Chapter: 10

**C++ Class, Objects with array and pointer**

Introduction to *Encapsulation* and *Inheritance* with array and pointer

**10.1 Introduction to CLASS**

The class is at the heart of many C++ features. The class is the mechanism that is used to create objects. A class is declared using the class keyword. The syntax of a class declaration is similar to that of a structure (actually *class is a kind of structure with more feature*). Its general form is shown here:

**class class\_name{**

/\* private functions and variables \*/

**public :**

/\* public functions and variables \*/

**} object\_list ;**

* Similar to a structure declaration in a class declaration, the object-list and class\_name are *optional*. From a practical point of view class\_name is virtually always needed. The reason for this is that the class\_name becomes a new type name that is used to declare objects of the class.
* Private and public members : Functions and variables declared inside a class declaration are said to be members of that class.
* By default, all *functions* and *variables* declared inside a class are private to that *class*. This means that they are accessible, *only by other members of that class*.
* To declare *public class members*, the public keyword is used, followed by a colon. All *functions* and *variables* declared after the public *specifier* are accessible both by *other members* of the class and by any *other part* of the program that contains the class. Here is a simple class declaration:

**class** myclass{

*/\* private to myclass \*/*

**int** a;

**public**:

**void** set\_a(**int** num ); */\* prototype \*/*

**int** get\_a();}; */\* prototype \*/*

* This class has one private variable, called a, and two public functions, ***set\_a()*** and ***get\_a()***. Functions that are declared to be part of a class are called member functions. Notice that ***set\_a(int num );*** and ***int get\_a();*** are *function prototype declaration inside the class*.
* Since a is private, it is not accessible by any code outside ***myclass***. since ***set\_a()*** and ***get\_a()*** are members of ***myclass***, they *can directly access* a.
* Further, ***set\_a()*** and ***get\_a()*** are declared as public members of ***myclass*** and can be called by *any other part* of the program that contains ***myclass***.
* Defining the member functions: Although the prototypes of the functions ***set\_a()*** and ***get\_a()*** are declared by ***myclass***, they are not yet defined.
* To define a *member function*, you must link the type name of the class with the *name of the function*. You do this by *preceding the function name with the class name followed by two colons*.
* The two colons are called the scope resolution operator.
* In general, *to define a member function* you must use this form:

**ret\_type class\_name :: func\_name ( parameter\_list ) {**

*/\* body of function \*/*

**}**

Here class\_name is the name of the class to which the *function belongs* and ret\_type is the *return type* of the function which *must be the same return type of the prototype* declared inside the class.

For example, here member functions ***set\_a()*** and ***get\_a()*** are defined:

**void** myclass **::** set\_a( **int** num ) { a = num; }

**int** myclass **::** get\_a() { **return** a; }

* Crating object of a class : This is similar to declaring a structure variable in C. Since class is similar to structure and object is similar to variable. The declaration of a class did not define any objects of type of that class - it only defines *the type of object* that will be created when one is *actually declared*.

To create an object, use the class name as a type specifier

***class\_name object\_name1, object\_name2, . . . , object\_nameN;***

For example, this line declares two objects of type *myclass*:

**myclass** ob1 , ob2 ; /\* these are objects of type myclass \*/

* A *class declaration* is a logical abstraction that defines a new type. It determines what *an object of that type will look like*,
* An *object declaration* creates a physical entity of that type. That is, an object occupies memory space, but a type definition does not.
* Like variables, there is local objects and global objects.
* Accessing members of an object: Once an object of a class has been created, your program can reference its public members by using the dot (period) operator in much the same way that structure members are *accessed*.

Assuming the preceding object declaration, the following statement calls ***set\_a()*** for objects ***ob1*** and ***ob2***:

**ob1.set\_a(10);** */\* sets ob1 's version of a to 10 \*/*

**ob2.set\_a(99);** */\* sets ob2 's version of a to 99 \*/*

* *Each object of a class has its own copy of every variable declared within the class.* Previous statements set ***ob1***'s copy of a to ***10*** and ***ob2***'s copy to ***99***. This means that ***ob1***'s ***a*** is distinct and different from the ***a*** linked to ***ob2***.

EXAMPLES

**10.2 CONSTRUCTOR and DESTRUCTOR Functions**

|  |  |
| --- | --- |
| General form of constructor | General form of destructor |
| **class class\_name{** */\* private functions and variables \*/*  **public :** */\* public functions and variables \*/* **class\_name();** */\* constructor \*/*  **} object\_list ;** | **class class\_name{** */\* private functions and variables \*/*  **public :** */\* public functions and variables \*/ ~***class\_name();** */\* destructor \*/*  **} object\_list ;** |

* Object initialization using "constructor": When applied to real problems, virtually every object you create will require some sort of initialization. To address this situation, C++ allows a constructor function to be included in a class declaration. A class’s *constructor is called each time an object of that class is created*. Thus, any initialization that needs to be performed on an object can be done automatically by the constructor function.
* A constructor function has the same name as the class of which it is a part and has no return type. According to the C++ formal syntax rules, it is *illegal for a constructor to have a return type*.
* The constructor is called when the object is created. An object is created when that object's declaration statement is executed.
* For global objects, an object's constructor is called once, when the program first begins execution. For local objects, the constructor is called each time the declaration statement is executed. Example:

#include <iostream>

**using namespace std;**

**class** myclass{**int** a;

**public** :

**myclass**(); /\* constructor \*/

**void** show(); };

myclass **::** **myclass**() {

**cout** << "In constructor \n"; a = 10; }

**void** myclass **::** show() {**cout** << a; }

**int** main() {

**myclass** ob; /\* object declaration \*/

ob.show(); /\* calling function \*/

**return** 0;}

* Destroy objects using "destructor": The *complement of a constructor* is the destructor. This function is called when an object is destroyed. For example, an object that allocates memory when it is created will want to free that memory when it is destroyed. The *name of destructor-function* is the name of its class, preceded by a "tilde" **~** .
* It is not possible to take the address of either a constructor or a destructor.
* A class's destructor is called when an *object is destroyed*. Local objects are destroyed when they go out of scope. Global objects are destroyed when the program ends. Example:

**class** myclass{ **int** a;

**public** : **myclass**(); /\* constructor \*/

**~myclass**(); /\* destructor \*/ **void** show(); };

myclass **::** **myclass**() {

**cout** << "In constructor \n"; a = 10; }

myclass **::** **~myclass**() {

**cout** << "Destructing. . .. \n";}

**void** myclass **::** show() {**cout** << a <<"\n"; }

**int** **main**() { **myclass** ob; /\* object declaration \*/

ob.show(); /\* calling function \*/

**return** 0;}

* Difference between variable and object declaration: It is important to understand that in C++, a *variable declaration statement* is an "action statement." When you are programming in C, it is easy to think of declaration statements as simply establishing variables. However, in C++, because an object might have a constructor, a variable (i.e. object ) declaration statement may, in fact, cause a considerable *number of actions to occur*.
* Other use Restriction: Having a constructor or destructor perform actions not directly related to the initialization or orderly destruction of an object makes for very *poor programming style* and should be avoided. Technically, a constructor or a destructor can perform *any type of operation*. The code within these functions *does not have to* initialize or reset anything related to the class for which they are defined. For example, a constructor for the preceding examples could have computed pi to 100 places.

**10.3 Constructors with Parameters**

It is possible to pass arguments to a *constructor function*. To allow this, simply add the appropriate parameters to the constructor function's declaration and definition. Then, when you declare an object, specify the arguments. By *Constructors with Parameters* we can initialize *different objects with different values* as we want.

**class class\_name{** /\* private functions and variables \*/

**public :** /\* public functions and variables \*/

**class\_name(type p1, type p2, . . . type pN);** /\* constructor with parameter \*/

**};** /\* no object list. Objects declared separately \*/

. . .

. . .

**class\_name** **object1**(p1, p2, . . . , pN); /\* passing arguments to constructor \*/

* Actually, the syntax for passing an argument to a *parameterized constructor* is shorthand for this longer form (which use constructor function):

***class\_name object1 = class\_name(p1, p2, . . . , pN);***

however the short-hand form is often used : ***class\_name object1(p1, p2, . . . , pN);***

* Destructor functions cannot have parameters. The reason is there exists no mechanism by which to pass arguments to an object that is being destroyed.

**Example: class** myclass{ **int** a;

**public** : **myclass**( **int** x ); /\* constructor with parameter \*/

**void** show(); };

myclass :: **myclass**( **int** x ) { **cout** << "In constructor \n"; a = x; }

**void** myclass **::** show() {**cout** << a <<"\n"; }

**int** **main**() { myclass ob(4); /\* object with arguments to constructor \*/

ob.show(); /\* calling function \*/

**return** 0;}

* Here the constructor for ***myclass*** takes one parameter.
* The value passed to ***myclass()*** is used to initialize a.
* ***ob*** is declared in ***main()*** with arguments ***ob(4)***. The value ***4***, specified in the parentheses following ***ob***, is the argument that is passed to ***myclass()***'s parameter x, which is used to initialize a.

Note

1. However, most C++ programmers use the short form. Actually, there is a slight technical difference between the two forms that relates to copy constructors.
2. You can pass an object's constructor *any valid expression*, including variables. Actually objects can be constructed as needed *to fit the exact situation* at the time of their creation.

**10.4 Relation between STRUCTURES-UNIONS and CLASSES**

* Structures: The class and the structure have virtually *identical capabilities*. In C++ structures include member functions, including constructor and destructor functions,. In fact, the only difference between a *structure and a class* is that, by default, the *members of a class are private* but the *members of a structure are public*. Because in C all structure members are public by default. General form of a structure in C++ :

**struct type\_name {**

/\* public function and data members \*/

**private :**

/\* private function and data members \*/

**} object\_list ;**

* In C++ both struct and class create new *class types*.
* Notice that a new keyword is introduced. It is private, and it tells the compiler that the member that follow are *private to that class* (applicable for both class and structure).
* Unions: In C++, a union defines a class type that can contain *both functions and data as members*. For a union all members are public by default until the private specifier is used.
* In a union, however, all data members share the *same memory location* (just as in C).
* Unions can contain constructor and destructor functions.
* The union's ability to *link code and data* allows you to create class types in which *all data uses a shared location*. This is something that you cannot do using a class.
* There are several restrictions that apply to unions as they relate to C++.
* First, they *cannot inherit* any other class and they *cannot be used* as a base class for any other type.
* Unions also *must not contain any object* that has a constructor or destructor. The union, *itself, can have a constructor and destructor though*.
* Unions must not have any static members.
* Finally, unions cannot have virtual member functions. (Virtual functions are described later.)
* Anonymous union: An anonymous union special type of union that *does not have a type name*, and *no variables can be declared* for this sort of union. Instead, it tells the compiler that its members will share the same memory location.
* However, in all other, respects, the members act and are treated like normal variables. That is, the members are *accessed directly, without the dot operator syntax*. For example, examine this fragment:

**union** { **int** i; **char** ch [4]; }; /\* an anonymous union \*/

i = 10; ch [0] = 'X'; /\* access i and ch directly \*/

Here i and ch are accessed directly because they are not part of any object. They share the same *memory space*.

* Anonymous union is that it gives you a simple way to tell the compiler that you want two or more variables to share the same memory location.
* Aside from this special attribute, members of an anonymous union behave like other variables.
* Anonymous unions have *all of the restrictions that apply to normal unions*, plus these additions.
* A global anonymous union must be declared static.
* An anonymous union cannot contain private members.
* The names of the members of an anonymous union must not conflict with other identifiers within the same scope.

Note

1. Although structures have the same capabilities as classes, most programmers restrict their use of structures to adhere to their *C-like form* and do not use them to include function members.
2. We reserve the use of struct for objects that have *no function members*.

**10.5 In-Line Functions & Automatic In-Lining**

* IN-LINE FUNCTIONS: In C++, it is possible to define functions that are *not actually called* but, rather, are *expanded in line*, at the point of each call. This is much the same way that a *C-like parameterized macro* works (re call 8.1).
* The advantage is, no overhead associated with the function *call and return* mechanism. This means that in-line functions can be executed much faster than normal functions. *(Normal function call and return take time each time a function is called. If there are parameters, even more time overhead is generated.)*
* The disadvantages of *in-line functions* is that if they are *too large* and called too often, your program *grows larger*. For this reason, in general *only short functions are declared as in-line* functions.
* *To declare an* in-line function*, simply precede the function's* "definition" *with the* "inline" *specifier*. For Example :

**inline** **int** even(**int** x) { **return** !(x%2); }

The function ***even()***, which returns true if its argument is even, is declared as being-in-line.

It means that, **if**(even(x)) **cout** << "Even"; is functionally equivalent to: **if**(!(x%2)) **cout** << " Even ";

* An *in-line function* must be defined before it is first called. If it isn't, the compiler has no way to know that it is supposed to be expanded *in-line*. This is why ***even()*** was defined before ***main()***.

**inline** **int** even(**int** x) { **return** !(x%2); }

**int** **main**(){ . . . . . . . . . }

* It is important to understand that the inline specifier is a request, *not a command*, to the compiler.
* If any *in-line restriction is violated*, the compiler is free to *generate a normal function* (i.e. the function is compiled as a normal function and the inline request is ignored.). Some compilers will *not in-line a function* if it contains a static variable, a loop statement, a switch or a goto, or if the function is recursive.
* AUTOMATIC IN-LINING: If a member function's definition is short enough, the definition can be included inside the class declaration. Doing so causes the function to *automatically become an in-line function*, if possible.
* When a function is defined *within a class declaration*, the inline keyword is *no longer necessary*.
* Of course if any *in-line restriction is violated*, the compiler is free to *generate a normal function*.
* For example, the ***divisible()*** is automatically in-lined as shown here:

**class** samp { int i, j;

**public**:

samp (int a, int b);

/\* divisible() is defined here and automatically in - lined . \*/

**int** divisible() { **return** !(i%j);}

};

samp :: **samp**(**int** a, **int** b) { i = a; j = b; }

* As you can see, the code associated with ***divisible()*** occurs inside the declaration for the class samp. Further notice that *no other definition* of ***divisible()*** is *needed-or permitted*.
* Notice how ***divisible()*** *occurs all on one line*. This format is very common in C++ programs when a function is declared within a class declaration. It allows the declaration to be more compact.

**10.6 Assigning Objects**

One object can be assigned to another provided that *both objects are of the same type* (similar to structure). By default, when one object is assigned to another, a bitwise copy of all the data members is made.

* For example, the contents of object called ***obj\_1*** is assigned to another object called ***obj\_2***, the contents of all of ***obj\_1'***s data are copied into the equivalent members of ***obj\_2***.

#include <iostream >

**using namespace std**;

**class** myclass {

**int** a, b;

**public** :

**void** set(**int** i, **int** j) {a = i; b = j; }

**void** show() {**cout** << a << ' ' << b << "\n";}

};

**int** **main**(){ **myclass** obj\_1 , obj\_2;

obj\_1.set(10 , 4);

/\* assign obj\_1 to obj\_2 \*/

obj\_2 = obj\_1;

obj\_1.show();

obj\_2.show();

**return** 0;}

* Here, object ***obj\_1*** has its member variables ***a*** and ***b*** set to the values ***10*** and ***4***, respectively. Next, ***obj\_1*** is assigned to ***obj\_2***. This causes the current value of ***obj\_1.a*** to be assigned to ***obj\_2.a*** and ***obj\_1.b*** to be assigned to ***obj\_2.b***.
* An assignment between two objects simply makes the data in those objects identical. The two objects are *still completely separate*, For example, after the assignment, calling ***obj\_1.set()*** to set the value of ***obj\_1.a*** has no effect on ***obj\_2*** or its a value.
* *Only objects of the same type* can be used in an assignment statement. If the objects are not of the same type, a *compile-time error* is reported. It is *not* *sufficient* that the types just be *physically similar*-their *type* names *must* be the *same*.

**class** myclass {

**int** a, b;

**public** :

**void** set(**int** i, **int** j) {a = i; b = j; }

**void** show() {**cout** << a << ' ' << b << "\n";}

};

**class** yourclass {

**int** a, b;

**public** :

**void** set(**int** i, **int** j) {a = i; b = j; }

**void** show() {**cout** << a << ' ' << b << "\n";}

};

**int** **main**(){ **myclass** obj\_1 ;

**yourclass** obj\_2;

obj\_1.set(10 , 4);

/\* assign obj\_1 to obj\_2 \*/

obj\_2 = obj\_1; /\* compile-time error will occur: not same type \*/

obj\_1.show(); obj\_2.show();

**return** 0;}

In this case, even though myclass and yourclass are physically the same, because they have different type names, they are treated as differing types by the compiler.

* *All data members of one object* are assigned to another when an assignment is performed. This includes compound data such as arrays.

**10.7 Object Pointers**

It is possible to access a *member of an object* via pointer to that object. When a pointer is used, the arrow operator (**->**) rather than the dot operator is employed. (This is exactly the same way the arrow operator is used when given a pointer to a structure.)

*However using the dot operator to access members of an object is the correct method*.

* You declare an object pointer just like you declare a pointer to any other variable. Specify its class name, and then precede the variable name with an asterisk.
* To obtain the address of an object, precede the object with **&** operator, just as you do when taking the address of any other type of variable.
* Just like pointers to other types, an object pointer, when incremented, will point to the next object of its type.
* It is important to understand that *creation of an object pointer does not create an object-it creates just a pointer to one*.

For example

**int** myclass :: get() { **return** a; }

**int** **main**() { **myclass** ob (120) ; /\* create object \*/

**myclass** \*p; /\* create pointer to object \*/

p = &ob; /\* put address of ob into p \*/

**cout** << " Value using object : " << ob.get();

**cout** << "\n";

**cout** << " Value using pointer : " << p->get();

**return** 0; }

* Notice how the declaration ***myclass \*p;*** creates a pointer to an object of myclass.
* The address of ob is put into p by using this statement: ***p = &ob;***
* Finally, the program shows how the members of an object can be accessed through a Pointer as using : ***p->get()***  .
* Pointer arithmetic using an object pointer is the same as it is for any other data type: it is performed relative to the type of the object. For example, when an object pointer is *incremented, it points to the next object*. When an object pointer is *decremented, it points to the previous object*.

**class** samp { **int** a, b;

/\* inline function as constructor \*/

**public** : samp(**int** n, **int** m) { a=n; b=m; }

int get\_a() { **return** a; }

int get\_b() { **return** b; } };

**int** **main**() {

**samp** ob[4] = {samp(1, 2), samp(3, 4),

samp(5, 6), samp(7, 8) };

**int** i;

**samp** \*p;

p = ob; /\* get starting address of array \*/

**for** (i=0; i <4; i**++**){

**cout** << p**->**get\_a () << ' ';

**cout** << p**->**get\_b () << "\n";

p**++**; /\* advance to next object \*/ }

. . . . . }

* Each time p is incremented, it points to the next object in the array.

**10.8 The "*this*" pointer**

C++ contains a special pointer that is called this. this is a pointer that is *automatically passed to any member function* when it is called, and it is a *pointer to the object* that generates the call (this works *implicitly*). For example, given this statement,

***ob.f1();*** /\* accessing f1() member function from object ob \*/

the function ***f1()*** is automatically passed a pointer to ***ob***-which is the object that invokes the call. This pointer is referred to as this.

* Only member functions are passed a this pointer. For example, a friend does not have a this pointer.
* When a member function refers to another member of a class, it does so directly without qualifying the member with either a class or an object specification. For example,

**class** inventory { **char** item[20];

**double** cost ;

**int** on\_hand ;

**public** :

inventory( **char** \*i, **double** c, **int** o) {

**strcpy** (item , i);

cost = c; on\_hand = o; }

**void** show (); };

**void** inventory **::** show(){

**cout** << item ;

**cout** << ": $" << cost ;

**cout** << " On hand : " << on\_hand << "\n"; }

Here, within the constructor ***inventory()*** and the member function ***show()***, the member variables item, cost, and on\_hand are referred to directly.

**class** inventory { **char** item[20];

**double** cost ;

**int** on\_hand ;

**public** :

inventory( **char** \*i, **double** c, **int** o) {

/\* accessing through "this" pointer \*/

**strcpy** (this**->**item , i);

this**->**cost = c;

this**->**on\_hand= o; }

**void** show (); };

**void** inventory **::** show(){

/\* use "this" to access members\*/

**cout** << this**->**item ;

**cout** << ": $" << this**->**cost ;

**cout** << " On hand : " << this**->**on\_hand << "\n"; }

* Here the member variables are accessed explicitly through the this pointer.
* Shorthand-form: within ***show()***, these two statements are equivalent:

***cost = 123.23;***

***this -> cost = 123.23;***

the first form is a shorthand for the second.

Note

1. By default, all member functions are *automatically* *passed* a pointer to the invoking object.
2. The this pointer has several uses, including aiding in overloading operators.

**10.9 ARRAYS OF OBJECTS**

* The syntax for declaring an *array of objects* is exactly like that used to declare an *array of any other type of variable*.
* *Arrays of objects* are accessed just like *arrays of other types of variables*.

Example: **class** samp { **int** a;

**public** : **void** set\_a(**int** n) { a = n; }

**int** get\_a() { **return** a; } };

**int** main() { **samp** ob [4]; **int** i;

**for** (i=0; i<4; i++) ob[i].set\_a(i);

**for** (i=0; i<4; i++) **cout**<< ob[i].get\_a();

**cout**<< "\n";

**return** 0;}

This creates a *four-element array* of objects of type samp and then loads each element's a with a value between ***0*** and ***3***. The *array name*, in this case ob, is indexed; then the *member access operator* is applied, followed by the name of the member function to be called.

* Initialization list (short and long form) for array with constructor: If a class type includes a constructor, an array of objects can be initialized. For example, here ob is an initialized array:

**class** samp { **int** a;

**public** :samp(**int** n) { a = n; } /\* constructor for initialization\*/

**int** get\_a() { **return** a; } };

**int** main() { **samp** ob [4] = { -1, -2, -3, -4 };

. . . . . . }

Actually, the syntax shown in the initialization list samp ob [4] = { -1, -2, -3, -4 }; is shorthand for this longer form (first shown in 10.3 ):

**samp** ob[4] = { samp( -1), samp( -2),

samp( -3), samp( -4) };

However, the shorthand form used in the program is applicable when constructors take only one argument. For constructors with two or more argument (multidimensional arrays of objects) we have to use the longer form. For example,

**class** samp { **int** a, b;

**public** : samp(**int** n, **int** m) { a = n; b = m; } /\* initialization for 2-D array \*/

**int** get\_a() { **return** a; }

**int** get\_b() { **return** b; } };

**int main**() { **samp** ob[4][2] = { samp(1, 2), samp(3, 4),

samp(5, 6), samp(7, 8),

samp(9, 10), samp(11 , 12),

samp(13 , 14), samp(15 , 16) };

. . . . . }

* The reason is, the formal C++ syntax allows only one argument at a time in a *comma-separated list*. There is no way, for example, to specify two (or more) arguments per entry in the list.
* Therefore, when you initialize arrays of objects that have constructors that take *more than one argument, you must use the "long form" initialization syntax* rather than the "shorthand form."
* You can *always use the long form of initialization even if the object takes only one argument*.

**10.10 PASSING objects to functions and RETURNING objects from function**

* PASSING objects to functions: Objects can be *passed to functions* as arguments in just the same way that other types of data are passed. Simply *declare the function's parameter as a class type* and then use an object of that class as an argument when calling the function.

***type function\_name( class\_type obj\_1, class\_type obj\_2, . . . . );***

* As with other types of data, by default all objects are *passed by value to a function*.

Example: in following program we *pass object to a function*

#include <iostream >

**using namespace std**;

**class** samp{ **int** i;

**public** :

samp(**int** n) { i = n; }

**int** get\_i() { **return** i;} };

/\* Return square of ***obj.i***. i.e. square of I of an object obj \*/

**int** sqr\_it(**samp** obj){

**return** obj.get\_i() \* obj.get\_i(); }

**int** **main**(){ **samp** a(10) , b(2) ;

**cout** << sqr\_it(a) << "\n";

**cout** << sqr\_it(b) << "\n";

**return** 0;}

Here ***sqr\_it()*** takes an argument of type samp and returns the square of that object's i value.

* The default method of parameter passing in C++, including objects, is by value. This means that a bitwise copy of the argument is made and it is *this copy that is used by the function*. Therefore, changes to the object inside the function do not affect the calling (original) object. *Objects , like other parameters , are passed by value . Thus changes to the parameter inside a function have no effect on the object used in the call* . Example :

#include <iostream >

**using namespace std**;

**class** samp{ **int** i;

**public** :

samp(**int** n) { i = n; }

**void** set\_i(**int** n) { i=n; }

**int** get\_i() {**return** i;} };

output : Copy of a has i value of 100

But, a.i is unchanged in main: 10

*/\* Set* ***obj.i*** *to its square . This has no effect on the*

*object used to call* ***sqr\_it()*** *\*/*

**void** sqr\_it( **samp** obj) {

obj.set\_i( obj.get\_i() \* obj.get\_i() );

**cout** << "Square = Copy of a has i value of :" ;

**cout** << obj.get\_i(); }

**int** **main**(){ **samp** a(10); /\*passed by value\*/

**cout** << "But , a.i is unchanged in *main* : ";

**cout** << a.get\_i();

**return** 0;}

* The address of an object can be passed to a function so that the argument used in the call can be modified by the function. That is, *changes to the object inside the function will affect the calling (original) object*. Let's consider the class of the previous example. If we change the ***sqr\_it()*** like below: it will modify the value of the object whose address is used in the call to ***sqr\_it()***.

*/\* Set* ***obj.i*** *to its square . This affects the*

*object used to call* ***sqr\_it()*** *\*/*

**void** sqr\_it( **samp** \*obj) {

obj->set\_i( obj->get\_i() \* obj->get\_i() );

**cout** << "Square = Copy of a has i value of :" ;

**cout** << obj->get\_i(); }

**int** **main**(){ **samp** a(10); /\*passed by value\*/

**cout** <<"Now , a.i is changed in *main* : ";

**cout** << a.get\_i();

**return** 0;}

output : Copy of a has i value of 100

Now, a.i is changed in main: 100

* These two example reflects the same thing that we've discussed in C's passing argument's address in function parameters ( recall 5.3 ).
* When a *copy of an object* is created because it is used as an argument to a function, the *constructor function is not called*. However, when the copy is destroyed (by going out of scope when the function returns), the *destructor function is called*.
* The reason for not calling the constructor function is that, the *constructor* function is generally *used* *to initialize some aspect of an object*, it must not be called when making a copy of an already existing object passed to a function. Doing so would *alter the contents of the object*. When passing an object to a function, *you want the current state of the object, not its initial state*.
* *Destructor* function is *called* because the object might perform *some operation that must be undone* when it goes out of scope. For Example :

**class** samp { **int** i;

**public** :

samp(**int** n) { i=n; */\* constructor \*/*

**cout** << " Constructing \n"; }

~samp() { */\* destructor \*/*

**cout** << " Destructing \n"; }

**int** get\_i() { **return** i; } };

/\* Return square of obj.i.\*/

**int** sqr\_it( **samp** obj){

**return** obj.get\_i() \* obj.get\_i(); }

**int** **main**() { **samp** a (10) ;

**cout** << sqr\_it (a) << "\n";

**return** 0; }

Outout : Constructing

Destructing

100

Destructing

As you can see, only one call to the constructor function is made. This occurs when a is created. However, two calls to the destructor are made. One is for the copy created when a is passed to ***sqr\_it().*** The other is for a itself.

* One important point : when an object is passed to a function, a copy of that object is made. Further, when that function returns, the copy's destructor function is called. The fact that the destructor for the object that is the copy of the argument is executed when the function terminates can be a source of problems. *For example,* if the object used as the argument allocates dynamic memory and frees that memory when destroyed, *its copy will free the same memory when its destructor is called*. This will leave the original object damaged and effectively useless. This problem can be resolved in two ways : one, using reference. Two, using copy-constructor.
* RETURNING OBJECTS FROM FUNCTIONS: To return objects from a function,

1. First declare the function as returning a class type.
2. Second, return an object of *that type* using the normal return statement.

***ret\_class\_type function\_name( any\_type obj\_1, any\_type obj\_2, . . . . );***

* Here is an example of a function that returns an object:

#include <iostream >

#include <cstring >

**using namespace std;**

**class** samp {**char** s[80];

**public** :

**void** show() { **cout** << s << "\n"; }

**void** set( **char** \*str ) { **strcpy**(s, str ); } };

/\* Function input() return an object of type samp \*/

**samp** input(){ **char** g[80]; /\* it is a local variable. However s[80] used in book. Which have no

connection with the private variable s[80] in the class samp \*/

**samp** str ; /\* declaring local object str \*/

**cout** << " Enter a string : "; **cin** >> g;

str.**set**(g); /\* copying g to str.s \*/

**return** str; } /\* returning object \*/

**int** **main**(){ **samp** ob;

ob = **input**(); /\* assign returned object to ob \*/

ob.**show**();

**return** 0;}

In this example, ***input()*** creates a local object called ***str*** and then reads a string from the keyboard. This string is copied into ***str.s*** and then ***str*** is returned by the function. This object is then assigned to ***ob*** inside ***main()*** when it is returned by the call to ***input()***. Notice in the book a private variable ***s[80]*** used in the function ***input()***.

* *One important point is :* When an object is returned by a function, a *temporary object is automatically created* which holds the return value. It is this object that is actually *returned by the function*. After the value has been returned, this object is destroyed. The destruction of this *temporary object* might cause unexpected side effects in some situations *(This problem can be resolved in two ways : one, using reference. Two, using copy-constructor.).*
* You must be careful about returning objects from functions if those objects contain destructor functions because the returned object goes out of scope as soon as the value is returned to the calling routine. For example, if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that is assigned the return value is still using it. Consider the program above with newly defined class, constructor and set() :

#include <cstdlib> /\* we'll use allocation function malloc() and free() \*/

**class** samp {**char** \*s;

**public** :

samp() {s="\0";} /\* "\0" means null \*/

~samp() {**if**(s) **free**(s); **cout**<< "freeing s \n";} /\* freeing memory \*/

**void** show() { **cout** << s << "\n"; }

**void** set( **char** \*str ); };

/\* load a string \*/

**void** samp :: **set**( **char** \*str ){ s=(**char** \*)**malloc**(**strlen**(str)+1); /\* allocating memory \*/

**if**(!s){**cout**<< "Allocation error \n"; **exit**(1);}

**strcpy**(s, str); }

In this case different error arise in different compiler. Here destructor called multiple times (In old complier three times actually).

1. First, it is called when the local object str goes out of scope when ***input()*** returns.
2. The second time ***~samp()*** is called is when the temporary object returned by ***input()*** is destroyed. Remember, *when an object is returned from a function, an invisible (to you) temporary object is automatically generated which holds the return value*. In this case, this object is simply a copy of str, which is the return value of the function. Therefore, destructor is executed.
3. Finally, the destructor for object ob, inside ***main()***, is called when the program terminates.

The trouble is that in this situation, the first time the destructor executes, the memory allocated to hold the string input by ***input()*** is freed. Thus, *not only do the other two calls to* ***samp***'s *destructor try to free an already released piece of dynamic memory, but they destroy the dynamic allocation system in the process*, as evidenced by the run-time message "Null pointer assignment." (Depending upon your compiler, you may or may not see this message).

* The key point to be understood from this example is that when an object is returned from a function, the temporary object used to effect the return will have its destructor function called. Thus, you should avoid returning objects in which this situation is harmful. (However, it is possible to use a copy constructor to manage this situation.)

**10.11 Memory allocation/release operators : *new*, *delete***

To allocate memory and to free the allocated memory we use the C's dynamic allocation functions ***malloc()*** and ***free()*** respectively (applicable in C++ also). These are the standard.

Memory allocation/release operators : C++ provides safer and more convenient operators (not functions): new and release to *allocate* and *free* (delete) memory respectively. These operators take these general forms:

p\_var = **new** type ;

**delete** p\_var ;

Here type is the type specifier of the object for which you want to allocate memory and p\_var is a pointer to that type.

* new is an operator that *returns* a pointer to dynamically allocated memory that is *large enough* to hold an object of type type.
* In Standard C++, the default behavior of new is to generate an exception when it *cannot satisfy* an allocation request. If this exception is not handled by your program, your program will be terminated. (*Exceptions* and *exception handling* are described later; loosely, an *exception is a run-time error that can be managed in a structured fashion*.)
* delete releases that memory when it is no longer needed. delete can be called only with an *invalid pointer*, the allocation system will be destroyed, possibly crashing your program.
* If there is insufficient available memory to fill an allocation request, one of two actions will occur. Either new will return a null pointer or it will generate an exception.

Example: // A simple example of new and delete .

# include <iostream >

**using namespace std;**

**int** **main**() {**int** \*p;

p = **new** **int**; /\* allocate room for an integer \*/

**if**(!p) {**cout** << "Allocation error \n";

**return** 1;}

\*p = 100;

**cout** << "Here is integer at p:" << \*p << "\n";

**delete** p; /\* release memory \*/

**return** 0;}

* Initializing dynamic variable : *Dynamically allocated objects* can be given initial values. We can give a dynamically allocated object an initial value by using this form of the new statement:

p\_var = **new** **type**( initial\_value );

Example 1: **class** **samp** { **int** i, j;

**public** :

samp(**int** a, **int** b) { i=a; j=b; }

**int** get\_product() { **return** i\*j; } };

**int** **main**() { **samp** \*p;

p = **new** **samp**(6, 5); /\* allocate *object* with initialization \*/

**if**(!p) { **cout** << " Allocation error \n"; **return** 1; } /\*allocation check\*/

**delete** p; /\* release memory \*/

**return** 0; }

Example 2:To initialize an integer variable,

p = **new** **int**(9); /\* give initial value of 9\*/

**if**(!p) { **cout** << " Allocation error \n"; **return** 1; } /\*always check if allocation is successful\*/

. . . . . . *STATEMENTS*. . . . . .

**delete** p; /\* release memory \*/

* Dynamically allocate a one-dimensional array : To dynamically allocate a one-dimensional array, use this form of new:

p\_var = **new** **type**[ size ];

After this statement has executed, ***p\_var*** will point to the *start of an array* of ***size*** elements of the ***type*** specified.

* For various technical reasons, it is *not possible* to initialize an array that is *dynamically allocated*.
* To delete a *dynamically allocated array*, use this form of delete:

**delete** [] p\_var ;

This syntax causes the compiler to call the destructor function for *each element* in the array. It does not cause ***p\_var*** to be freed *multiple times*. ***p\_var*** is still freed only once.

Example 3 : to allocate an array of integers

**int** \*p;

p = **new** **int**[5]; /\* allocate room for 5 integers \*/

**if**(!p) { **cout** << " Allocation error \n"; **return** 1; } /\*allocation check\*/

. . . . . . STATEMENTS. . . . . .

**delete** [] p;

Example 4 : to allocate *object* array named "*samp*",

**samp** \*p;

p = **new** **samp**[5]; /\* allocate object array named "samp" \*/

**if**(!p) { **cout** << " Allocation error \n"; **return** 1; } /\*allocation check\*/

. . . . . . STATEMENTS. . . . . .

**delete** [] p;

Note

1. new automatically allocates enough memory to hold an object of the specified type. You do not need to use sizeof, for example, to compute *the number of bytes* required. This reduces the possibility for error.
2. new automatically returns a pointer of the specified type. You do not need to use an explicit type cast.

* In C, no type cast is required when you are assigning the return value of ***malloc()*** to a pointer because the ***void \**** returned by ***malloc()*** is *automatically converted into a pointer* compatible with the *type of pointer* on the *left side of the assignment*.
* However, this is not the case in C++, which requires an explicit type cast when you use ***malloc()***. The reason for this difference is that *it allows C++ to enforce more rigorous type checking*.

1. Both new and delete can be overloaded, enabling you to easily implement your own custom allocation system.
2. It is possible to initialize a *dynamically allocated object*.
3. You no longer need to include <cstdlib> with your programs.

**10.12 References**

Reference a feature that is related to the pointer. A reference is an implicit pointer that for all intents and purposes acts like another name for a variable. There are three ways that a reference can be used.

1. First, a reference can be passed to a function.
2. Second, a reference can be returned by a function.
3. Finally, an independent reference can be created.

* Reference as parameter : The most important use of a reference is as a parameter to a function. Let's first start with a program that uses a pointer (not a *reference*) as a parameter:

**void** f(**int** \*n); /\* use a pointer parameter \*/

**int** **main**() { **int** i = 0;

f(&i);

**cout** << " Here is i's new value : " << i << "\n";

**return** 0;}

**void** f(**int** \*n){ \*n = 100; } /\* put 100 into the argument pointed to by n \*/

Here ***f()*** loads the value ***100*** into the integer pointed to by ***n***. In this program, ***f()*** is called with the address of ***i*** in ***main()***. Thus, after ***f()*** returns, ***i*** contains the value ***100***.

This program demonstrates how a pointer is used as a parameter to manually create a *call-by-reference parameter-passing mechanism*. In a C program, this is the *only way to achieve a call-by-reference*.

* However, in C++ reference parameter completely *automate* this process. To see how, let's rework the previous program. Here is a version that uses a reference parameter:

**void** f(**int** &n); /\* declare a reference parameter \*/

**int** **main**() { **int** i = 0;

f(i);

**cout** << " Here is i's new value : " << i << "\n";

**return** 0;}

/\* using now the reference parameter. Notice that no \* is needed in the following statement \*/

**void** f(**int** &n){ n = 100; } /\* put 100 into the argument pointed to by n \*/

* First, to declare a *reference variable or parameter*, you precede the variable's name with the &. This is how ***n*** is declared as a parameter to ***f()***.
* Now that ***n*** is a reference, it is *no longer necessary-or even legal-to apply* (not allowed) the ***\**** operator. Instead, each time ***n*** is used within ***f()***, it is automatically treated as a *pointer to the argument* used to call ***f()***. Within the function, the compiler *automatically* uses the variable pointed to by the reference parameter. This means that the statement ***n = 100;*** actually puts the value ***100*** into the variable used to call ***f()***, which in this case, is ***i***.
* Further, when ***f()*** is called, there is *no need* ( in fact *not allowed* ) to precede the argument with the &. Instead, because ***f()*** is declared as *taking a reference parameter*, the address to the argument is automatically passed to ***f()***.When you use a *reference parameter*, the compiler automatically passes the address of the variable used as the argument. Thus, a reference parameter fully automates the *call-by-reference parameter-passing mechanism*.
* You *cannot change what a reference is pointing to*. For example, if the statement ***" n++ ; "***  were put inside ***f()*** in the *preceding program*, ***"n"*** would still be *pointing to* ***i*** in ***main()***. Instead of incrementing ***n***, this statement *increments the value of the variable being referenced*  (in this case, ***i***).

Example: The classic example of passing arguments by reference is a function that *exchanges the values of the two arguments* with which it is called. Here is an example called ***swap\_args()*** that uses references *to swap its two integer arguments*:

|  |  |
| --- | --- |
| written using references | written using pointers instead of references *(recall 5.3)* |
| **void** swap\_args(**int** &x, **int** &y)  { **int** t;  t = x; x = y; y = t; } | **void** swap\_args(**int** \*x, **int** \*y)  { **int** t;  t = \*x; \*x = \*y; \*y = t; } |

* PASSING REFERENCES TO OBJECTS : As you learned in 10.10, when an object is passed to a function by use of *the default call-by-value parameter-passing mechanism*, a copy of that object is made. Although the parameter's *constructor function is not called*, its *destructor function is called* when the function returns.
* As you should recall, this can cause serious problems in some instances-when the destructor frees dynamic memory, for example. There is two way to solve this :
* One solution to this problem is to pass an object by reference.
* The other solution involves the use of copy constructors, which are discussed in Next chapter: Function overloading.
* When you pass the object by reference, *no copy is made*, and therefore its *destructor function is not called* when the function returns. Remember, however, that *changes made to the object inside the function affect the object used as the argument*.
* Note: It is critical to understand that a reference is not a pointer. Therefore, when an object is passed by reference, the *member access operator* remains the dot **(.)**, not the arrow **(->)**.
* Example: Lets first define class with constructor and destructor

#include <iostream >

using namespace std;

**class** myclass **{** **int** who ;

**public** : myclass (**int** n) { who = n;

**cout** << " Constructing " << who << "\n";}

~ myclass () { **cout** << " Destructing " << who << "\n"; }

**int** id() { **return** who; } **};**

|  |  |
| --- | --- |
| default call-by-value parameter-passing mechanism | call-by-reference parameter-passing mechanism |
| **void** f(**myclass** t){ /\* t is passed by value .\*/  **cout** << "Received" << t.id() << "\n"; }  **int** **main**(){ **myclass** x(1) ;  f(x);  **return** 0;} | **void** f(**myclass** &t){ /\* Now t is passed by reference .\*/  **cout** << "Received" << t.id() << "\n"; } /\*still "." used\*/  **int** **main**(){ **myclass** x(1) ;  f(x);  **return** 0;} |
| Output : Constructing 1  Received 1  Destructing 1  Destructing 1 /\*Two time appears \*/ | Output : Constructing 1  Received 1  Destructing 1 |
| Now we notice that the only difference between the two mechanism is "&" before t inside the function parameter . | |

* RETURNING REFERENCES : A function can return a reference. The effect of this is both powerful and startling.
* Returning a reference can be very useful when you are overloading certain types of operators. (discussed in next chapter)
* However, it also can be employed to allow a function to be used on the left side of an assignment statement.
* Example 1 : here is a very simple program that contains a function that returns a reference.

#include <iostream >

**using namespace std;**

**int** &f(); /\* *function declared that return a reference* \*/

**int** x;

/\* *without global variable returning reference is meaningless* \*/

**int** main(){ f() = 100; /\* *assign 100 to reference returned by* ***f()***\*/

**cout** << x << "\n";

**return** 0; }

**int** &f() { /\* *Return an int reference* \*/

**return** x; } /\* *returns a reference to x* \*/

* Here function ***f()*** returns a reference. So, ***f()*** is declared as *returning a reference* to an integer.
* Inside the body of the function, the statement ***" return x; "*** does not return the value of the global variable ***x***, but rather, it automatically returns ***x***'s address (int the form of a reference).
* When ***f()*** is used on the left side of the assignment statement, *it is this reference*, returned by ***f()***, that *is being assigned to*. Thus, inside ***main()***, the statement ***" f() = 100; "*** puts the value ***100*** into ***x*** because ***f()*** has returned a reference to ***x***. thus ***x*** recives the value ***100***.
* Be careful on not go out of scope. For example, here the reference returned by ***f()*** is useless:

**int** &f() { **int** x; /\* x is now a local variable \*/

**return** x; } /\* returns a reference to x \*/

In this case, ***x*** is now local to ***f()*** and will *go out of scope* when ***f()*** returns. This effectively means that the reference returned by ***f()*** is useless.

* Example 2 : One very good use of returning a reference is found when a bounded array type is created. In C++, you can create an array class that performs *automatic bounds checking*. An array class contains *two core* *functions*-one that *stores information into the array* and one that *retrieves information*. These functions can check, at run time, that the *array boundaries are not overrun*. The following program implements a bounds-checking array for characters:

// A simple bounded array example .

# include <iostream >

# include <cstdlib >

**using namespace std;**

**class** array{ **int** size ;

**char** \*p;

**public**:

**array**(**int** num); /\* constructor \*/

~**array**(){**delete** [] p;} /\* destructor \*/

**char** &put(**int** i);

**char** get(**int** i); };

array :: **array**(**int** num) /\* constructor define \*/

{p = **new** **char** [num]; /\* array declaration \*/

**if**(!p){ **cout** << "Allocation error \n";

**exit**(1) ; } /\* allocation check \*/

size = num ; } /\* array size \*/

/\* Put something into the array .\*/

char &array **::** put(**int** i) /\* ***&f()*** define \*/

{**if**(i<0 || i>=size ){

**cout** << " Bounds error !!!\ n";

**exit**(1);}

**return** p[i];} /\* return reference to ***p[i]***\*/

/\* Get something from the array.\*/

char array **::** get(**int** i)

{

**if**(i <0 || i >= size ){

**cout** << " Bounds error !!!\ n";

**exit**(1) ; }

**return** p[i];

}

**int** **main**() {

**array** a(10) ;

a.put (3) = 'X';

a.put (2) = 'R';

**cout** << a.get (3) << a.get (2) ;

**cout** << "\n";

/\* now generate run - time boundary error \*/

a.put (11) = '!';

**return** 0;

}

* Notice that the ***put()*** function returns a reference to the array element specified by parameter ***i***. This reference can then be used on the left side of an assignment statement to store something in the array-if the index specified by i is not out of bounds.
* The reverse is ***get()***, which returns the value stored at the specified index if that index is within range.
* This approach to maintaining an array is sometimes referred to as a safe array.
* The array is allocated dynamically by the use of new. This allows arrays of differing length to be declared.
* If you need to have array boundaries verified at run time, this is one way to do it. However, remember that bounds checking slows access to the array. So use it when it really needed.
* INDEPENDENT REFERENCES: Although not commonly used, the independent reference is another type of reference. An independent reference is a reference variable that in all effects is simply another name for another variable.
* Because references cannot be assigned new values, an independent reference must be initialized when it is declared.
* Example : **int** **main**(){ **int** x;

**int** &ref = x; /\* create an independent reference \*/

x = 10; /\* these two statements \*/

ref = 10; /\* are functionally equivalent \*/

ref = 100;

**cout** << x <<' '<< ref << "\n"; /\* this prints 100 twice \*/

**return** 0; }

* In this program, the independent reference ref serves as a *different name* for x. From a practical point of view, x and ref are equivalent.
* An *independent reference* can refer to a constant. For example, ***const int &ref = 10;*** is valid.
* RESTRICTIONS on using Reference : There are a number of restrictions that apply to all types of references.

1. You cannot reference another reference.
2. You cannot obtain the address of a reference.
3. You cannot create arrays of references.
4. You cannot reference a bit-field.
5. References must be initialized unless they are members of a class, are return values, or are function parameters.
6. Remember, *references are similar to pointers, but they are not pointers*.

Note

1. Reference parameters offer several advantages over their (more or less) equivalent pointer alternatives.
2. No longer need to remember to *pass the address of an argument*. When a *reference parameter* is used, the address is automatically passed.
3. Reference parameters offer a cleaner, more elegant interface than the rather clumsy *explicit pointer mechanism*.
4. When an object is passed to a function as a reference, *no copy is made*. This is one way to eliminate the *troubles associated with the copy of an argument damaging* something needed elsewhere in the program (recall change : destructor problem while passing object to a function) when its destructor function is called.