11.3 Overloading CONSTRUCTOR

It is possible ***overload*** a class's *constructor function* but not possible to overload a ***destructor***. There are ***three main reasons*** to overload constructor: the first two of these are discussed in this section.

[1] to gain flexibility, ***[2] to support arrays*** ***[3] to create copy constructors.***

- If a program attempts to create an object for which ***no matching constructor*** (means : with parameter or no parameter) is found, a ***compile-time error*** occurs. This is why ***overloaded constructor functions*** are so common to C++ programs.

- ***Giving an object an initialization or not:*** . The most frequent use of overloaded constructor functions is to provide the option of either giving an object an initialization or not giving it one. By providing both a ***parameterized*** and a ***parameterless*** constructor, your program allows the creation of objects that ***are either initialized or not as needed.*** ***For example***, in the following program, ***o1*** is given an initial value, but ***o2*** is not. Either removing the *constructor-with-no-parameter* or *constructor-with-parameter* will cause ***compile-time error*** because there is no match for an *initialized* object or *non-initialized* object.

```
class myclass { int x;
      public: /* overload constructor two ways */
    myclass() {x=0;}        /* no initializer */
    myclass(int n) {x=n;}  /* initializer */
    int getx() { return x; }
};
```

```
int main (){
myclass o(10) ;      /* declare with initial value */
myclass o2;          /* declare without initializer */
      cout << "o1: " << o1.getx() << '\n';
      cout << "o2: " << o2.getx() << '\n';
return 0;}
```

- ***Allowing both individual objects and arrays of objects:*** Overloaded constructor functions *allow both individual objects and arrays of objects* to occur within a program. It is fairly common to initialize a single variable, but it is *not as common to initialize an array.* So we must include a constructor that supports initialization and one that does not. From previous example, both of these declarations are valid:

```
myclass ob(10) ;
myclass ob[5];
```

We can use both ***parameterized*** and ***parameterless*** constructors to create *initialized and non-initialized arrays*. For example, the following program with previous example's class, declares two arrays of type ***myclass***; one is initialized and the other is not:

```
int main(){
myclass o1[10]; /* array without initializers */
      /* array with initializer */
myclass o2[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
int i; /* Output both array */
for(i=0; i <10; i++){
      cout <<"o1["<<i <<"]: "<< o1[i].getx()<<'\n';
      cout <<"o2["<<i <<"]: "<< o2[i].getx()<<'\n';}
return 0;}
```

Here, all elements of ***o1*** are set to ***0*** by the constructor function. The elements of ***o2*** are initialized as shown in the program.

- Overloading constructor functions is to *allow us to select the most convenient method of initializing an object*. For example following overloads the ***date()*** constructor two ways. One as a *character string*. Another *passed as three integers*.

```
#include<cstdio> /* included for sscanf()*/
class date{int day, month, year ;
  public : date( char *str );
           date( int m, int d, int y){
             day = d; month = m; year = y;}
            void show(){
   cout <<month<<'/'<<day<<'/'<<year<<'\n';}
};
```

```
date :: date( char *str ){
sscanf(str, "%d%*c%d%*c%d", &month, &day, &year);}
int main(){
   date sdate("12/31/99");        /* using string */
   date idate (12, 31, 99) ;      /* using integers */
      sdate.show();
      idate.show();
return 0;}
```

- ***Initializing dynamically allocated array:*** Overloading a class's constructor function allows us to initialize *dynamically allocated array*. We know that a dynamic array cannot be initialized. Thus, if the class contains a constructor that takes an *initializer*, you must include an overloaded version that takes *no-initializer*. For example, here is a program that allocates an object array dynamically:

```
class myclass{ int x;
      public :
myclass() {x=0;}      /* no initializer */
myclass(int n) {x=n;}  /* initializer */
int getx() {return x;}
};
```

```
int main(){  myclass *p;
       myclass ob(10) ;          /* initialize single variable */
       p = new myclass[10];      /* can 't use initializers here */
if(!p){cout << "Allocation error \n"; return 1;}
int i;
for(i=0; i <10; i++) p[i]=ob; /* initialize all elements to ob */
for (i=0; i<10; i++){
            cout <<"p["<<i<<"]: "<<p[i].getx()<<'\n';}
return 0;}
```

Without overloaded ***myclass()*** that has ***no-initializer***, "***new***" would've generated a *compile-time error*.

NOTE

[1] The C library function ***sscanf()*** of "***stdio.h***" reads *formatted input* from a ***string***.

Declaration: `int sscanf(const char *str, const char *format, ...)`

Parameters : *str* - This is the ***C string*** that the function processes as its source to retrieve the data.
format - This is the ***C string*** that contains one or more of the items: ***Whitespace*** character, ***Non-whitespace*** character and ***Format specifiers***.

[2] It is possible to overload a constructor as many times as you want but,

- ✓ Doing so excessively has a destructing effect on the class.
- ✓ It is best to overload a constructor to accommodate only those situations that are likely to occur frequently.

11.4 COPY CONSTRUCTOR (recall 10.10)

- ***Problem while passing objects to a function:*** When an object is passed to a function, a ***bitwise (i.e., exact) copy of that object is made*** and given to the ***function parameter that receives the object***. However, there are cases in which this identical copy is not desirable. For example, if the object contains ***a pointer to allocated memory***, the ***copy will point to the same memory as does the original object***. Therefore, if the copy makes a change to the contents of this memory, it will be ***changed*** for the ***original object*** too! Also, copy will be destroyed when the function terminates, the, causing its destructor to be called. This ***affect*** the ***original object***.

- ***Problem while returning object from a function:*** When an object is returned by a function. The compiler will commonly generate a ***temporary object that holds copy of the value returned by the function***. This temporary object goes ***out of scope*** once the value is ***returned*** to the calling routine, causing the temporary object's ***destructor*** to be called. ***If the destructor destroys something needed by the calling routine*** (for example, if it frees dynamically allocated memory), trouble will follow.

  ▶ At the core of these problems is the fact that a ***bitwise copy*** of the object is being made. By defining a copy constructor, you can fully specify exactly what occurs when a copy of an object is made.

❒ C++ defines two distinct types of situations in which the value of one object is given to another.

[1] The first situation is ***assignment***.

[2] The second situation is ***initialization*** which can occur three ways:

  [a] when an object is used to ***initialize another object*** in a declaration statement.

  [b] when an object is passed as a ***parameter to a function***, and

  [c] when a temporary object is created for use as a ***return value of a function***.

☞ The ***copy constructor only applies to initializations***. It does not apply to assignments.

❒ ***Declaring, defining & invoking (activating) copy-constructor:*** By default, when an initialization occurs, the compiler will automatically provides a default copy constructor that simply duplicates the object. However, it is possible to specify precisely how one object will initialize another by defining a copy constructor. Once defined, the copy constructor is called whenever an object is used to initialize another. The most common form of ***copy constructor*** is

```
class_name( const class_name &obj) {    /* body of constructor */  }
```

Here ***obj*** is a ***reference*** to an object that is being used to ***initialize another object***. ***const*** is an access modifier ***recall 5.7***.

☞ ***To declare & define :*** For example, a class called myclass, and that y is an object of type myclass, the ***declaration*** is

```
myclass( const myclass &ob );     /* const is an access modifier recall 5.7 */
```

The ***definition*** will be, ***myclass :: myclass(const myclass &ob) { /* body of copy-constructor*/ }***

☞ ***To invoke:*** Three types of statements can invoke the ***myclass copy constructor***:

```
[1] myclass x = y;      /* y explicitly initializing x */
[2] fun1 (y);           /* y passed as a parameter */
[3] y = func2 ();       /* y receiving a returned object */
```

In the first two cases, a ***reference*** to ***y*** would be passed to the ***copy constructor***. In the third, a ***reference*** to the object ***returned*** by ***func()*** is passed to the copy constructor.

☞ ***Example:*** This program creates a ***" safe " array class*** . Since space for the array is ***dynamically allocated*** , a ***copy constructor*** is provided to allocate memory when *one array object is used to initialize another* .

```
#include<cstdlib>                  /* for using exit() */
      class array {int *p, size ;
                   public: array (int sz);                  /* constructor */
                           array(const array &a);           /* copy constructor */
                           ~ array() { delete [] p; }       /* destructor */
                           void put(int i, int j) { if(i >=0 && i< size ) p[i] = j; }
                           int get(int i) { return p[i]; }  };
```

Copy constructor	***Normal constructor***

`array :: array( const array &a) { int i;`
`  size = a.size ;`
`  p = new int [a.size]; /* allocate memory for copy */`
`  if(!p){cout<< " Allocation error "; exit(1);}`
`  for(i=0; i<a.size; i++) p[i]=a.p[i]; /*copy*/`
`  cout << " Using copy constructor \n"; }`
	`array :: array( int sz) {`
	`  p = new int [sz ];`
	`  if(!p){cout<< " Allocation error "; exit(1);}`
	`  size = sz;`
	`  cout << "Using 'normal' constructor \n";`
	`  }`

`int main(){array num(10);  /* calls "normal" cnstrct */`
`  int i;`
`for(i=0; i<10; i++) num.put(i,i); /* array value */`
`for(i=9; i>=0; i--) cout<<num.get(i); // display`
`cout << "\n";`
	/* create another array and initialize with num */
	`array x = num; /* this invokes copy constructor */`
	`for(i=0; i<10; i++) cout<< x.get(i); // display x`
	`return 0; }`

✓ Here in the ***copy constructor***, memory is allocated specifically for the ***copy*** , and the ***address of this memory*** is assigned to ***p***. Therefore , ***p*** is ***not pointing*** to the same dynamically allocated memory as the ***original object*** .

✓ ***When copy-constructor is called:*** When ***num*** is used to initialize ***x*** (i.e., ***array x = num;***) the copy constructor is called, memory for the new array is allocated and stored in ***x.p***, and the contents of num are copied to ***x***'s array. In this way, ***x*** and ***num*** have arrays that have the same values, but ***each array is separate and distinct***. (That is, ***num.p*** and ***x.p*** do ***not point*** to the ***same*** piece of ***memory***.)

✓ If the copy constructor had not been created, the ***bitwise initialization array x = num;*** would have resulted in ***x*** and ***num*** ***sharing*** the ***same memory*** for their arrays! (That is, ***num.p*** and ***x.p*** would have ***pointed to the same location***.)

✓ ***When copy-constructor is not called:*** The copy constructor is called only for initializations. Copy constructors do not affect assignment operations. For example, the following sequence does not call the copy constructor defined in this program:
array a(10) ; array b(10) ; b = a; (***b = a*** performs the assignment operation. Rather calling copy-constructor)

NOTE: Old Overload keyword : For old C++ compiler the keyword overload was required to create an overloaded function. It is now obsolete and no longer supported by modern C++ compilers. The general form:

```
overload func_name ;        (It must precede the overloaded function declarations)
```

where ***func_name*** is the name of the function to be overloaded..

11.5 Default arguments

When a function (having one or more parameter) is *called without specifying corresponding arguments* the ***default arguments*** allows us to give default value to a parameter/s. Using default arguments is essentially a *shorthand form of function overloading*.

❒ To give a parameter a default argument, simply *follow that parameter with an equal sign and the value* you want it to default to if no corresponding argument is present. ***Example:*** Following function gives its two parameters default values of ***0***:

```
void f(int a=0, int b=0) ;
```

☞ Notice that this syntax is similar to variable initialization. This function can now be called three different ways.

[1] First, it can be called with ***both arguments*** specified. Example: `f(10, 99);  /* a is 10, b is 99*/`

[2] Second, it can be called with only the ***first argument*** specified. In this case, b will default to 0. Example:

```
f(10);        /* a is 10, b defaults to 0 */
```

[3] Finally, ***f()*** can be called with ***no arguments***, causing both ***a*** and ***b*** to default to ***0***. Example:

```
f();          /* a and b default to 0*/
```

☛ It is clear that there is no way to ***default a*** and ***specify b***.

❒ There are several rules to specify *default arguments* :

☞ Default arguments must be specified ***only once***: either in the function's ***prototype*** or in its ***definition*** if the *definition precedes the function's first use*. The defaults cannot be specified in both the *prototype* and the *definition*.

☞ All default parameters must be to the ***right side*** of any parameters that don't have defaults. Once you begin to define default parameters, you cannot specify any parameters that have no defaults (i.e. specified parameters can't stay ***left side*** of the default parameters).

☞ Default arguments must be constants or global variables. They cannot be local variables or other parameters.
Here is a program that illustrates the example described in the preceding discussion:

```
void f(int a=0, int b=0){
cout <<"a: "<< a <<", b: "<< b;
cout <<'\n';}

int main(){
    f();
    f(10) ;
    f(10 , 99);
    return 0;}
```

output:

```
a: 0, b: 0
a: 10, b: 0
a: 10, b: 99
```

☞ All arguments must specified as default: We can't make any specific argument to default. *Once the first default argument is specified, all following parameters must have defaults as well*. For example, this slightly different version of ***f()*** causes a ***compile-time error***:

```
void f(int a=0, int b)          /* wrong! b must have default , too */
```

Many times a constructor is overloaded simply to allow both initialized and uninitialized objects to be created. In many cases, you can *avoid overloading a constructor by giving it one or more default arguments*. For example,

```
myclass(int n = 0) { x = n; }     /* default argument instead of overloading constructor */
      . . . . . .
      . . . . . .
myclass o1(10) ;                  /* declare with initial value */
myclass o2;                       /* declare without initializer */
```

So it is possible to create objects that have explicit initial values and those for which the default value is sufficient.

☞ ***Cpy-Constructor with default arguments:*** We've seen the general form of a ***copy constructor*** with only one parameter. However, it is possible to create copy constructors that take additional arguments, as long as the additional arguments have ***default values***. For example,

```
myclass( const myclass &obj , int x = 0){ /* body of constructor */ }
```

As long as *the first argument is a reference to the object being copied*, and *all other arguments default*, the function qualifies as a ***copy constructor***. This flexibility allows you to create copy constructors that have other uses.

NOTE

[1] Another good application for a default argument is found when a parameter is used to select an option.

[2] Although default arguments are powerful and convenient, they can be misused. For example,

✓ If the argument is the value wanted nine times out of ten, giving a function a default argument to this effect is obviously a good idea.

✓ In cases in which no one value is more likely to be used than another, or when there is no benefit to using a default argument as a flag value, it makes little sense to provide a default value.

11.6 Ambiguity Caused By Overloading

Overloading-caused ambiguity can be introduced through ***type conversions***, ***reference parameters***, and ***default arguments***. Further, some types of ambiguity are caused by the overloaded functions themselves. Other types occur in the manner in which an overloaded function is called.

[1] ***Type conversion ambiguity:*** when a function is called with an argument that is of a *compatible (but not the same) type* as the parameter to which it is being passed, the type of the argument is automatically converted to the target type (C++'s automatic type conversion rules). Sometimes ***automatic type conversion*** will cause an ambiguous situation when a function is overloaded.

☞ It is this sort of type conversion that allows a function such as ***putchar()*** to be called with a character even though its argument is specified as an ***int***.

```
float f( float i) { return i/2.0; }
double f( double i) { return i/3.0; }
int mai() {   float x = 10.09;
              double y = 10.09;

cout << f(x); /* unambiguous - use f(float) */
cout << f(y); /* unambiguous - use f(double) */
cout << f (10) ; /* ambiguous , convert I0 to double or
                                              float ?? */
return 0; }
```

The compiler is able to select the correct version of ***f()*** when it is called with either a ***float*** or a ***double*** variable. However, what happens when it is called with an ***integer***? Does the compiler call ***f(float)*** or ***f(double)***? (Both are valid conversions!) In either case, it is valid to ***promote an integer*** into ***either a float or a double***. Thus, the ambiguous situation is created.

☞ However, when this function is called with the ***wrong type of argument***, C++'s automatic conversion rules cause an ambiguous situation,

```
void f( unsigned char c) { cout << c; }
void f( char c) { cout << c; }

int main() { f('c');
             f(86) ;   /* which f() is called ? */
             return 0; }
```

Here, when ***f()*** is called with the numeric constant ***86***, the compiler cannot know whether to call ***f(unsigned char)*** or ***f(char)***. Either conversion is equally valid, thus leading to ambiguity.

[2] ***Ambiguity by Reference:*** In C++ there is no syntactical difference between calling a function that ***takes a value parameter*** and calling a function that ***takes a reference parameter***, hence ambiguity arise. For example :

```
int f(int a, int b) { return a+b; }
              /* Following is inherently ambiguous */
int f(int a, int &b) { return a-b; }

int main() { int x=1, y=2;
cout << f(x, y);     /* which version of f() is called?*/
return 0; }
```

Here, ***f(x, y)*** is ambiguous because it could be calling either version of the function.

[3] ***Ambiguity by default arguments:*** Another type of ambiguity is caused when you are overloading a function in which one or more overloaded functions use a ***default argument***. Consider following program:

```
int f(int a) { return a*a; }
int f(int a, int b = 0) { return a*b; }

int main() { cout << f(10 , 2); /* calls f(int , int) */
cout << f (10) ;     /* ambiguous: f(int ) or f(int , int )? */
return 0; }
```

Here the call ***f(10, 2)*** is perfectly acceptable, and unambiguous. However, the compiler has now way of knowing whether the call ***f(10)*** is calling the first version of ***f()*** or the second version with ***b defaulting***.

11.7 Address of an OVERLOADED function (recall 5.8)

In C, you can assign the address of a function (that is, its entry point) to a pointer and access that function via that pointer. A function's address is obtained by putting its name on the right side of an assignment statement without any parentheses or arguments. For example, if ***zap()*** is a function, then we assign ***p*** the address of ***zap()***: ***p = zap;***

❒ In C, any type of pointer can be used to point to a function because there is only one function that it can point to.

❒ In C++ the situation is a bit more complex because a ***function can be overloaded***. When we assign the address of an overloaded function to a function pointer, ***it is the declaration of the pointer that determines which function's address is assigned***. Further, ***the declaration of the function pointer must exactly match one and only one of the overloaded functions***. If it does not, ambiguity will be introduced, causing a compile-time error.

```
void space(int n){ /* Type-1: Output n number of spaces */
             for( ; n ; n --) cout << ' '; }
void space(int n , char ch){ /* Type-2: Output n chs */
             for( ; n ; n --) cout << ch; }
int main() {
   /* pointer to void function with one int parameter */
void ( *fp1 )(int );          /* matches to Type-1 */
   /* pointer to void function with int and character parameter. */
void ( *fp2 )(int, char);  /* matches to Type-2 */
```

```
fp1 = space ;        /* gets address of space(int)*/
fp2 = space ;        /* gets address of space(int , char )*/
fp1(22) ;            /* output 22 spaces */
cout << "|\n";
fp2 (30 , 'x');      /* output 30 x's */
cout << "|\n";
return 0;}
```

- ▶ Here two versions of a function called ***space()***. The first version outputs ***n number of spaces*** to the screen. The second version outputs ***n number of*** whatever type of ***character*** is passed to ch.
- ▶ In ***main()***, two function pointers are declared. The first one is specified as a pointer to a function having ***only one integer*** parameter. The second is declared as a pointer to a function taking two parameters : ***one integer and one character***.
- ▶ The compiler is able to determine which overloaded function to obtain the address of based upon how ***fp1*** and ***fp2*** are declared.

11.8 Overloading MEMBER OPERATOR FUNCTIONS

- O Operator overloading is really just a type of ***function overloading*** with some ***additional rules*** apply.
- O When an operator is overloaded, that operator ***loses none of its original meaning***. Instead, it gains additional meaning relative to the ***class*** for which it is defined.
  - ✓ ***For example***, an operator is always overloaded relative to a ***user-defined type***, such as a ***class***.
- ❒ To overload an operator, you create an ***operator function***. Most often an operator function is :
  1. ***Member*** operator function
  2. ***Friend*** operator function

- However, there is a slight difference between a ***member operator function*** and a ***friend operator function***.

- The ***general form*** of a member operator function is,

```
return_type class_name::operator#( arg_list ) {
                                        /* operation to be performed */   }
```

  - ✓ The ***return type*** of an operator function is often the ***class*** for which it is defined. (operator function may free to return any type.)
  - ✓ The operator being overloaded is substituted for the **#**. For example, to overloaded + , the function name would be ***operator+***.
  - ✓ The contents of ***arg-list*** vary depending upon how the operator function is ***implemented*** and the ***type of operator*** being overloaded.
  - ***Restrictions:***
    - The ***precedence of the operator*** cannot be changed.
    - Second, the ***number of operands*** that an operator takes cannot be altered. I.e. a ***binary operator cannot be overload as an unary operator***. For example, you cannot overload the **/** operator so that it takes only one operand.
    - These operators we cannot overload: **. :: .* ?**
    - We cannot overload the ***preprocessor operators*** (i.e. **#, ##**). *(The **.*** operator is highly specialized and not discussed in this book.)*
  - The **[ ]** ***subscript*** operators, the **( )** ***function call*** operators, **new** and **delete**, and the **.** (***dot***) and **->** (***arrow***) operators can be overloaded.
  - Except for the **=**, operator functions are ***inherited*** by any derived class. However, a derived class is ***free to overload any operator*** it chooses (including those overloaded by the base class) relative to itself.
  - You have been using two overloaded operators: **<<** and **>>**. These operators have been overloaded to perform ***console I/O in C++***. As mentioned, ***overloading these operators to perform I/O*** does not prevent them from performing their traditional jobs of ***left shift*** and ***right shift***.
  - However, ***do not use*** any operator overloading ***abnormally***.

11.9 Overloading Binary Operators

When a member operator function overloads a ***binary operator***, the function will have only ***one parameter***.

- The parameter will ***receive the object*** that is on the ***right*** side of the operator.
- The object on the ***left*** side is the object that ***generates the call*** to the operator function and is passed ***implicitly*** by ***this*** (pointer).

- Operator functions can be written in various way, some are shown in the examples:

- ***Example 1(*** *return type, temporary object, no-operator modification* ***):*** ***Overload the + operator*** relative to the ***coord*** class. This class is used to maintain ***X, Y coordinates***.

```
class coord {int x, y;                  /* coordinate */
      public :
            coord() { x=0; y=0; };
            coord(int i, int j) {x=i; y=j; }
            void get_xy(int &i, int &j) {
                                    i=x; j=y; }
            coord operator+(coord ob2); };
      /* coord type used for overloaded operator */
```

```
                  /* Overload + relative to coord class.*/
coord coord :: operator +( coord ob2) {
      /* one coord for type and another is for class*/
            coord temp ;
            temp.x = x + ob2.x;
            temp.y = y + ob2.y;
            return temp ;
      }
```

```
      int main() { coord o1(10 , 10), o2(5, 3), o3;
                   int x, y;
                   o3 = o1 + o2;        /* add two objects – "this" calls operator + */
                   o3.get_xy(x, y);
                   cout << "(o1+o2) X: " << x << ", Y: " << y << "\n";
                   return 0; }
```

Output: **(o1+o2) X: 15, Y: 13**

- Notice that there is no ***ob1*** in operator function (it is ***implicit*** here and used to call ***ob2***), however we'll use both ***ob1*** and ***ob2*** in the *friend operator function*.
- The reason that the ***operator+()*** function returns an object of type coord is: ***o3 = o1 + o2*** is valid ***iff*** the result of ***o1 + o2*** is a ***coord object*** that can be assigned to ***o3***. If a *different type* had been *returned*, this statement would have been *invalid*.
- Notice that a *temporary object* called ***temp*** is used inside ***operator+()*** to hold the result, and it is the ***returned*** object.
  - ⇨ *The reason for **temp** is:* a ***temporary object*** is needed to hold the result. In this situation (as in most), the **+** has been overloaded in a manner consistent with its normal arithmetic use. Therefore, neither operand be changed. For example, for ***10+4=14***, the result is ***14***, but neither the ***10*** nor the ***4*** is modified.
  - ⇨ Because a ***coord*** object is ***returned***, the statement: ***(o1+o2). get_xy (x, y);*** is also perfectly valid. Here the *temporary object returned* by ***operator+()*** is used ***directly*** and after execution the temporary object is ***destroyed***.

- ***Example 2(*** *order of objects, implicit first object, assignment operator* ***):*** ***Overload the +, - and "=" operator*** relative to the ***coord*** class. This class is used to maintain ***X, Y coordinates*** similar to previous example.

```
      class coord {       /* all elements of public are same, only declare operator functions*/
            public:       . . . . . /* all elements similar to Example 1 */ . . . . . .
                  coord operator +( coord ob2);
                  coord operator -( coord ob2);
                  coord operator =( coord ob2); };
```

```
coord coord : : operator +( coord ob2) {
coord temp ;
      temp.x = x + ob2.x;
      temp.y = y + ob2.y;
      return temp ; }
```

```
coord coord : : operator -( coord ob2) {
coord temp ;
      temp.x = x - ob2.x;
      temp.y = y - ob2.y;
      return temp ; }
```

```
coord coord : : operator =( coord ob2) {
      x = ob2.x;
      y = ob2.y;
      return *this ;
/* return the object that is assigned */   }
```

```
int main() { coord o1(10 , 10) , o2(5, 3) , o3;
             int x, y;
o3 = o1+o2;  o3.get_xy(x, y); cout<< "(o1+o2) X: "<< x <<", Y: "<< y <<"\n";   /* add two objects */
o3 = o1-o2;  o3.get_xy(x, y); cout<< "(o1-o2) X: "<< x <<", Y: "<< y <<"\n";   /* subtract two objects */
o3 = o1;     o3.get_xy(x, y); cout<< "(o3=o1) X: "<< x <<", Y: "<< y <<"\n";   /* assign an object */
return 0; }
```

- ***Order of the operands:*** The ***operator-()*** function is implemented similarly to ***operator+()***. However the *order of the operands* is important while overloading an operator.
  - The order of the ***left-operand*** which "*generates the call* to ***operator-()***" and the ***right-operand*** which "*passed as an argument* to the ***operator-()***" is important for subtraction because $A - B \neq B - A$, It must be in the order: ***x - ob2.x;*** .
  - The order of the *left-operand* and *right-operand* is also important when we use ***built-in-type*** variables as *right-operand* .
  - *The order* of the *left-operand and the right*-operand is *not important for addition*.
- ***The assignment operator function:*** Here the ***left-operand*** is ***modified*** by the operation (that is, the object being assigned a value). This is in keeping with the normal meaning of assignment.
  - The function ***returns *this***. That is, the ***operator=()*** function *returns the object that is being assigned to*. The reason for this is to allow a series of assignments to be made. Eg: we used ***a = b = c = d = 0;*** for ***variables***, returning ****this*** by overloaded assignment operator allows us to use ***o3 = o2 = o1;*** for ***multiple objects***.
  - There is no rule that requires an overloaded assignment function *to return the object that receives the assignment*. However, if you want the overloaded **=** to behave relative to its class the *way it does for the built-in types*, it must return ****this***.

- ***Example 3(*** *built-in-type objects i.e. int-float-char, order of the operands* ***):*** *Overload the + operator* relative to the ***coord*** class with built-in-type objects (i.e int, float, char etc). This class is used to maintain ***X, Y coordinates*** similar to previous example.

```
class coord {
      public:
/* all elements similar to Example 1 */
coord operator+(coord ob2);      //obj+obj
coord operator+(int i); };       //obj+int
```

```
coord coord : : operator +( coord ob2) {
coord temp ;
      temp.x = x + ob2.x;
      temp.y = y + ob2.y;
      return temp ; }
```

```
coord coord : : operator +( int i) {
coord temp ;
      temp.x = x + i;
      temp.y = y + i;
      return temp ; }
```

```
int main() { coord o1(10 , 10) , o2(5, 3) , o3;
             int x, y;
o3 = o1+o2;  o3.get_xy(x, y); cout<< "(o1+o2) X: "<< x <<", Y: "<< y <<"\n";     /* add two objects */
o3 = o1+100; o3.get_xy(x, y); cout<< "(o1+100) X: "<< x <<", Y: "<< y <<"\n";    /* add object + int */
return 0; }
```

- The order of the *left-operand* and *right-operand* is important when we use ***built-in-type*** variables as *right-operand* . The reason is: It is the *object on the left that generates the call* to the operator function. For instance, `o3 = 19 + o1; /* int + obj */` generates a **compile-time error**. Because there is no built-in operation defined to handle the addition of an integer to an object.
- The overloaded ***operator+(int i)*** function works only when the object is on the ***left***. (However there is a solution around this restriction.)

- ***Example 4(*** *reference parameter in operator funtion* ***):*** *Overload the + operator* relative to the ***coord*** class using reference. This class is used to maintain ***X, Y coordinates*** similar to previous example.

```
coord coord :: operator+( coord &ob2) { coord temp ; /* using references.*/
                                        temp.x = x + ob2.x;
                                        temp.y = y + ob2.y;
                                        return temp ; }
```

- ***Efficiency:*** Passing the ***address*** of an object is always *quick and efficient* and do not consume CPU cycles as much as normal object parameters do. If the operator is going to be used often, using a reference parameter is a good choice.
- ***Prevent temporary object/operand destruction after execution (recall 10.10):*** when an argument is passed by value, a copy of that argument is made. If that object has a *destructor function*, when the function terminates, the copy's destructor is called.
  - Using a ***reference parameter*** instead of a ***value parameter*** is an easy (and efficient) way around the problem.
  - However, we could also define a ***copy constructor*** that would prevent this problem in the general case.

NOTE:
When a ***binary operator*** is overloaded, the *left operand is passed implicitly* to the function and the *right operand* is passed as an *argument*.

11.10 Overloading the RELATIONAL and LOGICAL operators

Overloading the relational and logical operators so that they behave in their traditional manner, they will return an integer that indicates either ***true*** or ***false***.

- It allows the operators to be integrated into larger relational and logical expressions that involve other types of data.

Example 1. In the following program, the == and && operators are overloaded: comparing two objects- same/true/false/different.

```
class coord { public: /* similar to 11.9 Ex 1 */     int operator==(coord ob2); int operator&&(coord ob2); };
```

```
int coord::operator==(coord ob2)
{
return (x==ob2.x)&&(y==ob2.y);
}
```

```
int coord::operator&&(coord ob2)
{
return (x&&ob2.x)&&(y&&ob2.y);
}
```

```
int main(){coord o1(10,10), o2(5,3);
     if(o1 == o2) cout<<" same \n";
          else cout<<" differs \n";
     if(o1&&o2) cout<<" true \n";
          else cout << " false \n";
return 0;}
```

- Here both objects corresponding member elements are compared and then gives ***true*** or ***false*** value.
- Notice that in the declaration of both ***operator==()*** and ***operator&&()*** returns ***int***. This is because ***true*** and ***false*** are corresponds to the values ***1*** and ***0***.

11.11 Overloading A UNARY Operator

Overloading a unary operator is similar to overloading a binary operator except that there is only one operand to deal with.

- When you overload a unary operator using a ***member function***, the function has ***no parameters***.
- Since there is only one operand, it is this operand that generates the call to the operator function.
- ***Example 1:*** The following program overloads the ***increment*** operator (**++**) relative to the ***coord*** class

```
class coord {
      public:
/* all elements similar to Example 1 */
coord operator++();
};
```

```
crood coord::operator++(){
                    x++;
                    y++;
          return *this ;}
```

```
int main(){coord o1(10, 10);  int x, y;
          ++o1; /* increment an object */
          o1.get_xy(x, y);
          cout<<"(++o1) X:"<<x<<", Y:"<<y;
          return 0;}
```

- ***++*** is designed to *increase* its *operand* by ***1***, the overloaded ***++*** modifies the object it operates upon.
- The function returns the object that it *increments* allowing ***++*** to be used as part of a larger statement, such as: ***o2 = ++o1;***
- There is *no rule* that says we must overload a unary operator so that it reflects its *normal meaning*.

- In early version of ***C++*** there was no way to determine whether an overloaded **++** or **--** ***preceded*** or ***followed*** its operand. That is these two statements would have been identical: ***o1++;*** and ***++o1;***
  - In modern C++ to distinguish between these two statements we declare following by which the compiler can distinguish.

    coord coord :: operator++(int notused);
    - If the ***++*** operator ***precedes*** its operand, the ***operator++()*** function is called.
    - If the ***++ follows*** its operand, the ***operator++(int notused)*** function is used. In this case, ***notused*** will always be passed the value ***0***.
  - So, if the difference between ***prefix*** and ***postfix*** increment or decrement is important to your class objects, you will need to implement both operator functions.

    coord coord :: operator++();
    coord coord :: operator++(int notused);

- ***Example 2*** (Use – as "unary" and "binary" both in a program)***:*** *The solution is:* you simply overload it ***twice***, once as a ***binary*** operator and once as a ***unary*** operator. This program shows how:

```
class coord {
      public:
/* all elements similar to Example 1 */
coord operator-(coord ob2);  //binary
coord operator-(); };        //unary
```

```
coord coord::operator-(){
                x = -x;
                y = -y;
          return *this ;}
```

```
coord coord::operator-( coord ob2){
                coord temp ;
                temp.x = x - ob2.x;
                temp.y = y - ob2.y;
                return temp ; }
```

```
int main() { coord o1(10, 10), o2(5, 7);  int x, y;
      o1=o1-o2;     o1.get_xy(x, y);  cout<< "(o1 -o2) X:"<< x <<", Y:"<< y << "\n"; /* subtraction */
      o1=-o1;       o1.get_xy(x, y);  cout<< "(-o1) X:" << x << ", Y: " << y << "\n"; /* negation */
      return 0; }
```

- This *difference in the number of parameters* is what makes it possible for the minus to be *overloaded* for *both* operations.
  - When the minus is overloaded as a ***binary*** operator, it takes ***one parameter***. When the minus sign is used as a ***binary*** operator, the ***operator-(coord ob2)*** function is called.
  - When it is overloaded as a ***unary*** operator, it takes ***no parameter***. When it is used as a ***unary*** minus, the ***operator-()*** function is called.

11.12 Overloading FRIEND OPERATOR FUNCTIONS

It is possible to *overload an operator* relative to a class by using a ***friend*** rather than a ***member*** function. Since a *friend function* does not have a ***this*** pointer:

- In the case of a ***binary operator***, this means that a friend operator function is ***passed*** *both operands* ***explicitly***.
- For ***unary operators***, the single operand is ***passed***.
- You ***cannot*** use a ***friend*** to overload the ***assignment operator***. The *assignment operator can be overloaded only by a member operator function*.

```
Example 1:   class coord { int x, y;          /* coordinate values */
                   public :     coord() { x=0; y=0; };
                                coord(int i, int j) { x=i; y=j; }
                                void get_xy(int &i, int &j) { i=x; j=y; }
                                friend coord operator+(coord ob1, coord ob2); };
```

```
coord operator+(coord ob1, coord ob2) {
                coord temp ;
                temp.x = ob1.x + ob2.x;
                temp.y = ob1.y + ob2.y;
                return temp ; }
```

```
int main(){  coord o1(10, 10), o2 (5, 3), o3;
             int x, y;
             o3 = o1 + o2;  o3.get_xy(x, y);
             cout<< "(o1+o2) X:"<< x <<", Y:"<< y << "\n";
             return 0;}
```

- ✓ Notice that there are ***two parameters*** present in both ***declaration*** and ***definition*** of the ***friend operator function***.
- ✓ Also the ***left operand*** is passed to the ***first parameter*** and the ***right operand*** is passed to the ***second parameter***.

❒ ***Example 2*** (No-order for objects)***:*** For an ***overloaded member operator*** function ***ob1=10+ob2;*** is ***illegal***. And the member operator function works only when the ***built-in*** object type is on the ***left***.

☞ A ***friend operator*** function allows objects to be used in operations involving built-in types in which the ***built-in type is on the left*** side of the ***operator***.

☞ To do this we have to make the overloaded operator functions friends and ***define both possible situations***. We can define one overloaded friend function so that the ***left operand*** is an ***object*** and the ***right operand*** is the ***other type***. Then we could overload the operator again with the ***left operand*** being the ***built-in*** type and the ***right operand*** being the ***object***. The following program illustrates this method:

```
class coord {
      public:
/* all elements similar to Example 1 */
friend coord operator+(coord ob1, int i);
friend coord operator+(int i, coord ob1);
};
```

```
/* Overload + for ob + int */
coord operator+(coord ob1, int i)
    {  coord temp ;
       temp.x = ob1.x + i;
       temp.y = ob1.y + i;
       return temp ;}
```

```
/* Overload + for int + ob */
coord operator+( int i, coord ob1)
    {  coord temp ;
       temp.x = ob1.x + i;
       temp.y = ob1.y + i;
       return temp ;   }
```

```
int main() { coord o1(10, 10); int x, y;
  o1=o1+10;  o1.get_xy(x, y);    cout<< "(o1 +10) X:"<< x <<", Y:"<< y <<"\n"; /* object + integer */
  o1=99+o1;  o1.get_xy(x, y);    cout<< "(99+ o1) X:"<< x <<", Y:"<< y <<"\n"; /* integer + object */
return 0; }
```

❒ ***Example 3*** (unary friend operator with prefix, postfix and reference parameter)***:*** to overload either the **++** or **--** ***unary*** operator, you ***must pass the operand to the function*** as a ***reference parameter*** (otherwise any modification inside the friend will not affect the object that generates the call). This is because friend functions do not have ***"this"*** **pointers**.

☞ If you pass the operand to the friend as a reference parameter, changes that occur inside the friend function affect the object that generates the call (***"this"*** connects the friend and object through ***reference***). For example, here is a program that overloads the ++ operator by using a ***friend function***:

```
class coord {
      public:
/* all elements similar to Example 1 */
friend coord operator++(coord &ob1);
};
```

```
/* Using reference */
coord operator++( coord &ob1) {
ob.x++;
ob.y++;
return ob;      // return generates the call
}
```

```
int main() {
coord o1(10, 10); int x, y;
++o1;    /* o1 is passed by reference */
o1.get_xy(x, y);
cout<<"(++ o1) X:"<< x
cout<<", Y:"<< y << "\n";
return 0; }
```

❒ To distinguish between the ***prefix*** and the ***postfix*** forms of the ***increment*** or ***decrement*** operators when using a ***friend operator function***: simply ***add an integer parameter*** when defining the ***postfix version*** (similar to member operator). For example, consider the previous ***coord*** class, the ***prototypes*** for both will be:

```
coord operator++( coord &ob);                  // prefix
coord operator++( coord &ob, int notused );    // postfix
```

☞ If the ***++ precedes*** its operand, the ***operator++(coord &ob)*** function is called.

☞ If the ***++ follows*** its operand, the ***operator++(coord &ob, int notused)*** function is used. In this case, ***notused*** will be passed the value ***0***.

11.13 Assignment Operator Advanced

By default (without overloading), when the assignment operator is applied to an object, ***a bitwise copy of the object on the right is put into the object on the left***. If this is what you want, there is no reason to provide your own ***operator=()*** function (i.e. overloading has no reason).

▶ However, there are cases in which a strict bitwise copy is not desirable and we need to provide a special assignment operation.

Example: Following program overloads the = operator so that the pointer p is not overwritten by an assignment operation.

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

```
class strtype { char *p; int len;
public :
strtype( char *s);
~ strtype(){cout<< "Freeing"<<(unsigned)p<<'\n'; delete [] p; }
char *get() { return p; }
strtype &operator=(strtype &ob); };      /* reference operator function */
```

```
strtype :: strtype( char *s) {
int l;
l = strlen(s)+1;
p = new char [l];
if(!p) {
cout << "Allocation error \n";
exit(1) ; }
len = l;      strcpy(p, s); }
```

```
/* Assign an object. */
strtype &strtype :: operator=(strtype &ob){
   /* need to allocate more memory */
if(len < ob.len ){ delete []p;
p = new char [ob.len ];
if(!p) {cout<<"Alloc. error \n"; exit(1);} }
len = ob.len ;
strcpy(p, ob.p);
return * this ; }
```

```
int main() {
strtype a(" Hello ");
strtype b(" There ");
cout << a.get () << '\n';
cout << b.get () << '\n';
a = b; // now p is not overwritten
cout<<a.get()<<b.get();
return 0; }
```

➢ The overloaded assignment operator prevents ***p*** from being overwritten.

☛ It first checks to see if the object on the left has allocated enough memory to hold the string that is being assigned to it. If it hasn't, that memory is freed and another portion is allocated.

☛ Then the string is copied to that memory and the length is copied into ***len***.

➢ Notice two other important features about the ***operator=()*** function.

✓ ***First***, it takes a ***reference parameter***. This prevents a copy of the object on the right side of the assignment from being made.

[Recall 10.10 and 10.12 : when a copy of an object is made when passed to a function, that copy is destroyed when the function terminates. In this case, destroying the copy would call the destructor function, which would free p. However, this is the same p still needed by the object used as an argument. Using a reference parameter prevents this problem.]

✓ The ***second*** important feature of the ***operator=()*** function is that it ***returns a reference***, <u>not an object</u>. The reason for this is the same as the reason it uses a reference parameter.

[Recall 10.10 and 10.12 :When a function returns an object, a temporary object is created that is destroyed after the return is complete. However, this means that the temporary object's destructor will be called, causing p to be freed, but p (and the memory it points to) is still needed by the object being assigned a value. Therefore, by returning a reference, you prevent a temporary object from being created.]

NOTE: We know creating a ***copy constructor*** is another way to prevent both of the problems described in the preceding two paragraphs. But the ***copy constructor*** might not be as efficient a solution as using a reference parameter and a ***return reference type***. This is because using a reference prevents the overhead associated with copying an object in either circumstances.

There are often several ways to accomplish the same end in C++. Learning to choose between them is part of becoming an excellent C++ programmer.

11.14 Overloading The [ ] SUBSCRIPT Operator

The general form of a member ***operator[ ]()*** function is:

```
type class_name :: operator[](int index) {  // ...  }
```

☛ Technically, the parameter does not have to be of type ***int***, but ***operator[ ]()*** functions are typically used to provide array subscripting and as such an ***integer*** value is ***generally*** used.

☛ In C++, the **[ ]** is considered a ***binary operator*** for the purposes of overloading.

☛ The **[ ]** can be overloaded only by a ***member function***.

❒ To understand how the **[ ]** operator works, assume that an object called ***O*** is indexed as: **O[9];** This index will translate into the following call to the ***operator[]()*** function:

```
O.operator[](9)
```

➢ The value of the expression within the subscripting operator is passed to the ***operator[]()*** function in its explicit parameter.

➢ The ***"this"*** pointer will point to ***O***, the object that generated the call.

❒ ***<u>Example 1:</u>*** In the following program, ***arraytype*** declares an *array of five integers*.

[1] Its ***constructor*** function ***initializes*** each member of the array.

[2] The overloaded ***operator[]()*** function ***returns*** the value of the element specified by its parameter.

```
const int SIZE = 5;
class arraytype { int a[ SIZE ];
                  public :
        arraytype() { int i;
        for(i=0; i<SIZE; i++) a[i] = i; }
        int operator[](int i) {return a[i]; } };

int main() { arraytype ob;
             int i;
             for(i=0; i< SIZE; i++)
             cout << ob[i] << " ";
        return 0; }
```

*[The initialization of the array **a** by the constructor in this and the following programs is for the sake of illustration only. It is not required.]*

❒ ***<u>Example 2 -Assigning values to & form using reference and []:</u>*** It is possible to design the ***operator[]()*** function in such a way that the **[ ]** can be used on ***both*** the ***left*** and ***right*** sides of an assignment statement *(i.e. **a=b[i]** and **b[i]=a**. Assigning values to & form array).* To do this, ***return*** a ***reference*** to the element being indexed.

```
class arraytype
{. . . .
public:
              /* as same as Example 1 of 11.14 */
int &operator[](int i){return a[i]; }     //reference
};
```

```
int main(){ arraytype ob;  int i;
      for(i=0; i<SIZE; i++) cout<<ob[i]<<" ";
cout << "\n";
        /* add 10 to each element in the array */
for(i=0; i< SIZE ; i++){
             ob[i] = ob[i ]+10;}  //[] on left of =
      for(i=0; i<SIZE; i++) cout<<ob[i]<<" ";
return 0;}
```

⇨ Because the ***operator[]()*** function now ***returns a reference*** to the *array element* indexed by ***i***, it can be used on the ***left*** side of an ***assignment*** to modify an element of the *array* (just like normal arrays).

❒ ***<u>Example 3 (Safe array):</u>*** Recall that a safe array is an array that is encapsulated within a class that performs bounds checking. This approach prevents the array boundaries from being overrun.

☛ To create a safe array with overloaded **[ ]** operator, simply add bounds checking to the ***operator[]()*** function. The ***operator[]()*** must also return a reference to the element being indexed. Following program proves that it works by *<u>generating a boundary error</u>*.

```
{ public:. . .
              /* as same as Example 1 of 11.14 */
int &operator[](int i) };                 //reference
           /* only declaration given inside class */
```

```
          /* Bound checks inside the if statement */
int &arraytype :: operator[](int i){
if(i<0 || i>(SIZE-1)){
   cout<<i<< "is out of bounds.\n"; exit(1);}
return a[i]; }
```

```
int main() { arraytype ob;  int i;
             for(i=0; i<SIZE; i++) cout<< ob[i] << " "; /* this is OK */
             ob[SIZE +100] = 99;  /* generates a run - time error because SIZE +100 is out of range */
        return 0;}
```

☞ In this program, when the statement ***ob[SIZE +100] = 99;*** executes, the boundary error is intercepted by ***operator[]()*** and the program is terminated before any damage can be done.

✓ ***Remark 1:*** Because the overloading of the **[ ]** operator allows you to create safe arrays that look and act just like ***regular arrays***.

✓ ***Remark 2:*** Be careful. A safe array adds ***overhead*** that might ***not be acceptable in all situations***. However, in applications in which you want to be sure that a boundary error does not take place, a safe array will be worth the effort.

11.15 INHERITANCE: access control of base class

❒ When one class ***inherits*** another, it uses this ***general form***:

```
class derived_class_name : access base_class_name { // ... }
```

☞ Here ***access*** is one of three keywords: ***public***, ***private***, or ***protected***.

❒ The *access specifier* determines how elements of the *base class* are ***inherited*** by the *derived class*.

☛ When the *access specifer* for the inherited *base class* is ***public***, all *public members* of the base become *public members* of the *derived class*.

☛ If the *access specifier* is ***private***, all ***public members*** of the *base class* become ***private members*** of the *derived class*.

⇨ But these ***private*** members are still accessible by *member functions* of the *derived class*.

☛ In either case, any ***private members of the base remain private to it*** and are *inaccessible by the derived class*.

☛ Technically, ***access*** is ***optional***:

➢ If the ***specifier*** is *not present*, it is ***private*** by ***default*** if the *derived class* is a ***class***.

➢ If the *derived class* is a ***struct***, ***public*** is the ***default*** in the absence of an ***explicit access specifier***.

➢ However, we explicitly specify access for the sake of clarity.

❒ ***Example 1:*** Here because ***base*** is inherited as ***public***, the *public members* of base- ***setx()*** and ***showx()*** - become *public members* of ***derived*** and are, therefore, *accessible by any other part of the program.* Specifically, they are ***legally called*** within ***main()***.

```
class base { int x;
      public :
      void setx (int n) { x = n; }
      void showx() {cout<< x <<'\n';}     };
                                          /* Inherit as public .*/
class derived : public base { int y;
  public : void sety (int n) { y = n; }
           void showy() {cout<< y <<'\n';}  };
```

```
int main() {  derived ob;          /* derived type object */
              ob.setx(10) ;        /* access member of base class through the derived class's object */
              ob.sety(20) ;        /* access member of derived class */
              ob.showx();          /* access member of base class through the derived class's object */
              ob.showy();          /* access member of derived class */
      return 0; }
```

❒ ***Example 2:*** It is important to understand that just because a derived class inherits a base as public, it does not mean that the derived class has access to the base's private members. For example, in previous ***example 1***:

```
class derived : public base { int y;
                      public:   /* Error! : x is a private member of base and not available within derived . */
                         void show_sum() {cout<< x+y << '\n';}
```

☞ In this example, the derived class attempts to access ***x***, which is a private member of ***base***. This is an ***error*** because the ***private*** parts of a ***base*** class *remain private* to it no matter how it is inherited.

❒ ***Example 3*** (with private specifier, public member of base become private to derived)***:*** this time derived inherits base as private.

```
class base { all same as Example 1     };
                /* Inherit as private .*/
class derived : private base { same inside };
```

```
int main(){ derived ob;     /* derived type object */
       ob.setx(10) ; /* ERROR - now setx() private to derived class */
       ob.showx(); /* ERROR - now showx() private to derived class */
return 0; }
```

☛ Both ***showx()*** and ***setx()*** become ***private*** to ***derived*** and are not accessible outside of it. Relative to *objects of type derived*, they become ***private***.

☛ Keep in mind that ***showx()*** and ***setx()*** are still ***public*** *within base* no matter how they are inherited by some *derived class*. This means that an *object of type base* could access these functions *anywhere*. For example, given this fragment:

```
base base_ob ;
base_ob.setx(1) ;
```

Is ***legal*** because ***base_ob*** is of *type base* and the call to ***setx()*** is legal because ***setx()*** is ***public*** within ***base***.

❒ ***Example 4:*** Even though *public members of a base* class become *private members of a derived* class when inherited using the ***private specifier***, they are still accessible within the derived class. In this case, the functions ***setx()*** and ***showx()*** are accessed *inside the derived class*, which is perfectly legal because they are private members of that class.

```
class derived : private base {
int y;
public :
void setxy(int n, int m) { setx(n); y=m; }  /* setx is accessible from within derived */
void showxy() { showx(); cout<< y; }  };     /* show is accessible from within derived */
```

```
int main(){
derived ob; // derived type
    ob.setxy(10, 20);
    ob.showxy();
return 0; }
```

11.16Accessing PROTECTED members

There will be times when you want to keep a member of a ***base class private*** but still allow a ***derived class*** access to it. To accomplish this goal, ***C++*** includes the ***protected*** *access specifier*.

- ☞ *Protected members* of a ***base*** class are accessible to members of any class ***derived*** from that base.
- ☞ *Outside the base or derived* classes, protected members are ***not accessible***.

❒ The ***protected*** *access specifier* can *occur anywhere* in the *class declaration*, although typically it occurs ***after the (default) private*** members are declared and ***before the public*** members. The full *general form of a class declaration* is shown here:

```
class class_name {          /* private members */
                  protected:  /* protected members (optional)*/
                  public:     /* public members */
                  };
```

❒ **When a *protected member* of a base class is inherited:**

- ☛ as public by the derived class, it becomes a protected member of the derived class.
- ☛ as private by the derived class, it becomes a private member of the derived class.

❒ ***When a base inherited as protected:*** public and protected members of the base class become protected members of the derived class. (Of course, private members of the base class remain private to it and not acessable.)

❒ The ***protected*** access specifier can also be used with ***structures***.

❒ ***Example1:*** This program illustrates how public, private, and protected members of a class *can be accessed* also what occurs when ***protected*** members are **inherited *as public***:

```
class base { int t;  /* private member*/
protected :
   int a, b;    /* private but accessible by derived */
public :
   int s;
   void setab(int n, int m) { a=n; b=m; }
};
```

```
class derived : public base {          /* new class */
                    int c;          /* private in deived */
       public :
                    void setc(int n) {c = n;}
void showbc(){          /* this has access to a and b from base */
                    cout<< a <<' '<< b <<' '<< c; }
};
```

```
int main(){ base ob_bs;             /* base type object */
       derived ob_drv;              /* derived type object */
       ob_bs.t=5;                   /* error: t is private member of base*/
       ob_bs.a=3; ob_bs.b=4;        /* error: a,b is protected member of base but accessible inside derived*/
       ob_bs.s=9                    /*ok: s is public member of base*/
       ob_bs.setab (3, 4);          /*ok: setab() is public member of base*/
       ob_drv.setab(1, 2);    /*ok: setab() is also public to derived because of public specifier in class declaration of derived */
       ob_drv.setc(3) ;             /*ok: c is public member of derived*/
       ob_drv.a=3; ob_drv.b=4;      /* error: a,b is protected member of base but accessible inside derived*/
       ob_drv.showabc();    return 0;}          /* accessing a,b inside of derived class*/
```

- ☞ Because ***a*** and ***b*** are ***protected*** in ***base*** and inherited as ***public*** by ***derived***, they are *available for use by member functions of derived* as we used to ***showabc( )***.
- ☞ However, outside of these two classes, ***a*** and ***b*** are effectively ***private*** and ***inaccessible***.

❒ ***Example 2:*** When a ***base class*** is inherited as ***protected***, *public and protected members of the base* class become ***protected*** members of the ***derived*** class. For example following statement ***returns error*** because of *inheriting base as* ***protected*** *instead of* ***public*** in the class declaration of derived in the preceding program,

```
class derived : protected base { . . . . . .  same . . .  . }
int main(){ derived ob_drv;          /* derived type object */
          ob_drv.setab (1, 2);  /* ERROR : setab() is now a protected member of base.  setab() is not accessible here . */
```

Because ***base*** is inherited as ***protected***, its *public and protected elements* become *protected members of derived* and are therefore inaccessible within ***main( )***.

11.17 INHERITANCE with Constructors-Destructors

❒ The ***base*** class, the ***derived*** class, or ***both*** may have ***constructor*** and/or ***destructor*** functions. When a base class and a derived class both have constructor and destructor functions:

- ☞ ***Constructor (Base-class first):*** The constructor functions are executed in order of derivation. That is, the *base class constructor is executed before the constructor in the derived class*. Because a base class has no knowledge of any derived class, any initialization it performs is separate from.
- ☞ ***Destructor (Derived-class first):*** The destructor functions are executed in reverse order. That is, the *derived class's destructor is executed before the base class's destructor* because the base class underlies the derived class. If the base class's destructor were executed first, it would imply the destruction of the derived class.

❒ ***Passing arguments:*** It is possible to passed arguments to either a derived or base class constructor. There are two cases:

- ☛ ***Only derived class takes arguments:*** When only the derived class takes an initialization, arguments are passed to the derived class's constructor in the normal fashion (as we did before).

☛ ***Base class takes arguments along with derived class:*** to pass an argument to the constructor of the base class, a chain of argument passing is established.

- First, all necessary arguments to *both the base class and the derived class* are passed to the ***derived class's constructor***.
- Then using an ***expanded form*** of the *derived class's constructor declaration*, pass the appropriate arguments along to the base class.
- The ***syntax*** for passing along an argument from the derived class to the base class is:

```
derived_constructor( arg_list ): base( arg_list ){
                                  /* body of derived class constructor */
                                  }
```

  - ☑ Here ***base*** is the name of the ***base class***.
  - ☑ Both the ***derived*** class and the ***base*** class can use the *same argument* (example 3).
  - ☑ It is also possible for the ***derived class*** to *ignore all arguments* and just pass them along to the ***base*** (example 5).

❒ ***Example 1*** (Base-Derived Constructor-Destructor execution)***:***

```
class base {
public :
  base(){ cout<< "Constructing base \n"; }
  ~base(){ cout<< "Destructing base \n"; }
};
```

```
class derived : public base{ /* no-arguments allowed */
public :
   derived(){ cout<< "Constructing derived \n"; }
   ~derived(){ cout<< "Destructing derived \n"; }
};
```

```
int main() { derived obj;        /* By declaring object Constructor-Destructor executes */
             return 0; }
```

This program displays the following output:

Constructing base
Constructing derived
Destructing derived /* Reverse order*/
Destructing base /* Reverse order*/

☞ As you can see : ➲ the ***constructors*** are executed ***in order*** of derivation and
➲ The ***destructors*** are executed ***in reverse order***.

❒ ***Example 2*** (Only pass argument to derived class constructor – Normal fashion)***: The BASE is SAME as Example 1***

```
class derived : public base { int j;
                  public :
            derived(int n) { cout<< "Constructing derived \n";
                             j = n; }
            ~derived() { cout<< "Destructing derived \n"; }
            void showj() { cout << j << '\n'; }     };
```

```
int main() { derived obj(10);
             obj.showj();
             return 0; }
```

☞ Notice that the argument is passed to the derived class's constructor in the normal fashion.

❒ ***Example 3*** (Base and derived uses same arguments)***:***

```
class base {
      int i;
public :
      base(int n) {cout<< "Constructing base \n";
                   i = n;}
      ~base() { cout<< "Destructing base \n"; }
      void showi() { cout << i << '\n';}   };
```

```
class derived : public base {
       int j;
public :
derived(int n) : base(n) {  /* pass arg to base */
                 cout<< "Constructing derived \n";
                 j = n;}
~derived() { cout<< "Destructing derived \n"; }
void showj() { cout << j << '\n'; }     };
```

```
int main() { derived obj(10) ;   obj.showi();    obj.showj();     return 0; }
```

☞ In the declaration of ***derived***'s constructor, the *parameter* ***n*** (which receives the initialization argument) is both used by ***derived()*** and passed to ***base()***.In this specific case, both use the same argument, and the derived class simply passes along the argument to the base.

❒ ***Example 4*** (Base and derived uses different arguments)***:*** Mostly the constructors for the base & derived ***won't*** use ***same argument***.

☞ When this is the case and you need *to pass one or more arguments to each*:

⇨ To the ***derived class's constructor***, we must **pass those arguments**: which are *needed by both* ***derived*** *and* ***base***.

⇨ Then the derived class simply passes along to the base those arguments required by it.

```
class derived : public base { int j;
   public :
       derived(int n, int m) : base(m) {cout<< "Constructing derived \n"; j=n;}  /* pass arg to base */
       ~derived() { cout<< "Destructing derived \n"; }
       void showj() { cout<< j << '\n'; }  };
int main() { derived ob(10 , 20);    ob.showi ();    ob.showj ();     return 0; }
```

❒ ***Example 5*** (Base uses the arguments and derived just pass these to base without using)***:***

```
class base { int i;
public :
base(int n){ cout<< "Constructing base \n"; i=n;}
~base() { cout<< "Destructing base \n"; }
void showi() { cout<< i << '\n'; }
};
```

```
class derived : public base { int j;
      public :
derived(int n) : base(n)      /* pass arg to base */
      {cout<< "Constructing derived \n"; j=0; }
~derived() { cout<< "Destructing derived \n"; }
void showj() { cout << j << '\n'; }     };
```

☞ If the derived class does not need an argument, *it ignores the argument and simply passes it along*. For example, in this fragment, parameter n is not used by ***derived()***. Instead, it is simply passed to ***base()***.

11.18 MULTIPLE INHERITANCE

There are two ways that a derived class can ***inherit more than one base class***.

[1] A ***derived class*** can be used ***as*** a ***base class*** for another **derived class**, creating a **multilevel class hierarchy**. In this case,

- ✓ The original base class is said to be an ***indirect base*** class of the *second derived class*.
- ✓ NOTE: Any class-no matter how it is created-can be used as a base class.

☞ When a ***base B1*** is used for a ***derive D1*** and this derived is ***used as a base*** for another ***derived D2***. (i.e. "B1 inherited by D1" & "D1 inherited by D2"). $base_1 B1 \rightarrow derived_1 base_2\ D1 \rightarrow derived_2\ D2$. The general case of ***11.17*** appears.

- ➢ ***Constructors*** called in the ***order of derivation***. That is ***B1*** first, ***D1*** second and ***D2*** third.
- ➢ ***Destructors*** called in the ***reverse order of derivation***. That is ***D2*** first, ***D1*** second and ***B1*** third.

☞ ***Argument passing:*** When a derived class inherits a ***hierarchy of classes***, each **derived class** in the chain must ***pass back to its preceding base*** any **arguments** it needs.

[2] A ***derived class*** can **directly inherit** ***more than one base class***. In this situation,

- ✓ Two or more ***base*** classes are combined to help create the ***derived*** class.

☛ ***One derived with multiple base:*** When a **derived** class ***directly inherits*** **multiple base** classes, it uses this expanded form:

```
class derived_class_name : access base_1 , access base_2 , ... , access base_N { /* body */ }
```

- ➢ Here ***base_1*** through ***base_N*** are the **base class names** and
- ➢ ***access*** is the ***access specifier*** (private/public/protected), which ***can be different for each base*** class.

☛ ***Execution of constructors & destructors:*** When *multiple base* classes are inherited, ***constructors*** are executed in the order, ***left to right,*** *that the base classes are specified*. ***Destructors*** are executed in the ***opposite order***.

☛ ***Argument passing:*** The derived passes the necessary arguments to the *multiple base* by using this expanded form of the ***derived class's constructor function:***

```
derived_constructor(arg-list) : base_1(arg-list), base_2(arg-list),... , base_N(arg-list) { /* body */ }
```

- ➢ Here ***base_1*** through ***base_N*** are the **base class names** and

❒ ***Example 1:*** A derived class that inherits a class derived from another class.

```
class B1 { int a;
public :
B1(int x) { a = x; }
int geta() {return a;}
};
```

```
        /* Inherit direct base class.*/
class D1 : public B1 { int b;
       public :
       /* Notice how pass y to B1*/
D1(int x, int y) : B1(y) {b = x;}
int getb() {return b;}
};
```

```
    /* Inherit a derived and an indirect base .*/
class D2 : public D1 { int c;
      public :
      /* Notice how pass args to D1*/
D2(int x, int y, int z) : D1(y, z){
                                c = x; }
void show(){ cout << geta() <<' ';
             cout << getb() <<' ';
             cout << c << '\n';} };
```

```
int main() { D2 ob_d2(1, 2, 3);
             ob_d2.show();
                  /* geta() and getb() are still public here, because both are public elements of B1 and D1 */
             cout << ob_d2.geta() << ' ' << ob_d2.getb() << '\n'; return 0;}
```

☞ The call to ***ob_d2.show()*** displays ***3 2 1***. In this example, ***B1*** is an indirect base class of ***D2***.

☞ Notice that D2 has access to public elements/members of both B1 and D1, because all access specifire is public .

☞ ***Notice how arguments are passed*** along the chain from ***D2*** to ***B1***. Each class in a ***class hierarchy*** must pass all arguments required by ***each preceding base*** class. ***D1(int x, int y) : B1(y)*** and ***D2(int x, int y, int z) : D1(y, z)*** Otherwise ***compile-time error*** occurs.

❒ ***How to draw C++-style inheritance graphs:*** Traditionally, C++ programmers usually draw inheritance charts as directed graphs in which the *arrow points from the derived class to the base class*. For example the class hierarchy created in preceding program is :

D2 ⟶ **D1** ⟶ **B1**

❒ ***Example 2:*** Here a derived class directly inherits two base classes. And illustrates how the destructor and constructors are called.

```
class B1 {   int a;        /* first base class */
      public : B1(int x) { a = x; }
               int geta() { return a; }
};
```

```
class B2 {   int b;        /* second base class */
      public : B2(int x) { b = x; }
               int getb() { return b; }
};
```

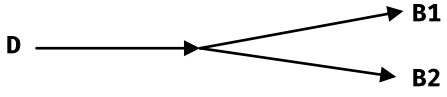
```
        /* Directly inherit two base classes .*/
class D : public B1, public B2{ int c;
      public :
// here z and y are passed directly to B1 and B2
D(int x, int y, int z) : B1(z), B2(y) {c=x;}
void show() {cout<< geta() <<' '<< getb()<<' ';
             cout << c << '\n';} };
```

```
int main(){  D ob_d(1, 2, 3);
             ob_d.show();
             return 0;    }
```

The call to ***ob_d2.show()*** displays ***3 2 1.***

☞ Because bases inherited as ***public*** , ***D*** has access to *public elements* of both ***B1*** and ***B2***.

☞ The ***arguments*** to ***B1*** and ***B2*** are passed *individually* to these classes by ***D***. This program creates a class that looks like this:



❒ ***Example 3:*** The following program illustrates the ***order*** in which ***constructor and destructor*** functions are called when a derived ***directly inherits*** multiple base:

```
class B1 {
public :
 B1(){cout< " Constructing B1\n";}
~B1(){cout<<" Destructing B1\n";}};
```

```
class B2 {
public :
 B2(){cout< " Constructing B2\n";}
~B2(){cout<<" Destructing B2\n";}};
```

```
class D : public B1, public B2{
public :
D(){cout< " Constructing D\n";}
~D(){cout<<" Destructing D\n";}  };
```

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

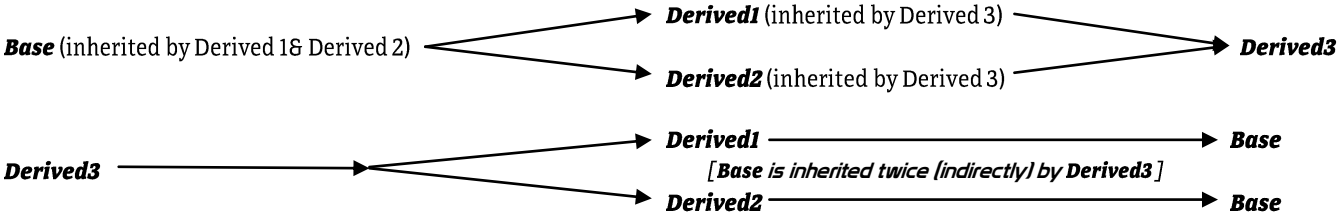
This program displays :

```
Constructing B1
Constructing B2
Constructing D
Destructing D
Destructing B2
Destructing B1
```

when multiple direct base classes are inherited, constructors are called in order, left to right, as specified in the inheritance list. Destructors are called in reverse order.

11.19 VIRTUAL BASE (problems with "one derived" & "multiple direct base")

❒ A potential problem exists when *multiple base classes are directly inherited by a derived class*. Consider the following class hierarchy:



(a) Here the base class **Base** is inherited by both **Derived1** and **Derived2**.
(b) **Derived3** directly inherits both **Derived1** and **Derived2**.

☞ These implies that **Base** is actually inherited (indirectly) ***twice*** by **Derived3** -first ***through* Derived1**, and then again ***through* Derived2**.

☞ This *causes ambiguity* when a " ***member of*** Base " is used by **Derived3**. Since two copies of **Base** are included in **Derived3**, which member should it ***refer/use***.

❒ To resolve this ambiguity: in which a derived class indirectly inherits the same base class more than once. We use the ***virtual base class***.

☛ If the **base** class ***inherited as virtual*** by any **derived** classes, it prevents two copies of the base from being present in the derived object.

☛ The ***virtual*** keyword ***precedes*** the *base class access specifier* when it is inherited by a derived class.

❒ ***Example1:*** Here ***virtual base*** class prevents two copies of base from being present in ***derived3***.

```
class base {
   public :
       int i;
};
```

```
/* Base as virtual.*/
class derived1 : virtual public base {
              public :  int j;
};
```

```
/* base as virtual here , too.*/
class derived2 : virtual public base {
              public : int k;
};
```

```
/* only one copy of base is present */
class derived3 : public derived1, public derived2
{
public :
       int product() { return i*j*k; }
};
```

```
int main() { derived3 ob;
       ob.i=10; // unambiguous : only one copy present
       ob.j=3;
       ob.k=5;
       cout<< "Product is"<< ob.product() <<'\n';
return 0; }
```

☞ If ***derived1*** and ***derived2*** had not inherited ***base*** as ***virtual***, the statement ***ob.i = 10;*** would have been *ambiguous* and a *compile-time error* would have *resulted*.

❒ It is important to understand that when a ***base*** is inherited as ***virtual*** by a ***derived***, that ***base*** still exists within that ***derived***. For example, this fragment is perfectly valid:

```
derived1 ob_drv1;   /* we've used derived3 type object in the above */
ob_drv1.i = 100;
```

☛ The only difference between a ***normal base*** and a ***virtual base*** occurs when an object inherits the base more than once.