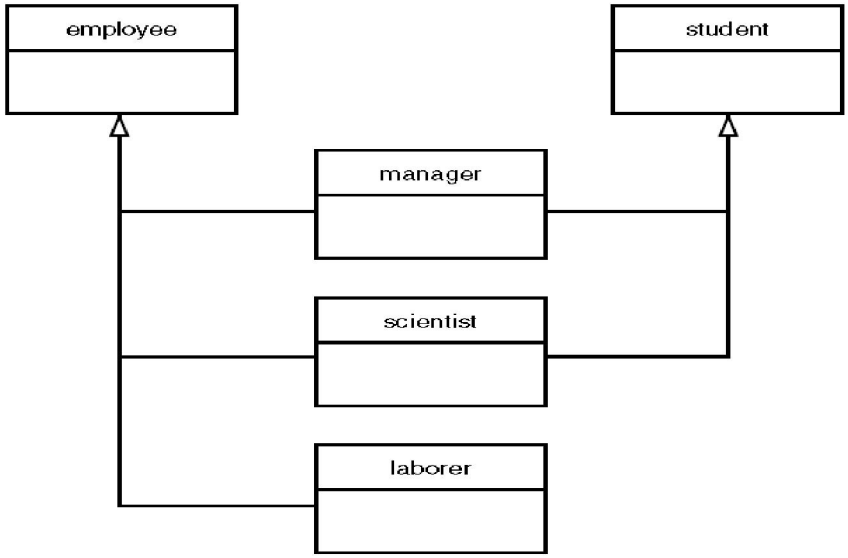


Figure5 :*UML class diagram for EMPMULT.*

Example 4:-Write an oo program to model employ database using multiple inheritances without using constructors.

```
#include <iostream.h>
const int LEN = 80; //maximum length of names
////////////////////////////////////////////////////////////////
class student //educational background
{
private:
char school[LEN]; //name of school or university
char degree[LEN]; //highest degree earned
public:
void getedu()
{
cout << " Enter name of school or university: ";
cin >> school;
cout << " Enter highest degree earned \n";
cout << " (Highschool, Bachelor's, Master's, PhD): ";
cin >> degree;
}
void putedu() const
{
cout << "\n School or university: " << school;
cout << "\n Highest degree earned: " << degree;
}
};
////////////////////////////////////////////////////////////////
class employee
{
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 : private employee, private student //management
{
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;
student::getedu();
}
void putdata() const
{
employee::putdata();
cout << "\n Title: " << title;
cout << "\n Golf club dues: " << dues;
student::putedu();
}
};
////////////////////////////////////////////////////////////////////
class scientist : private employee, private student //scientist
{
private:
int pubs; //number of publications
public:
void getdata()
{
employee::getdata();
cout << " Enter number of pubs: "; cin >> pubs;
student::getedu();
}
void putdata() const
{
employee::putdata();
cout << "\n Number of publications: " << pubs;
student::putedu();
}
};
```

```
////////////////////////////////////////////////////////////////////
class laborer : public employee //laborer
{
};
////////////////////////////////////////////////////////////////////
int main()
{
manager m1;
scientist s1, s2;
laborer l1;
cout << endl;
cout << "\nEnter data for manager 1"; //get data for
m1.getdata();            //several employees
cout << "\nEnter data for scientist 1";
s1.getdata();
cout << "\nEnter data for scientist 2";
s2.getdata();
cout << "\nEnter data for laborer 1";
l1.getdata();
cout << "\nData on manager 1";   //display data for
m1.putdata(); //several employees
cout << "\nData on scientist 1";
s1.putdata();
cout << "\nData on scientist 2";
s2.putdata();
cout << "\nData on laborer 1";
l1.putdata();
cout << endl;
return 0;
}
```

The getdata() and putdata() functions in the manager and scientist classes incorporate calls to functions in the student class, such as

student::getedu();

and

student::putedu();

These routines are accessible in manager and scientist because these classes are descended from student.

Here's some sample interaction with Eexample4:

Enter data for manager 1

Enter last name: Bradley

Enter number: 12

Enter title: Vice-President

Enter golf club dues: 100000

Enter name of school or university: Yale

Enter highest degree earned

(Highschool, Bachelor's, Master's, PhD): Bachelor's

Enter data for scientist 1

Enter last name: Twilling

Enter number: 764

Enter number of pubs: 99

Enter name of school or university: MIT

Enter highest degree earned

(Highschool, Bachelor's, Master's, PhD): PhD

Enter data for scientist 2

Enter last name: Yang

Enter number: 845

Enter number of pubs: 101

Enter name of school or university: Stanford

Enter highest degree earned

(Highschool, Bachelor's, Master's, PhD): Master's

Enter data for laborer 1

Enter last name: Jones

Enter number: 48323

Private Derivation in EMPMULT

The manager and scientist classes in EMPMULT are privately derived from the employee and student classes. There is no need to use public derivation because objects of manager and scientist never call routines in the employee and student base classes. However, the laborer class must be publicly

derived from employer, since it has no member functions of its own and relies on those in employee.

Example 5:-Write an oo program to compute distance using constructors in multiple inheritances.

```
#include <iostream.h>
typedef   char *String;
#include <string.h>
//////////////////////////////////////////////////////////////////////
class Type   //type of lumber
{
private:
String dimensions;
String grade;
public:    //no-arg constructor
Type() : dimensions("N/A"), grade("N/A")
{ }
           //2-arg constructor
Type(String di,String gr) : dimensions(di), grade(gr)
{ }
void gettype()  //get type from user
{
dimensions="2x2";

grade="ABC";
}
void showtype() const //display type
{
cout << "\n Dimensions: " << dimensions;
cout << "\n Grade: " << grade;
}
};
```

```
//////////////////////////////////////////////////////////////////////
class Distance //English Distance class
{
private:
int feet;
float inches;
public: //no-arg constructor
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 << " Enter feet: "; cin >> feet;
cout << " Enter inches: "; cin >> inches;
}
void showdist() const //display distance
{ cout << feet << "-" << inches ; }
};
//////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////
class Lumber : public Type, public Distance
{
private:
int quantity; //number of pieces
double price; //price of each piece
public: //constructor (no args)
Lumber() : Type(), Distance(), quantity(0), price(0.0)
{ }
Lumber(String di, String gr,int ft, float in, int qu, float prc ) : Type(di, gr), Distance(ft, in), quantity(qu), price(prc)
{ }
void getlumber()
{
Type::gettype();
Distance::getdist();
cout << " Enter quantity: "; cin >> quantity;
cout << " Enter price per piece: "; cin >> price;
}
void showlumber() const
{
Type::showtype();
cout << "\n Length: ";
Distance::showdist();
cout << "\n Price for " << quantity
<< " pieces: $" << price * quantity;
}
};
////////////////////////////////////////////////////////////////
void main()
{
Lumber siding;   //constructor (no args)
cout << "\nSiding data:\n";
siding.getlumber(); //get siding from user
Lumber studs( "2x4", "const", 8, 0.0, 200, 4.45F ); //constructor (6 args)
cout << "\nSiding"; siding.showlumber();   //display lumber data
cout << "\nStuds"; studs.showlumber();
cout << endl;
}
```

No-Argument Constructor

The no-argument constructor in Type looks like this:

Type()

{ strcpy(dimensions, “N/A”); strcpy(grade, “N/A”); }

This constructor fills in “N/A” (not available) for the dimensions and grade variables so the user will be made aware if an attempt is made to display data for an uninitialized lumber object. You’re already familiar with the no-argument constructor in the Distance class:

Distance() : feet(0), inches(0.0)

{ }

The no-argument constructor in Lumber calls both of these constructors.

Lumber() : Type(), Distance(), quantity(0), price(0.0)

{ }

The names of the base-class constructors follow the colon and are separated by commas. When the Lumber() constructor is invoked, these base-class constructors—Type() and Distance()— will be executed. The quantity and price attributes are also initialized.

Multi-Argument Constructors

Here is the two-argument constructor for Type:

Type(string di, string gr) : dimensions(di), grade(gr)

{ }

This constructor copies string arguments to the dimensions and grade member data items. Here’s the constructor for Distance, which is again familiar from previous programs:

Distance(int ft, float in) : feet(ft), inches(in)

{ }

The constructor for Lumber calls both of these constructors, so it must supply values for their arguments. In addition it has two arguments of its own: the quantity of lumber and the unit price.

Thus this constructor has six arguments. It makes two calls to the two constructors, each of which takes two arguments, and then initializes its own two data items. Here's what it looks like:

```
Lumber( string di, string gr, //args for Type
int ft, float in, //args for Distance
int qu, float prc ) :   //args for our data
Type(di, gr),     //call Type ctor
Distance(ft, in), //call Distance ctor
quantity(qu), price(prc) //initialize our data
{ }
```

Ambiguity in Multiple Inheritances

There are two types ambiguity in Multiple Inheritances

1. Two base classes have functions with the same name, while a class derived from both base classes has no function with this name. How do objects of the derived class access the correct base class function? The name of the function alone is insufficient, since the compiler can't figure out which of the two functions is meant.

Example: demonstrates ambiguity in multiple inheritance

```
#include <iostream.h>
class A
{
public:
void show() { cout << "Class A\n"; }
};
class B
```

```
{
public:
void show() { cout << "Class B\n"; }
};
class C : public A, public B
{
};
//////////////////////////////////////////////////////////////////
int main()
{
C objC; //object of class C
// objC.show(); //ambiguous--will not compile
objC.A::show(); //OK
objC.B::show(); //OK
return 0;
}
```

The problem is resolved using the scope-resolution operator to specify the class in which the function lies. Thus

objC.A::show();

refers to the version of show() that's in the A class, while

objC.B::show();

refers to the function in the B class.

2. Another kind of ambiguity arises if you derive a class from two classes that are each derived from the same class. This creates a diamond-shaped inheritance tree.

Example: investigates diamond-shaped multiple inheritance

```
#include <iostream.h>
class A
{
public:
void func();
};
class B : public A
{ };
class C : public A
{ };
class D : public B, public C
{ };
//////////////////////////////////////////////////////////////
int main()
{
D objD;
objD.func();      //ambiguous: won't compile
return 0;
}
```

Classes B and C are both derived from class A, and class D is derived by multiple inheritance from both B and C. Trouble starts if you try to access a member function in class A from an object of class D. In this example objD tries to access func(). However, both B and C contain a copy of func(), inherited from A. The compiler can't decide which copy to use, and signals an error.

Function Template

Suppose you want to write a function that returns the absolute value of two numbers. The absolute value of a number is its value without regard to its sign: The absolute value of 3 is 3, and the absolute value of –3 is also 3. Ordinarily this function would be written for a particular data type:

```
int abs(int n) //absolute value of ints
{
return (n<0) ? -n : n; //if n is negative, return -n
}
```

Here the function is defined to take an argument of type int and to return a value of this same type. But now suppose you want to find the absolute value of a type long. You will need to write a completely new function:

```
long abs(long n) //absolute value of longs
{
return (n<0) ? -n : n;
}
```

And again, for type float:

```
float abs(float n) //absolute value of floats
{
return (n<0) ? -n : n;
}
```

The body of the function is written the same way in each case, but they are completely different functions because they handle arguments and return values of different types. It's true that in C++ these functions can all be overloaded to have the same name, but you must nevertheless write a

separate definition for each one. (In the C language, which does not support overloading, functions for different types can't even have the same name. In the C function library this leads to families of similarly named functions, such as abs(), fabs(), labs(), and cabs().

Rewriting the same function body over and over for different types is time-consuming and wastes space in the listing. Also, if you find you've made an error in one such function, you'll need to remember to correct it in each function body. Failing to do this correctly is a good way to introduce inconsistencies into your program. It would be nice if there were a way to write such a function just once, and have it work for many different data types. This is exactly what function templates do for you. The idea is shown schematically in Figure 1.

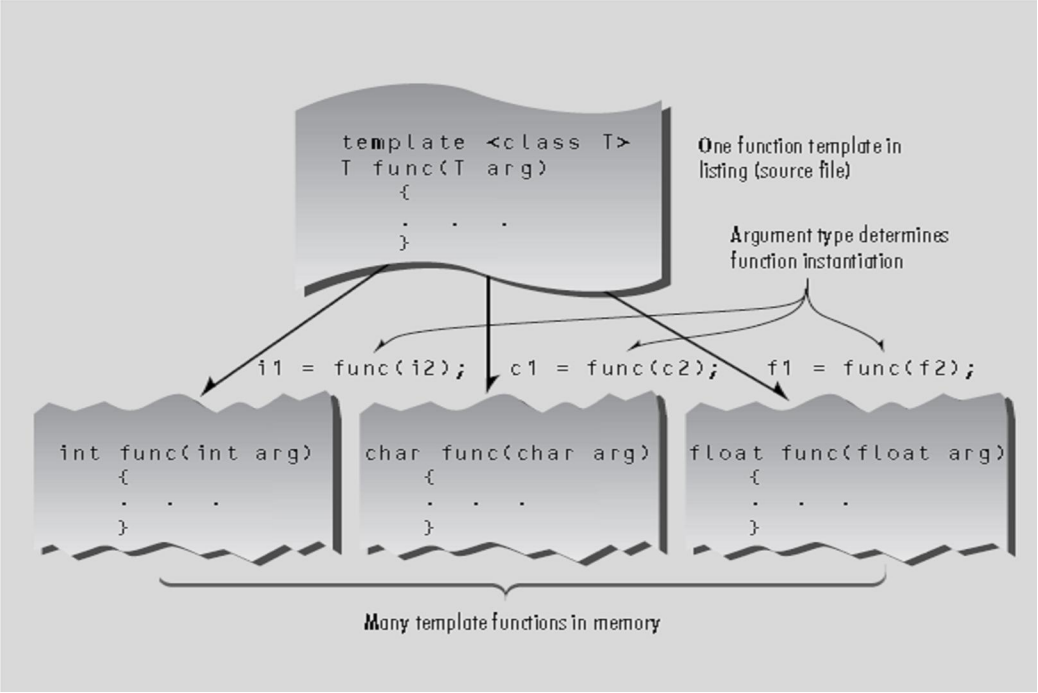


Figure1: ***A function template.***

A Simple Function Template

The first example shows how to write our absolute-value function as a template, so that it will work with any basic numerical type. This program defines a template version of abs() and then, in main(), invokes this function with different data types to prove that it works.

Example 1:Write an OO Program to find the absolute value using template function

```
#include <iostream.h>
//-----------------------------------------------------------------
template <class T> //function template
T abs(T n)
{
return (n < 0) ? -n : n;
}
//-----------------------------------------------------------------
int main()
{
int int1 = 5;
int int2 = -6;
long lon1 = 70000L;
long lon2 = -80000L;
double dub1 = 9.95;
double dub2 = -10.15;
//calls instantiate functions
cout << "\nabs(" << int1 << ")=" << abs(int1); //abs(int)
cout << "\nabs(" << int2 << ")=" << abs(int2); //abs(int)
cout << "\nabs(" << lon1 << ")=" << abs(lon1); //abs(long)
cout << "\nabs(" << lon2 << ")=" << abs(lon2); //abs(long)
cout << "\nabs(" << dub1 << ")=" << abs(dub1); //abs(double)
cout << "\nabs(" << dub2 << ")=" << abs(dub2); //abs(double)
cout << endl;
return 0;
}
```

the output of the program:

abs(5)=5
abs(-6)=6
abs(70000)=70000
abs(-80000)=80000
abs(9.95)=9.95
abs(-10.15)=10.15

The **abs()** function now works with all three of the data types (int, long, and double) that we use as arguments. It will work on other basic numerical types as well, and it will even work on user-defined data types, provided that the less-than operator (<) and the unary minus operator (-) are appropriately overloaded.

Here's how we specify the **abs()** function to work with multiple data types:

```
template <class T> //function template
T abs(T n)
{
return (n<0) ? -n : n;
}
```

This entire syntax, with a first line starting with the keyword template and the function definition following, is called a ***function template***. How does this new way of writing abs() give it such amazing flexibility?

Function Template Syntax

The key innovation in function templates is to represent the data type used by the function not as a specific type such as int, but by a name that can stand for *any* type. In the preceding function template, this name is T. The template keyword signals the compiler that we're about to define a function template. The keyword class, within the angle brackets, might just as well be called type. You can define your own data types using classes, so there's really no distinction between types and classes. The variable following the keyword class (T in this example) is called the ***template argument***.

Throughout the definition of the template, whenever a specific data type such as **int** would ordinarily be written, we substitute the template argument, T. In the abs() template this name appears only twice, both in the first line (the function declarator), as the argument type and return type. In more complex functions it may appear numerous times throughout the function body as well.

What the Compiler Does

What does the compiler do when it sees the template keyword and the function definition that follows it? The function template itself doesn't cause the compiler to generate any code. It can't generate code because it doesn't know yet what data type the function will be working with. It simply remembers the template for possible future use.

Code generation doesn't take place until the function is actually called (invoked) by a statement within the program. In example 1 this happens in expressions like abs(int1) in the statement

cout << "\nabs(" << int << ")=" << abs(int1);

When the compiler sees such a function call, it knows that the type to use is int, because that's the type of the argument int1. So it generates a specific version of the abs() function for type int, substituting int wherever it sees the name T in the function template. This is called ***instantiating*** the function template, and each instantiated version of the function is called a ***template function***. (That is, a *template function* is a specific instance of a *function template*.)The compiler also generates a call to the newly instantiated function, and inserts it into the code where abs(int1) is. Similarly, the

expression abs(lon1) causes the compiler to generate a version of abs() that operates on type long and a call to this function, while the abs(dub1) call generates a function that works on type double.

Function Templates with Multiple Arguments

The example bellow takes three arguments: two that are template arguments and one of a basic type. The purpose of this function is to search an array for a specific value. The function returns the array index for that value if it finds it, or –1 if it can't find it. The arguments are a pointer to the array, the value to search for, and the size of the array.

Example 2:Write an OO Program to find number in the array value using template function

```
#include <iostream.h>
//-----------------------------------------------------------------
//function returns index number of item, or -1 if not found
template <class atype>
int find(atype* array, atype value, int size)
{
for(int j=0; j<size; j++)
if(array[j]==value)
return j;
return -1;
}
//-----------------------------------------------------------------
char chrArr[] = {1, 3, 5, 9, 11, 13}; //array
char ch = 5; //value to find
int intArr[] = {1, 3, 5, 9, 11, 13};
int in = 6;
long lonArr[] = {1L, 3L, 5L, 9L, 11L, 13L};
long lo = 11L;
double dubArr[] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};
double db = 4.0;
int main()
{
cout << "\n 5 in chrArray: index=" << find(chrArr, ch, 6);
cout << "\n 6 in intArray: index=" << find(intArr, in, 6);
cout << "\n11 in lonArray: index=" << find(lonArr, lo, 6);
cout << "\n 4 in dubArray: index=" << find(dubArr, db, 6);
cout << endl;
return 0;
}
```

The output of program

5 in chrArray: index=2
6 in intArray: index=-1
11 in lonArray: index=4
4 in dubArray: index=-1

Class Templates

The template concept can be extended to classes. Class templates are generally used for data storage (container) classes. The Stack class," for example, could store data only of type int. Here's a condensed version of that class.

```
class Stack
{
private:
int st[MAX]; //array of ints
int top; //index number of top of stack
public:
Stack(); //constructor
void push(int var); //takes int as argument
int pop(); //returns int value
};
```

If we wanted to store data of type long in a stack, we would need to define a completely new class:

```
class LongStack
{
private:
long st[MAX]; //array of longs
int top; //index number of top of stack
public:
LongStack(); //constructor
void push(long var); //takes long as argument
long pop(); //returns long value
};
```

Similarly, we would need to create a new stack class for every data type we wanted to store.

Example 3:Write an OO Program to implement stack class using template class

```
#include <iostream.h>
const int MAX = 100; //size of array
////////////////////////////////////////////////////////////////
template <class Type>
class Stack
{
private:
Type st[MAX]; //stack: array of any type
int top; //number of top of stack
public:
Stack() //constructor
{ top = -1; }
void push(Type var) //put number on stack
{ st[++top] = var; }
Type pop() //take number off stack
{ return st[top--]; }
};
////////////////////////////////////////////////////////////////
int main()
{
Stack<float> s1; //s1 is object of class Stack<float>
s1.push(1111.1F); //push 3 floats, pop 3 floats
s1.push(2222.2F);
s1.push(3333.3F);
cout << "1: " << s1.pop() << endl;
cout << "2: " << s1.pop() << endl;
cout << "3: " << s1.pop() << endl;
Stack<long> s2; //s2 is object of class Stack<long>
s2.push(123123123L); //push 3 longs, pop 3 longs
s2.push(234234234L);
s2.push(345345345L);
cout << "1: " << s2.pop() << endl;
cout << "2: " << s2.pop() << endl;
cout << "3: " << s2.pop() << endl;
return 0;
}
```

Here's the output:

1: 3333.3 //float stack
2: 2222.2
3: 1111.1
1: 345345345 //long stack
2: 234234234
3: 123123123

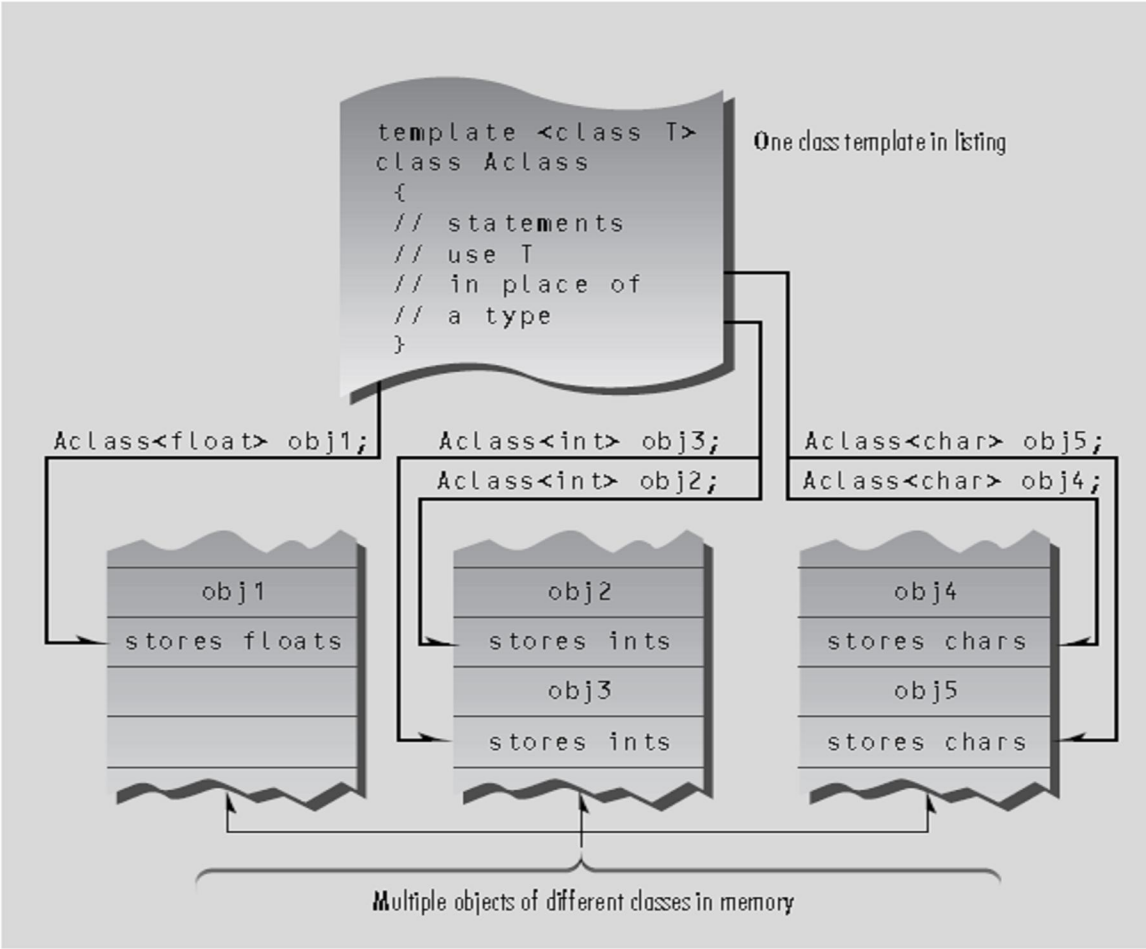


Figure2: ***A class template.***

<u>Class Name Depends on Context</u>

In the example 3, the member functions of the class template were all defined within the class. If the member functions are defined externally (outside of the class specification), we need a new syntax. The next program shows how this works.

Example 4: Write an OO Program to implement stack class using template class with scope operator.

```
#include <iostream.h>
const int MAX = 100;
////////////////////////////////////////////////////////////////
template <class Type>
class Stack
{
private:
Type st[MAX]; //stack: array of any type
int top; //number of top of stack
public:
Stack(); //constructor
void push(Type var); //put number on stack
Type pop(); //take number off stack
};
////////////////////////////////////////////////////////////////
template<class Type>
Stack<Type>::Stack() //constructor
{
top = -1;
}
//--------------------------------------------------------------
template<class Type>
void Stack<Type>::push(Type var) //put number on stack
{
st[++top] = var;
}
//--------------------------------------------------------------
template<class Type>
Type Stack<Type>::pop() //take number off stack
{
return st[top--];
}
//--------------------------------------------------------------
int main()
{
Stack<float> s1; //s1 is object of class Stack<float>
s1.push(1111.1F); //push 3 floats, pop 3 floats
s1.push(2222.2F);
s1.push(3333.3F);
cout << "1: " << s1.pop() << endl;
cout << "2: " << s1.pop() << endl;
cout << "3: " << s1.pop() << endl;

Stack<long> s2; //s2 is object of class Stack<long>
s2.push(123123123L); //push 3 longs, pop 3 longs
s2.push(234234234L);
s2.push(345345345L);
cout << "1: " << s2.pop() << endl;
cout << "2: " << s2.pop() << endl;
cout << "3: " << s2.pop() << endl;
return 0;
}
```

The expression template***<class Type>*** must precede not only the class definition, but each externally defined member function as well. Here's how the push() function looks:

```
template<class Type>
void Stack<Type>::push(Type var)
{
st[++top] = var;
}
```

The name Stack<Type> is used to identify the class of which push() is a member. In a normal non-template member function the name Stack alone would suffice:

```
void Stack::push(int var) //Stack() as a non-template function
{
st[++top] = var;
}
```

but for a function template we need the template argument as well: Stack<Type>.

Thus we see that the name of the template class is expressed differently in different contexts.

Within the class specification, it's simply the name itself: Stack. For externally defined member functions, it's the class name plus the template argument name: Stack<Type>. When you define actual objects for storing a specific data type, it's the class name plus this specific type:

Stack<float>, for example.

```
class Stack //Stack class specifier
{ };
void Stack<Type>::push(Type var) //push() definition
{ }
Stack<float> s1; //object of type Stack<float>
```

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. 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.

1) First, all the different classes of shapes, such as balls and triangles, must be descended from a single base class.
2) Second, the draw() function must be declared to be virtual in the base class.

Normal Member Functions Accessed with Pointers

Example 1 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.

Example 1:

```
#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 . 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 not virtual 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 :**

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 1

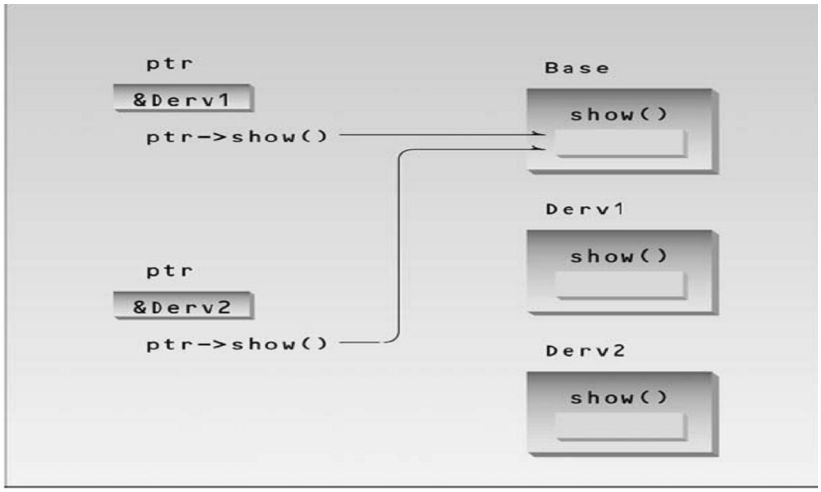


Figure 1 ***Nonvirtual pointer access.***

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.

Example 2:

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

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 not virtual. This is shown in Figure 2

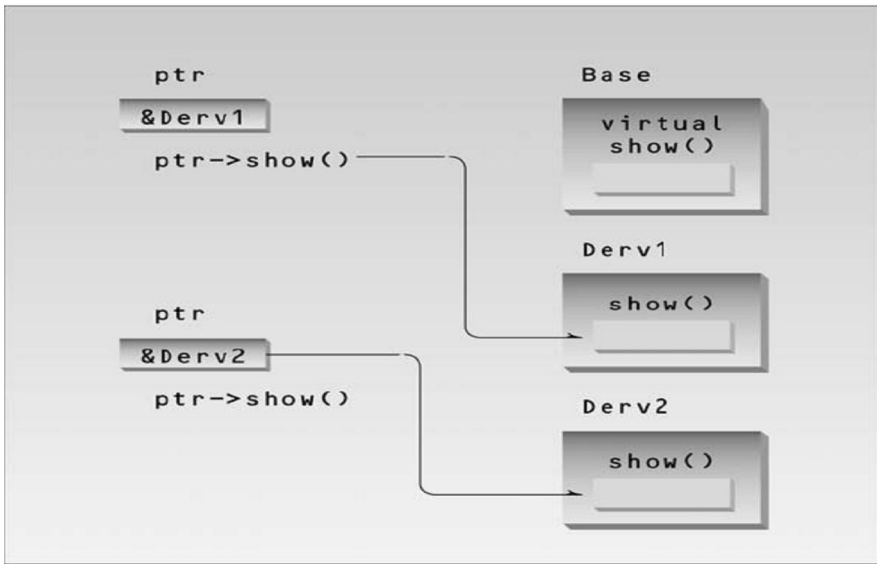


Figure 2 ***Virtual pointer access.***

<u>Late Binding</u>

The astute reader may wonder how the compiler knows what function to compile. In not virtual the compiler has no problem with the expression **ptr->show();** It always compiles a call to the show() function in the base class. But in virtual the compiler doesn't know what class the contents of ptr may contain. It could be the address of an object of the Derv1 class or of the Derv2 class. Which version of draw() does the compiler call? In fact the

compiler doesn't know what to do, so it arranges for the decision to be deferred until the program is running.

At runtime, when it is known what class is pointed to by ptr, the appropriate version of draw will be called. This is called ***late binding*** or ***dynamic binding***.

(Choosing functions in the normal way, during compilation, is called ***early binding*** or ***static binding***.) Late binding requires some overhead but provides increased power and flexibility.

Abstract Classes and Pure Virtual Functions

When we will never want to instantiate objects of a base class, we call it an *abstract class*. Such a class exists only to act as a parent of derived classes that will be used to instantiate objects. It may also provide an interface for the class hierarchy. By placing at least one ***pure virtual function*** in the base class.

A pure virtual function is one with the expression =0 added to the declaration. This is shown in the example3.

Example 3:

```
#include <iostream.h>
////////////////////////////////////////////////////////////////
class Base //base class
{
public:
virtual void show() = 0; //pure virtual function
};
////////////////////////////////////////////////////////////////
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()
{
// Base bad; //can´t make object from abstract class
Base* arr[2]; //array of pointers to base class
Derv1 dv1; //object of derived class 1
Derv2 dv2; //object of derived class 2
arr[0] = &dv1; //put address of dv1 in array
arr[1] = &dv2; //put address of dv2 in array
arr[0]->show(); //execute show() in both objects
arr[1]->show();
return 0;
}
```

Here the virtual function show() is declared as **virtual void show() = 0; // pure virtual function** .The equal sign here has nothing to do with assignment; the value 0 is not assigned to anything. The =0 syntax is simply how we tell the compiler that a virtual function will be pure. Now if in main() you attempt to create objects of class Base, the compiler will complain that you're trying to instantiate an object of an abstract class. It will also tell you the name of the pure virtual function that makes it an abstract class. Notice that, although this is only a declaration, you never need to write a definition of the base class show().

Virtual Functions and the person Class

Now that we understand some of the mechanics of virtual functions, let's look at a situation where it makes sense to use them. It uses the person class, with two derived classes, student and professor. These derived classes each contain a function called **isOutstanding()**. The person class is an abstract class because it contains the pure virtual functions getData() and isOutstanding().

Example 4:

```
#include <iostream.h>
////////////////////////////////////////////////////////////////
class person //person class
{
protected:
char name[40];
public:
void getName()
{ cout << " Enter name: "; cin >> name; }
void putName()
{ cout << "Name is: " << name << endl; }
virtual void getData() = 0; //pure virtual func
virtual bool isOutstanding() = 0; //pure virtual func
};
////////////////////////////////////////////////////////////////
class student : public person //student class
{
private:
float gpa; //grade point average
public:
void getData() //get student data from user
{
person::getName();
cout << " Enter student's GPA: "; cin >> gpa;
}
bool isOutstanding()
{ return (gpa > 3.5) ? true : false; }
};
////////////////////////////////////////////////////////////////
class professor : public person //professor class
{
private:
int numPubs; //number of papers published
public:
void getData() //get professor data from user
{
person::getName();
cout << " Enter number of professor's publications: ";
cin >> numPubs;
}
```

```
bool isOutstanding()
{ return (numPubs > 100) ? true : false; }
};
int main()
{
person* persPtr[100]; //array of pointers to persons
int n = 0; //number of persons on list
char choice;
do {
cout << "Enter student or professor (s/p): ";
cin >> choice;
if(choice=='s') //put new student
persPtr[n] = new student; // in array
else //put new professor
persPtr[n] = new professor; // in array
persPtr[n++]->getData(); //get data for person
cout << " Enter another (y/n)? "; //do another person?
cin >> choice;
} while( choice=='y' ); //cycle until not 'y'
for(int j=0; j<n; j++) //print names of all
{ //persons, and
persPtr[j]->putName(); //say if outstanding
if( persPtr[j]->isOutstanding() )
cout << " This person is outstanding\n";
}
return 0;
} //end main()
```

Here's some sample interaction:

Enter student or professor (s/p): s

Enter name: Timmy

Enter student's GPA: 1.2

Enter another (y/n)? y

Enter student or professor (s/p): s

Enter name: Brenda

Enter student's GPA: 3.9

Enter another (y/n)? y

Enter student or professor (s/p): s

Enter name: Sandy

Enter student's GPA: 2.4

Enter another (y/n)? y

Enter student or professor (s/p): p

Enter name: Shipley

Enter number of professor's publications: 714

Enter another (y/n)? y

Enter student or professor (s/p): p

Enter name: Wainright

Enter number of professor's publications: 13

Enter another (y/n)? n

Name is: Timmy

Name is: Brenda

This person is outstanding

Name is: Sandy

Name is: Shipley

This person is outstanding

Name is: Wainright

Virtual Destructors

Base class destructors should always be virtual. Suppose you use delete with a base class pointer to a derived class object to destroy the derived-class object. If the base-class destructor is not virtual then delete, like a normal member function, calls the destructor for the base class, not the destructor for the derived class. This will cause only the base part of the object to be destroyed. The program shows how this looks.

```
#include <iostream.h>
class Base
{
```

```
public:
~Base() //non-virtual destructor
// virtual ~Base() //virtual destructor
{ cout << "Base destroyed\n"; }
};
class Derv : public Base
{
public:
~Derv()
{ cout << "Derv destroyed\n"; }
};
//////////////////////////////////////////////////////////////////
int main()
{
Base* pBase = new Derv;
delete pBase;
return 0;
}
```

The output for this program as written is **Base destroyed**

This shows that the destructor for the Derv part of the object isn't called. In the listing the base class destructor is not virtual, but you can make it so by commenting out the first definition for the destructor and substituting the second.

Now the output is

Derv destroyed

Base destroyed

Now both parts of the derived class object are destroyed properly. Of course, if none of the destructors has anything important to do (like deleting memory obtained with new) then virtual destructors aren't important. But in general, to ensure that derived-class objects are destroyed properly, you should make destructors in all base classes virtual.

Most class libraries have a base class that includes a virtual destructor, which ensures that all derived classes have virtual destructors.

Virtual Base Classes

Consider the situation shown in Figure 3, with a base class, Parent; two derived classes, Child1 and Child2; and a fourth class, Grandchild, derived from both Child1 and Child2.

In this arrangement a problem can arise if a member function in the Grandchild class wants to access data or functions in the Parent class.

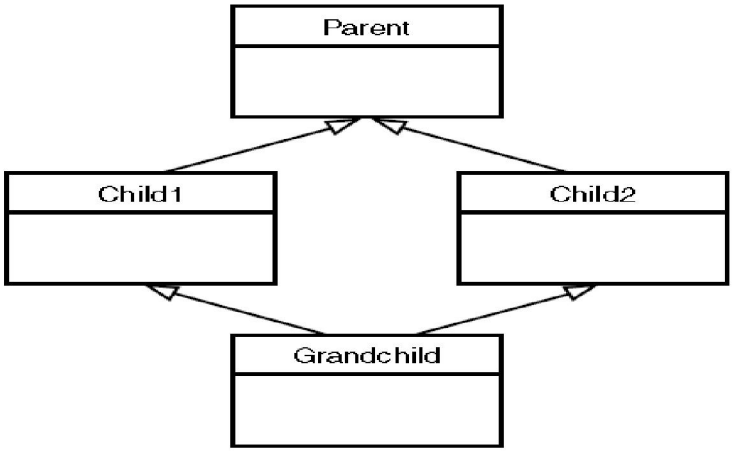


Figure 3 ***Virtual base classes.***

// ambiguous reference to base class

class Parent

{

```
protected:
int basedata;
};
class Child1 : public Parent
{ };
class Child2 : public Parent
{ };
class Grandchild : public Child1, public Child2
{
public:
int getdata()
{ return basedata; } // ERROR: ambiguous
};
```

A compiler error occurs when the getdata() member function in Grandchild attempts to access basedata in Parent. Why? When the Child1 and Child2 classes are derived from Parent, each inherits a copy of Parent; this copy is called a *subobject*. Each of the two subobjects contains its own copy of Parent's data, including basedata. Now, when Grandchild refers to basedata, which of the two copies will it access? The situation is ambiguous, and that's what the compiler reports.

To eliminate the ambiguity, we make Child1 and Child2 into virtual base classes, as shown by the example bellow.

```
 // virtual base classes
class Parent
{
```

```
protected:
int basedata;
};
class Child1 : virtual public Parent // shares copy of Parent
{ };
class Child2 : virtual public Parent // shares copy of Parent
{ };
class Grandchild : public Child1, public Child2
{
public:
int getdata()
{ return basedata; } // OK: only one copy of Parent
};
```

The use of the keyword virtual in these two classes causes them to share a single common subobject of their base class Parent. Since there is only one copy of base data, there is no ambiguity when it is referred to in Grandchild. The need for virtual base classes may indicate a conceptual problem with your use of multiple inheritances, so they should be used with caution.

Polymorphism

Polymorphism is one of the crucial features of object oriented programming. It simply means "*one name, multiple forms*". However, polymorphism allows an entity (variable, function or object) to take a variety of representations (take a multiple forms).

Polymorphism refers to situation in which objects belonging to different classes can respond to the same message, usually in different ways. For example suppose we have classes box, triangle and circle, whose objects represent the corresponding geometrical figures. The objects of these classes might all understand a message draw (), which causes an object to draw the corresponding figure on the screen.

In C++ Polymorphism is implemented via ***virtual functions.***

Therefore we have to distinguish different three types of polymorphism:

- Polymorphism of Variables (or Members).
- Polymorphism of Functions (or Methods).
- Polymorphism of Objects.

A. Polymorphism of Variables:

The first type of polymorphism is similar to the concept of dynamic binding. Here, the type of a variable depends on its content. Thus, its type depends on the content at a specific time:

```
int     a=5;    //use a as integer
…..
char    a='g';  //use a as character
……
```

B. Polymorphism of Functions:

Another type of polymorphism can be defined for functions. For example, suppose you want to define a function *isNull( )* which returns TRUE if its argument is zero and FALSE otherwise. For integer numbers this is easy:

```
Bool  isNull(int  r)
{
If (r==0)
Return(true)
Else
Return(false)
}
```

However, if we want to check this for float numbers, we should use another comparison due to the precision problem:

```
Bool  isNull(float  k)
{
If (k<=0.01)&&(k>-0.99)
Return(true)
Else
Return(false)
}
```

In both cases we want the function to have the name *isNull*. In programming languages without polymorphism for functions we cannot declare these two functions because the name *isNull* would be doubly defined. Without polymorphism for functions, doubly defined names would be ambiguous. However, if the language would take the *parameters* of the function into account it would work. Thus, functions (or methods) are uniquely identified by:

- The *name* of the function (or method) and
- The *types* of its *parameter list*.

Since the parameter list of both *isNull* functions differs, the compiler is able to figure-out the correct function call by using the actual types of the arguments.

```
Int  r;
Float   k;
r=0;
k=0.0;
…..
If ( isNull(r))   //use isNull integer
…..
If ( isNull(k))   //use isNull float
```

This type of polymorphism allows us to reuse the same name for functions (or methods) as long as the parameter list differs. Sometimes this type of polymorphism is called ***overloading***.

C. Polymorphism of Objects:

The last type of polymorphism allows an object to choose correct methods. In this type, polymorphism refers to situation in which objects belong to different classes can be respond to the same message, usually in different ways. For example, suppose we have classes box, triangle, and circle, whose objects represent the corresponding geometrical figures, as shown in figure (4). The objects of these classes might all understand a message draw( ), which causes an object to draw the corresponding figure on the screen.

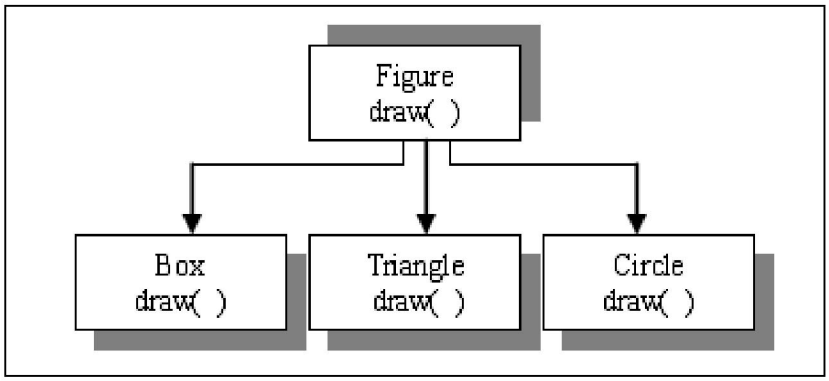


Figure 4: The class hierarchy for the figures example.

An essential feature of polymorphism is that we are able to send messages without knowing the class of the recipient object. For example, we might have a list of objects representing the figures that are to appear on the screen. To display the figures, we can send a draw( ) message to every object on the list, without having to worry about which objects represent boxes, which represent circles, and so on.

A list containing objects from different classes is called a heterogeneous list. Polymorphism greatly simplifies manipulating the objects in a heterogeneous list.