UNIT-2

Input/Output, Arrays and Working with Classes

- Basic I/O in C++
- Arrays in C++ : introduction, declaration, initialization of one , two and multi-dimensional arrays, operations on arrays
- Working with strings : introduction, declaration, string manipulation and arrays of string
- Classes and objects in C++
- Constructors : default, parameterized, copy, constructor overloading and destructor
- Access specifiers, implementing and accessing class members
- Working with objects : constant objects, nameless objects, live objects, arrays of objects

BASIC I/O in C++

The C++ standard libraries provide an extensive set of input/output capabilities. C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device likes a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device likes a display screen, a printer, a disk drive, or a network connection, etc, this is called **output operation**.

I/O Library Header Files:
There are following header files important to C++ programs:

Header File	Function and Description
<iostream>	This file defines the **cin, cout, cerr**and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
<iomanip>	This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as **setw**and **setprecision**.
<fstream>	This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter.

The standard output stream (cout):

The predefined object **cout**is an instance of **ostream**class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout**is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include <iostream>
int main( )
{
charstr[] = "Hello C++";
cout<< "Value of str is : " <<str<<endl;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of str is: Hello C++
```

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.
The insertion operator << may be used more than once in a single statement as shown above and **endl**is used to add a new-line at the end of the line.

The standard input stream (cin):
The predefined object **cin**is an instance of **istream**class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin**is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
#include <iostream>
int main( )
{
char name[50];
cout<< "Please enter your name: ";
cin>> name;
cout<< "Your name is: " << name <<endl;
}
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the result something as follows:

```
Please enter your name: cplusplus
Your name is: cplusplus
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.
The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following:

```
cin>> name >> age;
```

This will be equivalent to the following two statements:

```
Please enter your name: cplusplus
Your name is: cplusplus
```

The standard error stream (cerr):
The predefined object **cerr**is an instance of **ostream**class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr**is un-buffered and each stream insertion to cerr causes its output to appear immediately.
The **cerr**is also used in conjunction with the stream insertion operator as shown in the following example.

```
#include <iostream>
int main( )
{
charstr[] = "Unable to read....";
cerr<< "Error message : " <<str<<endl;
}
```

When the above code is compiled and executed, it produces the following result:

```
Error message : Unable to read....
```

The standard log stream (clog):
The predefined object **clog** is an instance of **ostream**class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

The **clog** is also used in conjunction with the stream insertion operator as shown in the following example.

```
#include <iostream>
int main( )
{
charstr[] = "Unable to read....";
clog<< "Error message : " <<str<<endl;
}
```

When the above code is compiled and executed, it produces the following result:

```
Error message : Unable to read....
```

You would not be able to see any difference in cout, cerr and clog with these small examples, but while writing and executing big programs then difference becomes obvious. So this is good practice to display error messages using cerr stream and while displaying other log messages then clog should be used.

C++ ARRAYS

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.
All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays:
To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

```
Type arrayName [ arraySize ];
```

This is called a single-dimension array. The **arraySize**must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement:

```
double balance[10];
```

Initializing Arrays:
You can initialize C++ array elements either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

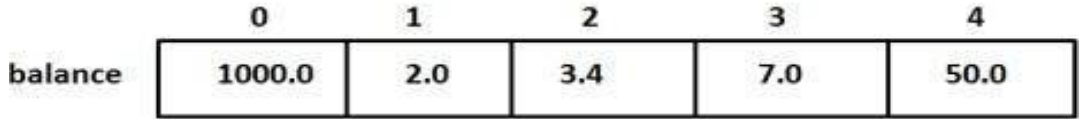
The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:



Accessing Array Elements:

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <iostream>
#include <iomanip>
usingstd::setw;
int main ()
{
int n[ 10 ]; // n is an array of 10 integers
// initialize elements of array n to 0
for ( inti = 0; i< 10; i++ )
{
n[ i ] = i + 100; // set element at location i to i + 100
}
cout<< "Element" <<setw( 13 ) << "Value" <<endl;
// output each array element's value
for ( int j = 0; j < 10; j++ )
{
cout<<setw( 7 )<< j <<setw( 13 ) << n[ j ] <<endl;
}
return 0;
}
```

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result:

```
Element Value
0 100
1 101
2 102
3 103
4 104
5 105
6 106
7 107
8 108
```

C++ Arrays in Detail:

Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer:

Concept	Description
Multi-dimensional arrays	C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
Pointer to an array	You can generate a pointer to the first element of an array by simply specifying the array name, without any index.
Passing arrays to functions	You can pass to the function a pointer to an array by specifying the array's name without an index.
Return array from functions	C++ allows a function to return an array.

C++ Multi-dimensional Arrays

C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 .10 . 4 integer array:

```
intthreedim[5][10][4];
```

Two-Dimensional Arrays:

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows:

```
typearrayName [ x ][ y ];
```

Where **type** can be any valid C++ data type and **arrayName**will be a valid C++ identifier.
A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form **a[i][j]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays:

Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {
{0, 1, 2, 3} ,/* initializers for row indexed by 0 */
{4, 5, 6, 7} ,/* initializers for row indexed by 1 */
{8, 9, 10, 11} /* initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
intval = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram.

```
#include <iostream>
int main ()
{
// an array with 5 rows and 2 columns.
int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
// output each array element's value
for ( inti = 0; i< 5; i++ )
for ( int j = 0; j < 2; j++ )
{
cout<< "a[" <<i<< "][" << j << "]: ";
cout<< a[i][j]<<endl;
}
return 0;
}
```

Output:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

C++ STRINGS

C++ provides following two types of string representations:

- The C-style character string.
- The string class type introduced with Standard C++.

The C-Style Character String:

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

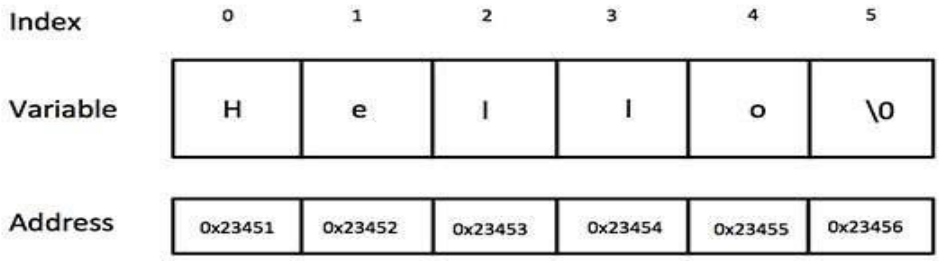
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows:

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C/C++:



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string:

```
#include <iostream>
int main ()
{
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
cout<< "Greeting message: ";
cout<< greeting <<endl;
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows

```
Greeting message: Hello
```

C++ supports a wide range of functions that manipulate null-terminated strings:

S.N.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions:

```
#include <iostream>
#include <cstring>
using namespace std;
int main ()
{
char str1[10] = "Hello";
char str2[10] = "World";
char str3[10];
intlen ;
// copy str1 into str3
strcpy( str3, str1);
cout<< "strcpy( str3, str1) : " << str3 <<endl;
// concatenates str1 and str2
strcat( str1, str2);
cout<< "strcat( str1, str2): " << str1 <<endl;
// total lenghth of str1 after concatenation
len = strlen(str1);
cout<< "strlen(str1) : " <<len<<endl;
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

The String Class in C++:

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. We will study this class in C++ Standard Library but for now let us check following example:

At this point, you may not understand this example because so far we have not discussed Classes and Objects. So can have a look and proceed until you have understanding on Object Oriented Concepts.

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
string str1 = "Hello";
string str2 = "World";
string str3;
intlen ;
// copy str1 into str3
str3 = str1;
cout<< "str3 : " << str3 <<endl;
// concatenates str1 and str2
str3 = str1 + str2;
cout<< "str1 + str2 : " << str3 <<endl;
// total lenghth of str3 after concatenation
len = str3.size();
cout<< "str3.size() : " <<len<<endl;
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() : 10
```

Array of strings in C++

Array of strings in C++ is used to store a null terminated string which is a character array. This type of array has a string with a null character at the end of the string. Usually array of strings are declared one character long to accommodate the null character.

```
#include <iostream.h>
#include <string.h>
constint DAYS =7;
constint MAX =10;
void main()
{
char week[DAYS] [MAX] = {"Sunday", "Monday", "Tuesday",
"Wednesday", "Thursday", "Friday", "Saturday" };
for(int j=0;j
{ cout<< week[j] <<endl; }
}
```

Result:

Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

In the above example a two dimensional array is used as an array of strings. The first index specifies the total elements of an array, the second index the maximum length of each string. The "MAX" value is set to "10" since the string "Wednesday" has length of "9" with a null makes it "10".

Class & Object in C++

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.
A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

C++ Class Definitions:

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows:

```
class Box
{
public:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
};
```

The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

Define C++ Objects:

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

```
Box Box1; // Declare Box1 of type Box
Box Box2; // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members:

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear:

```
#include <iostream>
using namespace std;
class Box
{
public:
double length; // Length of a box
double breadth; // Breadth of a box
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

Classes & Objects in Detail:
So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below:

Concept	Description
Class member functions	A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.
Class access modifiers	A class member can be defined as public, private or protected. By default members would be assumed as private.
Constructor & destructor	A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.
C++ copy constructor	The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.
C++ friend functions	A **friend** function is permitted full access to private and protected members of a class.
C++ inline functions	With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.
The this pointer in C++	Every object has a special pointer **this** which points to the object itself.
Pointer to C++ classes	A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.
Static members of a class	Both data members and function members of a class can be declared as static.

C++ Class member functions

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Let us take previously defined class to access the members of the class using a member function instead of directly accessing them:

```
class Box
{
public:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
doublegetVolume(void);// Returns box volume
};
```

Member functions can be defined within the class definition or separately using **scope resolution operator, ::**. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. So either you can define **Volume ()** function as below:

```
class Box
{
public:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
doublegetVolume(void)
{
return length * breadth * height;
}
};
```

If you like you can define same function outside the class using **scope resolution operator, ::** as follows:

```
double Box::getVolume(void)
{
return length * breadth * height;
}
```

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (**.**) on a object where it will manipulate data related to that object only as follows:

```
Box myBox; // Create an object
myBox.getVolume(); // Call member function for the object
```

Let us put above concepts to set and get the value of different class members in a class:

```
#include <iostream>
using namespace std;
class Box
{
public:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
// Member functions declaration
doublegetVolume(void);
voidsetLength( double len );
voidsetBreadth( double bre );
voidsetHeight( double hei );
};

// Member functions definitions
double Box::getVolume(void)
{
return length * breadth * height;
}
```

```
void Box::setLength( double len )
{
length = len;
}
void Box::setBreadth( double bre )
{
breadth = bre;
}
void Box::setHeight( double hei )
{
height = hei;
}
// Main function for the program
int main( )
{
Box Box1; // Declare Box1 of type Box
Box Box2; // Declare Box2 of type Box
double volume = 0.0; // Store the volume of a box here
// box 1 specification
Box1.setLength(6.0);
Box1.setBreadth(7.0);
Box1.setHeight(5.0);

// box 2 specification
Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);

// volume of box 1
volume = Box1.getVolume();
cout<< "Volume of Box1 : " << volume <<endl;

// volume of box 2
volume = Box2.getVolume();
cout<< "Volume of Box2 : " << volume <<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

C++ Class access modifiers

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. The access restriction to the class members is specified by the labeled **public, private,** and **protected** sections within the class body. The keywords public, private, and protected are called access specifiers.

A class can have multiple public, protected, or private labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. The default access for members and classes is private.

```
class Base {
public:
// public members go here
protected:
// protected members go here
private:
// private members go here
};
```

The public members:

A **public** member is accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function as shown in the following example:

```
#include <iostream>
using namespace std;
class Line
{
public:
double length;
voidsetLength( double len );
doublegetLength( void );
};

// Member functions definitions
double Line::getLength(void)
{
return length ;
}

void Line::setLength( double len )
{
length = len;
}
```

```
// Main function for the program
int main( )
{
Line line;
// set line length
line.setLength(6.0);
cout<< "Length of line : " <<line.getLength() <<endl;
// set line length without member function
line.length = 10.0; // OK: because length is public
cout<< "Length of line : " <<line.length<<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Length of line : 6
Length of line : 10
```

The private members:

A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.

By default all the members of a class would be private, for example in the following class **width** is a private member, which means until you label a member, it will be assumed a private member:

```
class Box
{
double width;
public:
double length;
voidsetWidth( double wid );
doublegetWidth( void );
};
```

Practically, we define data in private section and related functions in public section so that they can be called from outside of the class as shown in the following program.

```
#include <iostream>
using namespace std;
class Box
{
public:
double length;
voidsetWidth( double wid );
doublegetWidth( void );
private:
double width;
};
```

```
// Member functions definitions
double Box::getWidth(void)
{
return width ;
}
void Box::setWidth( double wid )
{
width = wid;
}
// Main function for the program
int main( )
{
Box box;
// set box length without member function
box.length = 10.0; // OK: because length is public
cout<< "Length of box : " <<box.length<<endl;
// set box width without member function
// box.width = 10.0; // Error: because width is private
box.setWidth(10.0); // Use member function to set it.
cout<< "Width of box : " <<box.getWidth() <<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Length of box : 10
Width of box : 10
```

The protected members:

A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

You will learn derived classes and inheritance in next chapter. For now you can check following example where I have derived one child class **SmallBox**from a parent class **Box**.

Following example is similar to above example and here **width** member will be accessible by any member function of its derived class SmallBox.

```
#include <iostream>
using namespace std;
class Box
{
protected:
double width;
};
classSmallBox:Box // SmallBox is the derived class.
{
public:
voidsetSmallWidth( double wid );
doublegetSmallWidth( void );
};
```

```
// Member functions of child class
doubleSmallBox::getSmallWidth(void)
{
return width ;
}
voidSmallBox::setSmallWidth( double wid )
{
width = wid;
}
// Main function for the program
int main( )
{
SmallBox box;
// set box width using member function
box.setSmallWidth(5.0);
cout<< "Width of box : "<<box.getSmallWidth() <<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Width of box: 5
```

C++ Class Constructor and Destructor

The Class Constructor:
A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.
A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.
Following example explains the concept of constructor:

```
#include <iostream>
using namespace std;
class Line
{
public:
voidsetLength( double len );
doublegetLength( void );
Line(); // This is the constructor
private:
double length;
};
// Member functions definitions including constructor
Line::Line(void)
{
cout<< "Object is being created" <<endl;
}
```

```
void Line::setLength( double len )
{
length = len;
}
double Line::getLength( void )
{
return length;
}
// Main function for the program
int main( )
{
Line line;
// set line length
line.setLength(6.0);
cout<< "Length of line : " <<line.getLength() <<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line : 6
```

Parameterized Constructor:

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example.

```
#include <iostream>
using namespace std;
class Line
{
public:
voidsetLength( double len );
doublegetLength( void );
Line(double len); // This is the constructor
private:
double length;
};
// Member functions definitions including constructor
Line::Line( double len)
{
cout<< "Object is being created, length = " <<len<<endl;
length = len;
}
void Line::setLength( double len )
{
length = len;
}
```

```
double Line::getLength( void )
{
return length;
}
// Main function for the program
int main( )
{
Line line(10.0);
// get initially set length.
cout<< "Length of line : " <<line.getLength() <<endl;
// set line length again
line.setLength(6.0);
cout<< "Length of line : " <<line.getLength() <<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

Using Initialization Lists to Initialize Fields:

In case of parameterized constructor, you can use following syntax to initialize the fields:

```
Line::Line (double len): length (len)
{
cout<< "Object is being created, length = " <<len<<endl;
}
```

Above syntax is equal to the following syntax:

```
Line::Line (double len)
{
cout<< "Object is being created, length = " <<len<<endl;
length = len;
}
```

If for a class C, you have multiple fields X, Y, Z, etc., to be initialized, and then use can use same syntax and separate the fields by comma as follows:

```
C::C (double a, double b, double c): X (a), Y (b), Z(c)
{
....
}
```

C++ Copy Constructor

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one.If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here:

```
classname (constclassname&obj)
{
// body of constructor
}
```

Here, **obj**is a reference to an object that is being used to initialize another object.

```
#include <iostream>
using namespace std;
class Line
{
public:
intgetLength( void );
Line(intlen ); // simple constructor
Line(const Line &obj); // copy constructor
~Line(); // destructor
private:
int *ptr;
};
// Member functions definitions including constructor
Line::Line(intlen)
{
cout<< "Normal constructor allocating ptr" <<endl;
// allocate memory for the pointer;
ptr = new int;
*ptr = len;
}
Line::Line(const Line &obj)
{
cout<< "Copy constructor allocating ptr." <<endl;
ptr = new int;
*ptr = *obj.ptr; // copy the value
}
Line::~Line(void)
{
cout<< "Freeing memory!" <<endl;
deleteptr;
}
```

```
int Line::getLength( void )
{
return *ptr;
}
void display(Line obj)
{
cout<< "Length of line : " <<obj.getLength() <<endl;
}
// Main function for the program
int main( )
{
Line line(10);
display(line);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
```

Let us see the same example but with a small change to create another object using existing object of the same type:

```
#include <iostream>
using namespace std;
class Line
{
public:
intgetLength( void );
Line(intlen ); // simple constructor
Line(const Line &obj); // copy constructor
~Line(); // destructor
private:
int *ptr;
};
// Member functions definitions including constructor
Line::Line(intlen)
{
cout<< "Normal constructor allocating ptr" <<endl;
// allocate memory for the pointer;
ptr = new int;
*ptr = len;
}
```

```
Line::Line(const Line &obj)
{
cout<< "Copy constructor allocating ptr." <<endl;
ptr = new int;
*ptr = *obj.ptr; // copy the value
}
Line::~Line(void)
{
cout<< "Freeing memory!" <<endl;
deleteptr;
}
int Line::getLength( void )
{
return *ptr;
}
void display(Line obj)
{
cout<< "Length of line : " <<obj.getLength() <<endl;
}
// Main function for the program
int main( )
{
Line line1(10);
Line line2 = line1; // This also calls copy constructor
display(line1);
display(line2);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Normal constructor allocating ptr
Copy constructor allocating ptr.
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Copy constructor allocating ptr.
Length of line : 10
Freeing memory!
Freeing memory!
Freeing memory!
```

The Class Destructor:

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor:

```
#include <iostream>
using namespace std;
class Line
{
public:
voidsetLength( double len );
doublegetLength( void );
Line(); // This is the constructor declaration
~Line(); // This is the destructor: declaration
private:
double length;
};
// Member functions definitions including constructor
Line::Line(void)
{
cout<< "Object is being created" <<endl;
}
Line::~Line(void)
{
cout<< "Object is being deleted" <<endl;
}
void Line::setLength( double len )
{
length = len;
}
double Line::getLength( void )
{
return length;
}
// Main function for the program
int main( )
{
Line line;
// set line length
line.setLength(6.0);
cout<< "Length of line : " <<line.getLength() <<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line : 6
Object is being deleted
```

C++ Friend Functions

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.
A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.
To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows:

```
class Box
{
double width;
public:
double length;
friend void printWidth( Box box );
voidsetWidth( double wid );
};
```

To declare all member functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne:

```
friend class ClassTwo;
```

Consider the following program:

```


#include <iostream>
using namespace std;
class Box
{
double width;
public:
friend void printWidth( Box box );
voidsetWidth( double wid );
};
// Member function definition
void Box::setWidth( double wid )
{
width = wid;
}
```

```
// Note: printWidth() is not a member function of any class.
voidprintWidth( Box box )
{
/* Because printWidth() is a friend of Box, it can
directly access any member of this class */
cout<< "Width of box : " <<box.width<<endl;
}
// Main function for the program
int main( )
{
Box box;
// set box width without member function
box.setWidth(10.0);
// Use friend function to print the wdith.
printWidth( box );
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Width of box : 10
```

C++ Inline Functions

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Following is an example, which makes use of inline function to return max of two numbers:

```
#include <iostream>
using namespace std;
inlineint Max(int x, int y)
{
return (x > y)? x : y;
}
// Main function for the program
int main( )
{
cout<< "Max (20,10): " << Max(20,10) <<endl;
cout<< "Max (0,200): " << Max(0,200) <<endl;
cout<< "Max (100,1010): " << Max(100,1010) <<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

C++ this Pointer

Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this**pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.
Friend functions do not have a **this**pointer, because friends are not members of a class. Only member functions have a **this**pointer.
Let us try the following example to understand the concept of this pointer:

```
#include <iostream>
using namespace std;
class Box
{
public:
// Constructor definition
Box(double l=2.0, double b=2.0, double h=2.0)
{
cout<<"Constructor called." <<endl;
length = l;
breadth = b;
height = h;
}
double Volume()
{
return length * breadth * height;
}
int compare(Box box)
{
return this->Volume() >box.Volume();
}
private:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
};
int main(void)
{
Box Box1(3.3, 1.2, 1.5); // Declare box1
Box Box2(8.5, 6.0, 2.0); // Declare box2
if(Box1.compare(Box2))
{
cout<< "Box2 is smaller than Box1" <<endl;
}
```

```
else
{
cout<< "Box2 is equal to or larger than Box1" <<endl;
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```

Static members of a C++ class

We can define class members static using static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.
Let us try the following example to understand the concept of static data members:

```
#include <iostream>
using namespace std;
class Box
{
public:
staticintobjectCount;
// Constructor definition
Box(double l=2.0, double b=2.0, double h=2.0)
{
cout<<"Constructor called." <<endl;
length = l;
breadth = b;
height = h;
// Increase every time object is created
objectCount++;
}
double Volume()
{
return length * breadth * height;
}
private:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
};
```

```
// Initialize static member of class Box
int Box::objectCount = 0;
int main(void)
{
Box Box1(3.3, 1.2, 1.5); // Declare box1
Box Box2(8.5, 6.0, 2.0); // Declare box2
// Print total number of objects.
cout<< "Total objects: " << Box::objectCount<<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Constructor called.
Constructor called.
Total objects: 2
```

Static Function Members:

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this**pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Let us try the following example to understand the concept of static function members:

```
#include <iostream>
using namespace std;
class Box
{
public:
staticintobjectCount;
// Constructor definition
Box(double l=2.0, double b=2.0, double h=2.0)
{
cout<<"Constructor called." <<endl;
length = l;
breadth = b;
height = h;
// Increase every time object is created
objectCount++;
}
double Volume()
{
return length * breadth * height;
}
```

```
staticintgetCount()
{
returnobjectCount;
}
private:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
};
// Initialize static member of class Box
int Box::objectCount = 0;
int main(void)
{
// Print total number of objects before creating object.
cout<< "Inital Stage Count: " << Box::getCount() <<endl;
Box Box1(3.3, 1.2, 1.5); // Declare box1
Box Box2(8.5, 6.0, 2.0); // Declare box2
// Print total number of objects after creating object.
cout<< "Final Stage Count: " << Box::getCount() <<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

Arrayof objects in C++

Arrays of variables of type “class” are known as "**Array of objects**". The "identifier" used to refer the array of objects is a user defined data type.

```
#include <iostream.h>
constint MAX =100;
class Details
{
private:
int salary;
float roll;
public:
voidgetname( )
{
cout<< "\n Enter the Salary:";
cin>> salary;
cout<< "\n Enter the roll:";
cin>> roll;
}
```

```
voidputname( )
{
cout<< "Employees" << salary << "and roll is" << roll << '\n'; }
};
void main()
{
Details det[MAX];
int n=0;
charans;
do
{
cout<< "Enter the Employee Number::" << n+1;
det[n++].getname; cout<< "Enter another (y/n):? " ;
cin>>ans;
}
while ( ans != 'n' );
for (int j=0; j<n; j++)
{
cout<< "\nEmployee Number is:: " << j+1; det[j].putname( );
}
}
```

Output is

```
Enter the Employee Number:: 1
Enter the Salary:20
Enter the roll:30
Enter another (y/n)?: y
Enter the Employee Number:: 2
Enter the Salary:20
Enter the roll:30
Enter another (y/n)?: n
```

In the above example an array of object "det" is defined using the user defined data type "Details". The class element "getname()" is used to get the input that is stored in this array of objects and putname() is used to display the information.

Constant Object

Like member functions and member function arguments, the objects of a class can also be declared as **const**. an object declared as const cannot be modified and hence, can invoke only const member functions as these functions ensure not to modify the object.
A const object can be created by prefixing the const keyword to the object declaration. Any attempt to change the data member of const objects results in a compile-time error.

Syntax:

```
constClass_NameObject_name;
```

ct as

- Whenever an object is declared as const, it needs to be initialized at the time of declaration. however, the object initialization while declaring is possible only with the help of constructors.

A function becomes const when the const keyword is used in the function's declaration. The idea of const functions is not to allow them to modify the object on which they are called. It is recommended the practice to make as many functions const as possible so that accidental changes to objects are avoided.

```
#include <iostream>
using namespace std;
class Test
{
int value;
public:
Test(int v = 0)
{ value = v; }
intgetValue() const
{ return value; }
};
int main()
{
                  Test t(20);
                  cout<<t.getValue();
                  return 0;
}
```

Output is

```
20
```

Nameless Object

Anonymous class is a class which has no name given to it. C++ supports this feature.

- These classes cannot have a constructor but can have a destructor.
- These classes can neither be passed as arguments to functions nor can be used as return values from functions.

An **anonymous object** is essentially a value that has no name. Because they have no name, there's no way to refer to them beyond the point where they are created. Consequently, they have "expression scope", meaning they are created, evaluated, and destroyed all within a single expression.

Example

Anonymous class is created with object name obj1. The scope of the obj1 is throughout the program. So, we can access this into the main function. In main, using obj1, a call is given to member functions of the anonymous class.

```
// CPP program to illustrate
// concept of Anonymous Class
#include <iostream>
using namespace std;
// Anonymous Class : Class is not having any name
class
{
// data member
inti;
public:
voidsetData(inti)
{
// this pointer is used to differentiate
// between data member and formal argument.
this->i = i;
}
void print()
{
cout<< "Value for i : " << this->i<<endl;
}
} obj1;

// object for anonymous class
// Driver function
int main()
{
obj1.setData(10);
obj1.print();
return 0;
}
```

Live Object

Objects created dynamically with their data members initialized during creation are known as Live Objects. To create a live object, constructor must be invoked automatically which performs initialization of data members. Similarly the destructor for an object must be invoked automatically before the memory for that object is deallocated.

A class whose live object is to be crated must have atleast one constructor. The syntax for creating a live object is as follows.

- Dynamic initialization of object refers to initializing the objects at a run time i.e., the initial value of an object is provided during run time.

- It can be achieved by using constructors and by passing parameters to the constructors.
- This comes in really handy when there are multiple constructors of the same class with different inputs.

```
Pointer_to_Object = new Class_name(Parameters)
```

Example:

```
// C++ program for dynamic allocation
#include <iostream>
using namespace std;
class geeks {
                  int* ptr;
public:
// Default constructor
geeks()
{
// Dynamically initializing ptrusing new
                      ptr = new int;
                      *ptr = 10;
                  }
                  // Function to display the valueof ptr
                  void display()
                  {
                      cout<< *ptr<<endl;
                  }
};
// Driver Code
int main()
{
                  geeks obj1;
                  // FunctionCall
                  obj1.display();
                  return 0;
}
```

Output

```
10
```

Assignment Questions:
1. Explain the methods of defining class members with an example.
2. How many types of constructors we can create in C++ programs and also write few characteristics of constructor.
3. Explain declaration and initialization of one dimensional array with example.
4. What is private and public mode in class?