OBJECT ORIENTED PROGRAMMING USING C++ (CS2105ES)

UNIT-I

The programming paradigm is the way of writing computer programs. There are four programming paradigms and they are as follows.

- **Monolithic programming paradigm**
- **Structured-oriented programming paradigm**
- **Procedural-oriented programming paradigm**
- **Object-oriented programming paradigm**

Monolithic Programming Paradigm

The Monolithic programming paradigm is the oldest. It has the following characteristics. It is also known as the imperative programming paradigm.

- In this programming paradigm, the whole program is written in a single block.
- We use the **goto** statement to jump from one statement to another statement.
- It uses all data as global data which leads to data insecurity.
- There are no flow control statements like if, switch, for, and while statements in this paradigm.
- There is no concept of data types.

An **example** of a Monolithic programming paradigm is **Assembly language**.

Structure-oriented Programming Paradigm

The Structure-oriented programming paradigm is the advanced paradigm of the monolithic paradigm. It has the following characteristics.

- This paradigm introduces a modular programming concept where a larger program is divided into smaller modules.
- It provides the concept of code reusability.
- It is introduced with the concept of data types.
- It also provides flow control statements that provide more control to the user.
- In this paradigm, all the data is used as global data which leads to data insecurity.

Examples of a structured-oriented programming paradigm is **ALGOL, Pascal, PL/I and Ada**.

Procedure-oriented Programming Paradigm

The procedure-oriented programming paradigm is the advanced paradigm of a structure-oriented paradigm. It has the following characteristics.

- This paradigm introduces a modular programming concept where a larger program is divided into smaller modules.
- It provides the concept of code reusability.
- It is introduced with the concept of data types.
- It also provides flow control statements that provide more control to the user.

- It follows all the concepts of structure-oriented programming paradigm but the data is defined as global data, and also local data to the individual modules.
- In this paradigm, functions may transform data from one form to another.

Examples of procedure-oriented programming paradigm is **C, visual basic, FORTRAN**, etc.

Object-oriented Programming Paradigm

The object-oriented programming paradigm is the most popular. It has the following characteristics.

- In this paradigm, the whole program is created on the concept of objects.
- In this paradigm, objects may communicate with each other through function.
- This paradigm mainly focuses on data rather than functionality.
- In this paradigm, programs are divided into what are known as objects.
- It follows the bottom-up flow of execution.
- It introduces concepts like data abstraction, inheritance, and overloading of functions and operators overloading.
- In this paradigm, data is hidden and cannot be accessed by an external function.
- It has the concept of friend functions and virtual functions.
- In this paradigm, everything belongs to objects.

Examples of procedure-oriented programming paradigm is **C++, Java, C#, Python**, etc

C++ OOPs Concepts

The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.

Object Oriented Programming is a paradigm that provides many concepts such as **inheritance, data binding, polymorphism etc.**

The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object

- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base and Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −

class derived-class: access-specifier base-class

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows −

```
#include <iostream>

using namespace std;

// Base class
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};

// Derived class
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);
    }
};

int main(void) {
  Rectangle Rect;

  Rect.setWidth(5);
  Rect.setHeight(7);

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;

  return 0;
}
```

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

```
Total area: 35
```

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way −

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions −

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Type of Inheritance

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied −

- **Public Inheritance** − When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Multiple Inheritance

A C++ class can inherit members from more than one class and here is the extended syntax −

class derived-class: access baseA, access baseB....

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example −

```
#include <iostream>

using namespace std;

// Base class Shape
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};

// Base class PaintCost
class PaintCost {
  public:
    int getCost(int area) {
      return area * 70;
    }
};

// Derived class
class Rectangle: public Shape, public PaintCost {
  public:
    int getArea() {
      return (width * height);
    }
};

int main(void) {
  Rectangle Rect;
  int area;

  Rect.setWidth(5);
  Rect.setHeight(7);

  area = Rect.getArea();

  // Print the area of the object.
```

```
   cout << "Total area: " << Rect.getArea() << endl;

   // Print the total cost of painting
   cout << "Total paint cost: $" << Rect.getCost(area) << endl;

   return 0;
}
```

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

```
Total area: 35
Total paint cost: $2450
```

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In C++, we use Function overloading and Function overriding to achieve polymorphism.

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes −

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


class Shape {
   protected:
      int width, height;


   public:
      Shape( int a = 0, int b = 0){
         width = a;
         height = b;
      }
```

```
    int area() {
      cout << "Parent class area :" << width * height << endl;
      return width * height;
    }
};
class Rectangle: public Shape {
  public:
    Rectangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
      cout << "Rectangle class area :" << width * height << endl;
      return (width * height);
    }
};

class Triangle: public Shape {
  public:
    Triangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
      cout << "Triangle class area :" << (width * height)/2 << endl;
      return (width * height / 2);
    }
};

// Main function for the program
int main() {
  Shape *shape;
  Rectangle rec(10,7);
  Triangle  tri(10,5);
  // store the address of Rectangle
  shape = &rec;
  // call rectangle area.
  shape->area();
  // store the address of Triangle
  shape = &tri;
  // call triangle area.
  shape->area();
  return 0;
}
```

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

```
Parent class area :70
Parent class area :50
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this −

```
#include  <iostream>
using namespace std;
class Shape {
  protected:
    int width, height;
  public:
    Shape( int a = 0, int b = 0){
      width = a;
      height = b;
    }
    virtual int area() {
      cout << "Parent class area :" << width * height << endl;
```

```
         return width * height;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }
      int area () {
         cout << "Rectangle class area :" << width * height << endl;
         return (width * height);
      }
};
class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Triangle class area :" << (width * height)/2 << endl;
         return (width * height / 2);
      }
};
// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;
   // call rectangle area.
   shape->area();
   // store the address of Triangle
   shape = &tri;
   // call triangle area.
   shape->area();
   return 0;
}
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result −

```
Rectangle class area :70
Triangle class area :25
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Abstraction:

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.

In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this −

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

int main() {
  cout << "Hello C++" <<endl;
  return 0;
}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of ‘cout’ is free to change.

Access Labels Enforce Abstraction

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels −

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction

Data abstraction provides two important advantages −

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

Data Abstraction Example

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example −

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

class Adder {
  public:
    // constructor
    Adder(int i = 0) {
      total = i;
    }

    // interface to outside world
    void addNum(int number) {
      total += number;
    }

    // interface to outside world
    int getTotal() {
      return total;
    };

  private:
    // hidden data from outside world
    int total;
};

int main()
 {
  Adder a;
    a.addNum(10);
  a.addNum(20);
  a.addNum(30);

  cout << "Total " << a.getTotal() <<endl;
  return 0;
}
```

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

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members - **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

All C++ programs are composed of the following two fundamental elements −

- **Program statements (code)** − This is the part of a program that performs actions and they are called functions.
- **Program data** − The data is the information of the program which gets affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private. For example –

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

  private:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};
```

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

Data Encapsulation Example

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example −

```
#include  <iostream>
using namespace std;
class Adder {
  public:
    // constructor
    Adder(int i = 0) {
      total = i;
    }
        // interface to outside world
    void addNum(int number) {
      total += number;
    }

    // interface to outside world
    int getTotal() {
      return total;
    };
    private:
    // hidden data from outside world
    int total;
};

int main() {
  Adder a;

  a.addNum(10);
  a.addNum(20);
  a.addNum(30);

  cout << "Total " << a.getTotal() <<endl;
  return 0;
}
```

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

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

Advantage of OOPs over Procedure-oriented programming language:

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.

3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

Primitive Built-in Types

C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types −

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these type modifiers −

- signed
- unsigned
- short
- long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range

char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	8bytes	-2,147,483,648 to 2,147,483,647
signed long int	8bytes	same as long int
unsigned long int	8bytes	0 to 4,294,967,295
long long int	8bytes	-(2^63) to (2^63)-1
unsigned long long int	8bytes	0 to 18,446,744,073,709,551,615
float	4bytes	
double	8bytes	

long double	12bytes	
wchar_t	2 or 4 bytes	1 wide character

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

Following is the example, which will produce correct size of various data types on your computer.

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

int main() {
   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;
   cout << "Size of short int : " << sizeof(short int) << endl;
   cout << "Size of long int : " << sizeof(long int) << endl;
   cout << "Size of float : " << sizeof(float) << endl;
   cout << "Size of double : " << sizeof(double) << endl;
   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;

   return 0;
}
```

This example uses **endl**, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using **sizeof()** operator to get size of various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine −

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 4
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

typedef Declarations

You can create a new name for an existing type using **typedef**. Following is the simple syntax to define a new type using typedef −

```
typedef type newname;
```

For example, the following tells the compiler that feet is another name for int −

typedef int feet;

Now, the following declaration is perfectly legal and creates an integer variable called distance −

feet distance;

Enumerated Types

An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. Each enumerator is a constant whose type is the enumeration.

Creating an enumeration requires the use of the keyword **enum**. The general form of an enumeration type is −

enum enum-name { list of names } var-list;

Here, the enum-name is the enumeration's type name. The list of names is comma separated.

For example, the following code defines an enumeration of colors called colors and the variable c of type color. Finally, c is assigned the value "blue".

```
enum color { red, green, blue } c;
c = blue;
```

By default, the value of the first name is 0, the second name has the value 1, and the third has the value 2, and so on. But you can give a name, a specific value by adding an initializer. For example, in the following enumeration, **green** will have the value 5.

enum color { red, green = 5, blue };

Here, **blue** will have a value of 6 because each name will be one greater than the one that precedes it.

Variable Declaration:

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive −

There are following basic types of variable in C++ as explained in last chapter −

Sr.No	Type & Description
1	**Bool** Stores either value true or false.

2	**char** Typically a single octet (one byte). This is an integer type.
3	**int** The most natural size of integer for the machine.
4	**float** A single-precision floating point value.
5	**double** A double-precision floating point value.
6	**void** Represents the absence of type.
7	**wchar_t** A wide character type.

C++ also allows to define various other types of variables, which we will cover in subsequent chapters like **Enumeration, Pointer, Array, Reference, Data structures,** and **Classes**.

Following section will cover how to define, declare and use various types of variables.

Variable Definition in C++

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type, and contains a list of one or more variables of that type as follows −

```
type variable_list;
```

Here, **type** must be a valid C++ data type including char, w_char, int, float, double, bool or any user-defined object, etc., and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

```
int    i, j, k;
char c, ch;
float  f, salary;
double d;
```

The line **int i, j, k;** both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows −

```
type variable_name = value;
```

Some examples are −

```
extern int d = 3, f = 5;    // declaration of d and f.
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined.

Variable Declaration in C++

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable definition at the time of linking of the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use **extern** keyword to declare a variable at any place. Though you can declare a variable multiple times in your C++ program, but it can be defined only once in a file, a function or a block of code.

Example

Try the following example where a variable has been declared at the top, but it has been defined inside the main function −

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

// Variable declaration:
extern int a, b;
extern int c;
extern float f;

int main () {
  // Variable definition:
  int a, b;
  int c;
  float f;

  // actual initialization
  a = 10;
  b = 20;
  c = a + b;

  cout << c << endl ;

  f = 70.0/3.0;
  cout << f << endl ;

  return 0;
}
```

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

```
30
23.3333
```

Same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example −

```
// function declaration
int func();
int main() {
  // function call
  int i = func();
}

// function definition
int func() {
  return 0;
}
```

Lvalues and Rvalues

There are two kinds of expressions in C++ −

- **lvalue** − Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.
- **rvalue** − The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement −

Declaration of Expressions in C++

C++ expression **consists of operators, constants, and variables which are arranged according to the rules of the language**. It can also contain function calls which return values. An expression can consist of one or more operands, zero or more operators to compute a value.

Declaration of operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

There are following arithmetic operators supported by C++ language −

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0

++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.

<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

There are following logical operators supported by C++ language.

Assume variable A holds 1 and variable B holds 0, then −

Show Examples

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows −

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1

1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

Show Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the	A << 2 will give 240 which is 1111

	number of bits specified by the right operand.	0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators

There are following assignment operators supported by C++ language −

Show Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left	C /= A is equivalent to C = C / A

	operand.	
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Misc Operators

The following table lists some other operators that C++ supports.

Sr.No	Operator & Description
1	**sizeof** sizeof operator returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4.
2	**Condition ? X : Y** Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.
3	**,** Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-

	separated list.
4	**. (dot) and -> (arrow)** Member operators are used to reference individual members of classes, structures, and unions.
5	**Cast** Casting operators convert one data type to another. For example, int(2.2000) would return 2.
6	**&** Pointer operator & returns the address of a variable. For example &a; will give actual address of the variable.
7	***** Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

Operators Precedence in C++

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Show Examples

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left

Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

C++ Expression Evaluation

In the C++ programming language, an expression is evaluated based on the operator precedence and associativity. When there are multiple operators in an expression, they are evaluated according to their precedence and associativity. The operator with higher precedence is evaluated first and the operator with the least precedence is evaluated last.

To understand expression evaluation in c, let us consider the following simple example expression.

Type Conversion:

Type conversion is the process of converting a data value from one data type to another data type.

In the C++ programming language, the data conversion is performed in two different methods and they are as follows.

- Implicit Conversion (Type Conversion)
- Explicit Conversion (Type Casting)

Implicit Conversion (Type Conversion)

The type conversion is the process of converting a data value from one data type to another data type automatically by the compiler. Sometimes type conversion is also called **implicit type conversion**. The implicit type conversion is automatically performed by the compiler. For example, in c programming language, when we assign an integer value to a float variable the integer value automatically gets converted to float value by adding decimal value 0. And when a float value is assigned to an integer variable the float value automatically gets converted to an integer value by removing the decimal value. To understand more about type conversion observe the following...

int i = 10 ;
float x = 15.5 ;
char ch = 'A' ;

i = x ; =======> x value 15.5 is converted as 15 and assigned to variable i

x = i ; =======> Here i value 10 is converted as 10.000000 and assigned to variable x

i = ch ; =======> Here the ASCII value of A (65) is assigned to i

```
#include <iostream>
using namespace std;
int main()
{
   int i = 95 ;
   float f = 90.99 ;
   char ch = 'A' ;
   i = f ;     //float to int --> 90.99 to 90
   cout << "i value is " << i << endl;
   f = i ;     // int to float --> 90 to 90.000000
```

```
    cout << "f value is " << f << endl;
    i = ch ; // char to int --> 'A' to 65
    cout << "i value is " << i << endl;


    return 0;
```

Explicit Conversion (Type Casting)

Typecasting is also called an **explicit type conversion**. Compiler converts data from one data type to another data type implicitly. When compiler converts implicitly, there may be a data loss. In such a case, we convert the data from one data type to another data type using explicit type conversion. To perform this we use the **unary cast operator**. To convert data from one type to another type we specify the target data type in parenthesis as a prefix to the data value that has to be converted. The general syntax of typecasting is as follows.

Example

int totalMarks = 450, maxMarks = 600 ;
float average ;

average = (float) totalMarks / maxMarks * 100 ;

In the above example code, both totalMarks and maxMarks are integer data values. When we perform totalMarks / maxMarks the result is a float value, but the destination (average) datatype is a float. So we use type casting to convert totalMarks and maxMarks into float data type.

```
#include <iostream>


using namespace std;


int main()
{
    int a, b, c ;
    float avg ;
    cout << "Enter any three integer values : ";
    cin >> a >> b >> c;
    avg = (a + b + c) / 3 ;
    cout << "avg before type casting = " << avg << endl;
```

```
   avg = (float)(a + b + c) / 3 ;
   cout << "avg after type casting = " << avg << endl;
   return 0;
```

Pointers:

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined −

```
#include  <iostream>
using namespace std;
int main () {
  int var1;
  char var2[10];
  cout << "Address of var1 variable: ";
  cout << &var1 << endl;
  cout << "Address of var2 variable: ";
  cout << &var2 << endl;
  return 0;
}
```

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

```
Address of var1 variable: 0xbfebd5c0
Address of var2 variable: 0xbfebd5b6
```

What are Pointers?

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is −

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration −

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Using Pointers in C++

There are few important operations, which we will do with the pointers very frequently. **(a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations −

```
#include <iostream>

using namespace std;

int main () {
   int var = 20;  // actual variable declaration.
   int  *ip;        // pointer variable
   ip = &var;       // store address of var in pointer variable
   cout << "Value of var variable: ";
   cout << var << endl;
   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;
   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;
   return 0;
}
```

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

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

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 { } can not 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[] = {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 representaion 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>
using namespace std;

#include <iomanip>
using std::setw;
```

```
int main () {

  int n[ 10 ]; // n is an array of 10 integers

  // initialize elements of array n to 0
  for ( int i = 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
      9        109
```

Arrays and Pointers:

It is most likely that you would not understand this chapter until you go through the chapter related C++ Pointers.

So assuming you have bit understanding on pointers in C++, let us start: An array name is a constant pointer to the first element of the array. Therefore, in the declaration −

double balance[50];

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns **p** the address of the first element of **balance** −

```
double *p;
double balance[10];
```

p = balance;

It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].

Once you store the address of first element in p, you can access array elements using *p, *(p+1), *(p+2) and so on. Below is the example to show all the concepts discussed above −

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

int main () {
   // an array with 5 elements.
   double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
   double  *p;
   p = balance;
    // output each array element's value
   cout << "Array values using pointer " << endl;
    for ( int i = 0; i < 5; i++ ) {
      cout << "*(p + " << i << ") : ";
      cout << *(p + i) << endl;
   }
   cout << "Array values using balance as address " << endl;
      for ( int i = 0; i < 5; i++ ) {
      cout << "*(balance + " << i << ") : ";
      cout << *(balance + i) << endl;
   }
    return 0;
}
```

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

```
Array values using pointer
*(p + 0) : 1000
*(p + 1) : 2
*(p + 2) : 3.4
*(p + 3) : 17
*(p + 4) : 50
Array values using balance as address
*(balance + 0) : 1000
*(balance + 1) : 2
*(balance + 2) : 3.4
*(balance + 3) : 17
*(balance + 4) : 50
```

In the above example, p is a pointer to double which means it can store address of a variable of double type. Once we have address in p, then ***p** will give us value available at the address stored in p, as we have shown in the above example.

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>

using namespace std;

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 the following result −

```
Greeting message: Hello
```

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

Sr.No	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];
   int len ;
   // 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. Let us check the following example −

```
#include <iostream>
#include <string>

using namespace std;

int main () {
   string str1 = "Hello";
   string str2 = "World";
   string str3;
   int len ;
   // copy str1 into str3
   str3 = str1;
   cout << "str3 : " << str3 << endl;
   // concatenates str1 and str2
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;
   // total length 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
```

Reference:

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

References vs Pointers

References are often confused with pointers but three major differences between references and pointers are −

- You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
- Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

Creating References in C++

Think of a variable name as a label attached to the variable's location in memory. You can then think of a reference as a second label attached to that memory location. Therefore, you can access the contents of the variable through either the original variable name or the reference. For example, suppose we have the following example −

int i = 17;

We can declare reference variables for i as follows.

int& r = i;

Read the & in these declarations as **reference**. Thus, read the first declaration as "r is an integer reference initialized to i" and read the second declaration as "s is a double reference initialized to d.". Following example makes use of references on int and double −

```
#include  <iostream>

using namespace std;

int main () {
   // declare simple variables
   int    i;
   double d;
    // declare reference variables
   int&    r = i;
   double& s = d;
      i = 5;
   cout << "Value of i : " << i << endl;
   cout << "Value of i reference : " << r << endl;
    d = 11.7;
   cout << "Value of d : " << d << endl;
   cout << "Value of d reference : " << s << endl;
      return 0;
}
```

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

```
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7
```

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages −

C++ programming language provides the following type of loops to handle looping requirements.

Sr.No	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop Like a ‘while’ statement, except that it tests the condition at the end of the loop body.
4	nested loops You can use one or more loop inside any another ‘while’, ‘for’ or ‘do..while’ loop.

Loop Control Statements(flow)

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C++ supports the following control statements.

Sr.No	Control Statement & Description
1	break statement Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	continue statement Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	goto statement Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

The Infinite Loop

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the ‘for’ loop are required, you can make an endless loop by leaving the conditional expression empty.

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

int main () {
  for( ; ; ) {
    printf("This loop will run forever.\n");
  }

  return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the ‘for (;;)’ construct to signify an infinite loop.

NOTE − You can terminate an infinite loop by pressing Ctrl + C keys.

Functions

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

Defining a Function

The general form of a C++ function definition is as follows −

```
return_type function_name( parameter list ) {
  body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and return the biggest of both −

```
// function returning the max between two numbers

int max(int num1, int num2) {
   // local variable declaration
```

```
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration −

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration −

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it’s return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example −

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

// function declaration
int max(int num1, int num2);

int main () {
  // local variable declaration:
  int a = 100;
  int b = 200;
  int ret;
  // calling a function to get max value.
  ret = max(a, b);
  cout << "Max value is : " << ret << endl;
   return 0;
}

// function returning the max between two numbers
int max(int num1, int num2) {
  // local variable declaration
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result −

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function –

Sr.No	Call Type & Description
1	**Call by Value** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

2	**Call by Pointer** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	**Call by Reference** This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example −

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

int sum(int a, int b = 20) {
   int result;
   result = a + b;

   return (result);
}
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   int result;

   // calling a function to add the values.
   result = sum(a, b);
   cout << "Total value is :" << result << endl;

   // calling a function again as follows.
   result = sum(a);
```

```
  cout << "Total value is :" << result << endl;

  return 0;
}
```

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

```
Total value is :300
Total value is :120
```

Inline function:

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;

inline int 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
```

Recusive function:

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

Example

```
int sum(int k) {
  if (k > 0) {
    return k + sum(k - 1);
  } else {
    return 0;
  }
}

int main() {
  int result = sum(10);
  cout << result;
  return 0;
}
```

pointers to functions:

C++ allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function −

```
#include <iostream>
```

```
#include <ctime>

using namespace std;
void getSeconds(unsigned long *par);

int main () {
   unsigned long sec;
   getSeconds( &sec );

   // print the actual value
   cout << "Number of seconds :" << sec << endl;

   return 0;
}

void getSeconds(unsigned long *par) {
   // get the current number of seconds
   *par = time( NULL );

   return;
}
```

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

```
Number of seconds :1294450468
```

The function which can accept a pointer, can also accept an array as shown in the following example −

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

// function declaration:
double getAverage(int *arr, int size);

int main () {
   // an int array with 5 elements.
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   // pass pointer to the array as an argument.
   avg = getAverage( balance, 5 ) ;

   // output the returned value
   cout << "Average value is: " << avg << endl;

   return 0;
}
```

```
double getAverage(int *arr, int size) {
  int i, sum = 0;
  double avg;

  for (i = 0; i < size; ++i) {
    sum += arr[i];
  }
  avg = double(sum) / size;

  return avg;
}
```

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

```
Average value is: 214.4
```

Dynamic memory allocation:

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts −

- **The stack** − All variables declared inside the function will take up memory from the stack.
- **The heap** − This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

New and delete Operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

new data-type;

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new** operator with the following statements −

```
double* pvalue = NULL; // Pointer initialized with null
pvalue  = new double;  // Request memory for the variable
```

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below −

```
double* pvalue  = NULL;
if( !(pvalue  = new double )) {
  cout << "Error: out of memory." <<endl;
  exit(1);
}
```

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the 'delete' operator as follows −

```
delete pvalue;        // Release memory pointed to by pvalue
```

Let us put above concepts and form the following example to show how 'new' and 'delete' work −

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

int main () {
  double* pvalue = NULL; // Pointer initialized with null
  pvalue  = new double;  // Request memory for the variable

  *pvalue = 29494.99;     // Store value at allocated address
  cout << "Value of pvalue : " << *pvalue << endl;

  delete pvalue;         // free up the memory.

  return 0;
}
```

If we compile and run above code, this would produce the following result −

```
Value of pvalue : 29495
```

Dynamic Memory Allocation for Arrays

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

```
char* pvalue  = NULL;         // Pointer initialized with null
pvalue  = new char[20];       // Request memory for the variable
```

To remove the array that we have just created the statement would look like this −

```
delete [] pvalue;            // Delete array pointed to by pvalue
```

Following the similar generic syntax of new operator, you can allocate for a multi-dimensional array as follows −

```
double** pvalue  = NULL;      // Pointer initialized with null
pvalue  = new double [3][4];  // Allocate memory for a 3x4 array
```

However, the syntax to release the memory for multi-dimensional array will still remain same as above −

```
delete [] pvalue;            // Delete array pointed to by pvalue
```

Dynamic Memory Allocation for Objects

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept −

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

class Box {
  public:
     Box() {
       cout << "Constructor called!" <<endl;
     }
     ~Box() {
       cout << "Destructor called!" <<endl;
     }
};
int main() {
  Box* myBoxArray = new Box[4];
  delete [] myBoxArray; // Delete array

  return 0;
}
```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result −

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```

Preprocessor directives: The preprocessors are the directives, which give instructions to the compiler to preprocess the information before actual compilation starts.

All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;).

You already have seen a **#include** directive in all the examples. This macro is used to include a header file into the source file.

There are number of preprocessor directives supported by C++ like #include, #define, #if, #else, #line, etc. Let us see important directives −

The #define Preprocessor

The #define preprocessor directive creates symbolic constants. The symbolic constant is called a **macro** and the general form of the directive is −

```
#define macro-name replacement-text
```

When this line appears in a file, all subsequent occurrences of macro in that file will be replaced by replacement-text before the program is compiled. For example −

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

#define PI 3.14159

int main () {
  cout << "Value of PI :" << PI << endl;

  return 0;
}
```

Now, let us do the preprocessing of this code to see the result assuming we have the source code file. So let us compile it with -E option and redirect the result to test.p. Now, if you check test.p, it will have lots of information and at the bottom, you will find the value replaced as follows −

```
$gcc -E test.cpp > test.p

...
int main () {
  cout << "Value of PI :" << 3.14159 << endl;
  return 0;
}
```

Function-Like Macros

You can use #define to define a macro which will take argument as follows −

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

```
#define MIN(a,b) (((a)<(b)) ? a : b)

int main () {
   int i, j;

   i = 100;
   j = 30;

   cout <<"The minimum is " << MIN(i, j) << endl;

   return 0;
}
```

If we compile and run above code, this would produce the following result −

```
The minimum is 30
```

Conditional Compilation

There are several directives, which can be used to compile selective portions of your program's source code. This process is called conditional compilation.

The conditional preprocessor construct is much like the ‘if’ selection structure. Consider the following preprocessor code −

```
#ifndef NULL
   #define NULL 0
#endif
```

You can compile a program for debugging purpose. You can also turn on or off the debugging using a single macro as follows −

```
#ifdef DEBUG
   cerr <<"Variable x = " << x << endl;
#endif
```

This causes the **cerr** statement to be compiled in the program if the symbolic constant DEBUG has been defined before directive #ifdef DEBUG. You can use #if 0 statment to comment out a portion of the program as follows −

```
#if 0
   code prevented from compiling
#endif
```

Let us try the following example −

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

#define MIN(a,b) (((a)<(b)) ? a : b)
```

```
int main () {
   int i, j;

   i = 100;
   j = 30;

#ifdef DEBUG
   cerr <<"Trace: Inside main function" << endl;
#endif

#if 0
   /* This is commented part */
   cout << MKSTR(HELLO C++) << endl;
#endif

   cout <<"The minimum is " << MIN(i, j) << endl;

#ifdef DEBUG
   cerr <<"Trace: Coming out of main function" << endl;
#endif

   return 0;
}
```

If we compile and run above code, this would produce the following result −

```
The minimum is 30
Trace: Inside main function
Trace: Coming out of main function
```

The # and ## Operators

The # and ## preprocessor operators are available in C++ and ANSI/ISO C. The # operator causes a replacement-text token to be converted to a string surrounded by quotes.

Consider the following macro definition −

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

#define MKSTR( x ) #x

int main () {

   cout << MKSTR(HELLO C++) << endl;

   return 0;
}
```

If we compile and run above code, this would produce the following result −

```
HELLO C++
```

Let us see how it worked. It is simple to understand that the C++ preprocessor turns the line −

```
cout << MKSTR(HELLO C++) << endl;
```

Above line will be turned into the following line −

```
cout << "HELLO C++" << endl;
```

The ## operator is used to concatenate two tokens. Here is an example −

```
#define CONCAT( x, y ) x ## y
```

When CONCAT appears in the program, its arguments are concatenated and used to replace the macro. For example, CONCAT(HELLO, C++) is replaced by "HELLO C++" in the program as follows.

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

#define concat(a, b) a ## b
int main() {
  int xy = 100;

  cout << concat(x, y);
  return 0;
}
```

If we compile and run above code, this would produce the following result −

```
100
```

Let us see how it worked. It is simple to understand that the C++ preprocessor transforms −

```
cout << concat(x, y);
```

Above line will be transformed into the following line −

```
cout << xy;
```

Predefined C++ Macros

C++ provides a number of predefined macros mentioned below −

Sr.No	Macro & Description
1	**__LINE__** This contains the current line number of the program when it is being compiled.

2	**—FILE__** This contains the current file name of the program when it is being compiled.
3	**—DATE__** This contains a string of the form month/day/year that is the date of the translation of the source file into object code.
4	**—TIME__** This contains a string of the form hour:minute:second that is the time at which the program was compiled.

Let us see an example for all the above macros −

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

int main () {
  cout << "Value of__LINE__: " << __LINE__<< endl;
  cout << "Value of__FILE__: " << __FILE__<< endl;
  cout << "Value of__DATE__: " << __DATE__<< endl;
  cout << "Value of__TIME__: " << __TIME__<< endl;

  return 0;
}
```

If we compile and run above code, this would produce the following result −

```
Value of__LINE__: 6
Value of__FILE__: test.cpp
Value of__DATE__: Feb 28 2011
Value of__TIME__: 18:52:48
```

Unit-2

C++ Classes/Objects

C++ is an object-oriented programming language.

Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an **object**. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

Attributes and methods are basically **variables** and **functions** that belongs to the class. These are often referred to as "class members".

A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the class keyword:

Example

Create a class called "MyClass":

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};
```

Example explained

- The class keyword is used to create a class called MyClass.
- The public keyword is an access specifier, which specifies that members (attributes and methods) of the class are accessible from outside the class. You will learn more about access specifiers later.
- Inside the class, there is an integer variable myNum and a string variable myString. When variables are declared within a class, they are called **attributes**.
- At last, end the class definition with a semicolon ;.

Create an Object

In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.

To create an object of MyClass, specify the class name, followed by the object name.

To access the class attributes (myNum and myString), use the dot syntax (.) on the object:

Example

Create an object called "myObj" and access the attributes:

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};
int main() {
  MyClass myObj;  // Create an object of MyClass

  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";
  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;

}
```

Friends to a class

No one shares secrets with strangers for privacy reasons. But everyone has that one friend with whom they share most of their secrets, if not all. Friend class in C++ refers to the same concept. Public data members and functions are accessible by every class in C++ and many other programming languages. But the C++ friend class is special and can access even the private data members and functions of other classes.

Friend Keyword in C++

Now, you know the benefits of having a friend class in C++. But, to declare any class as a friend class, you do it with the friend keyword. You can use the friend keyword to any class to declare it as a friend class. This keyword enables any class to access private and protected members of other classes and functions.

Use of Friend Class in C++

Friend class has numerous uses and benefits. Some of the primary use cases include:

- Accessing private and protected members of other classes (as you would know by now)
- Declaring all the functions of a class as friend functions

- Allowing to extend storage and access its part while maintaining encapsulation
- Enabling classes to share private members' information
- Widely used where two or more classes have interrelated data members

Syntax of Implementing Friend Class in C++

The syntax for implementing a friend class is:

```
class ClassB;

class ClassA {

  // ClassB is a friend class of ClassA

  friend class ClassB;

  ... .. ...

}

class ClassB {

  ... .. ...

}
```

As you can see in the syntax, all you need to do is use the keyword friend in front of a class to make it a friend class. Using this syntax will make ClassB a friend class of ClassA. Since ClassB becomes the friend class, it will have access to all the public, private, and protected members of ClassA. However, the opposite will not be true. That’s because C++ only allows granting the friend relation and not taking it. Hence, ClassA will not have access to private members of ClassB.

Understanding The Friend Class in C++ with Example Code
By SimplilearnLast updated on Sep 18, 202114555

Table of Contents

No one shares secrets with strangers for privacy reasons. But everyone has that one friend with whom they share most of their secrets, if not all. Friend class in C++ refers to the same concept. Public data members and functions are accessible by every class in C++ and many other programming languages. But the C++ friend class is special and can access even the private data members and functions of other classes.

Friend Keyword in C++

Now, you know the benefits of having a friend class in C++. But, to declare any class as a friend class, you do it with the friend keyword. You can use the friend keyword to any class to declare it as a friend class. This keyword enables any class to access private and protected members of other classes and functions.

Use of Friend Class in C++

Friend class has numerous uses and benefits. Some of the primary use cases include:

- Accessing private and protected members of other classes (as you would know by now)
- Declaring all the functions of a class as friend functions
- Allowing to extend storage and access its part while maintaining encapsulation
- Enabling classes to share private members' information
- Widely used where two or more classes have interrelated data members

Syntax of Implementing Friend Class in C++

The syntax for implementing a friend class is:

```
class ClassB;

class ClassA {

  // ClassB is a friend class of ClassA

  friend class ClassB;  ... .. ...

}

class ClassB {

  ... .. ...

}
```

As you can see in the syntax, all you need to do is use the keyword friend in front of a class to make it a friend class. Using this syntax will make ClassB a friend class of ClassA. Since ClassB becomes the friend class, it will have access to all the public, private, and protected members of ClassA. However, the opposite will not be true. That's because C++ only allows granting the friend relation and not taking it. Hence, ClassA will not have access to private members of ClassB.

Examples of Friend Class in C++

Now that you have seen the syntax to implement a friend class in C++ and its use, look at some examples to understand the concept better.

Example 1: A Simple Example to Access Private Members of Other Class

```
#include <iostream>

using namespace std;

class Exmp_A{

    int i =3;

    // Declaring the friend class

    friend class Exmp_B;

};

class Exmp_B

{

  public:

    void display(Exmp_A &a)

    {

        cout<<"The value of i is : "<<a.i;

    }

};
```

```
int main(){

    Exmp_A a;

    Exmp_B b;

    b.display(a);

    return 0;

}
```

Output:

```
The value of i is 3
```

Implementing Friend Function in C++ Through a Method of Another Class

```
#include  <iostream>

using namespace std;

class Exmp_A{

    private:

    int A_value;

    public:

    // Default Constructor

    Exmp_A(){

        A_value = 20;

    }
    friend class Exmp_B; // Friend Class

};

class Exmp_B{

    private:

    int B_value;

    public:

    // Accessing private members

    void display(Exmp_A& i) {

        cout<<"The private member's value accessed using friend class is: " << i.A_value<<endl;

    }

};

int main(){

cout<<"Welcome to Simplilearn!"<<endl<<endl;

Exmp_A A_value;

Exmp_B B_value;

B_value.display(A_value);

return 0;

}
```

Output:

```
Welcome to Simplilearn!
```

```
The private member's  value accessed friend class is: 20
```

Implementing Friend Function in C++ Through a Global Function

```
#include  <iostream>

using namespace std;

class Example{

    // A private member by default

    string s;

public:

        friend void show( Example value );

    void input( string val );

};

void Example::input( string val ){

    s = val;

}

void show( Example value ){

    cout<<"Value of private string data is : "<<value.s<<endl;

}

int main(){
    cout<<"Welcome to Simplilearn!"<<endl<<endl;

    Example value;

    value.input("Simplilearn");

    // Using friend function to display string

    show( value );

    return 0;

}
```

Static class members

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:
    static int objectCount;

    // 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:
    static int objectCount;

    // 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;
    }
    static int getCount() {
      return objectCount;
    }

  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;
}
```

Output:

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

Final Stage Count: 2

Constant member functions

The const member functions are the functions which are declared as constant in the program. The object called by these functions cannot be modified. It is recommended to use const keyword so that accidental changes to object are avoided.

A const member function can be called by any type of object. Non-const functions can be called by non-const objects only.

Here is the syntax of const member function in C++ language,

datatype function_name const();

Here is an example of const member function in C++,

Example

```
#include<iostream>
using namespace std;
class Demo {
  int val;
  public:
  Demo(int x = 0) {
    val = x;
  }
  int getValue() const {
    return val;
  }
};
int main() {
  const Demo d(28);
  Demo d1(8);
  cout << "The value using object d : " << d.getValue();
  cout << "\nThe value using object d1 : " << d1.getValue();
  return 0;
}
```

Output

```
The value using object d : 28
The value using object d1 : 8
```

Constructors and Destructors:

Providing the initial value as described in the earlier chapters of C++ does not conform to the philosophy of C++. So C++ provides a special member function called the constructor which enables an object to initialize itself at the time of its creation. This is known as automatic initialization of objects. This concept of C++ also provides another member function called destructor which is used to destroy the objects when they are no longer required. In this chapter, you will learn about how constructors and destructors work, types of constructors and how they can be implemented within C++ program.

The process of creating and deleting objects in C++ is a vital task. Each time an instance of a class is created the constructor method is called. Constructors is a special member function of class and it is used to initialize the objects of its class. It is treated as a special member function because its name is the same as the class name. These constructors get invoked whenever an object of its associated class is created. It is named as "constructor" because it constructs the value of data member of a class. Initial values can be passed as arguments to the constructor function when the object is declared.

This can be done in two ways:

- By calling constructor explicitly
- By calling constructor implicitly

The declaration and definition of constructor is as follows

syntax:

```
class class_name
{
 int g, h;
 public:
 class_name(void);  // Constructor Declared
 . . .
 };
class_name :: class_name()
{
 g=1; h=2;  // Constructor defined
}
```

Special characteristics of Constructors:

- They should be declared in the public section
- They do not have any return type, not even void
- They get automatically invoked when the objects are created
- They cannot be inherited though derived class can call the base class constructor
- Like other functions, they can have default arguments
- You cannot refer to their address

- Constructors cannot be virtual

C++ offers four types of constructors. These are:

1. Do nothing constructor
2. Default constructor
3. Parameterized constructor
4. Copy constructor

Do nothing Constructor

Do nothing constructors are that type of constructor which does not contain any statements. Do nothing constructor is the one which has no argument in it and no return type.

Default Constructor

The default constructor is the constructor which doesn't take any argument. It has no parameter but a programmer can write some initialization statement there.

Syntax:

```
class_name()
{
 // Constructor Definition ;
}
```

```
//Code Snippet:
#include <iostream>
using namespace std;

class Calc {
 int val;

public:
 Calc()
 {
 val = 20;
 }
};
int main()
{
 Calc c1;
 cout << c1.val;
}
```

A default constructor is very important for initializing object members, that even if we do not define a constructor explicitly, the compiler automatically provides a default constructor implicitly.

Parameterized Constructor

A default constructor does not have any parameter, but programmers can add and use parameters within a constructor if required. This helps programmers to assign initial values to an object at the time of creation.

Example:

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

 class Calc
 {
  int val2;
  public:
  Calc(int x)
  {
   val2=x;
 }
};

 int main()
 {
  Calc c1(10);
  Calc c2(20);
  Calc c3(30);
  cout << c1.val2;
  cout << c2.val2;
  cout << c3.val2;
}
```

Copy Constructor

C++ provides a special type of constructor which takes an object as an argument and is used to copy values of data members of one object into another object. In this case, copy constructors are used to declaring and initializing an object from another object.

Example:

```
Calc C2(C1);
Or
Calc C2 = C1;
```

The process of initializing through a copy constructor is called the copy initialization.

Syntax:

```
class-name (class-name &)
{
 . . .
}
```

Example:

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

class CopyCon {
 int a, b;

public:
 CopyCon(int x, int y)
 {
 a = x;
 b = y;
 cout << "\nHere is the initialization of Constructor";
 }
 void Display()
 {
 cout << "\nValues : \t" << a << "\t" << b;
 }
};

void main()
{
 CopyCon Object(30, 40);
 //Copy Constructor
 CopyCon Object2 = Object;
 Object.Display();
 Object2.Display();
}
```

What are Destructors?

As the name implies, destructors are used to destroy the objects that have been created by the constructor within the C++ program. Destructor names are same as the class name but they are preceded by a tilde (~). It is a good practice to declare the destructor after the end of using constructor. Here's the basic declaration procedure of a destructor:

destructor neither takes an argument nor returns any value and the compiler implicitly invokes upon the exit from the program for cleaning up storage that is no longer accessible.

The base class members can be accessed by its sub-classes through access specifiers. There are three types of access specifies. They are public, private and protected.

1. Public

When the base class is publicly inherited, the public members of the base class become the derived class public members.

Also read : Discuss the different parameter passing techniques

They can be accessed by the objects of the derived class.

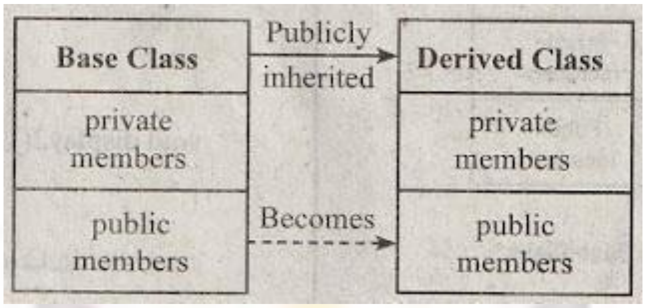


Figure: Publicly Inherited Base Class

Syntax

class classname

{

public:

datatype variablename;

returntype functionname();

};

2. Protected

When the base class is derived in protected mode, the 'protected' and 'public' members of the base class become the protected members of the derived class.

Also read :Write a C++ program to find out GCD of given two numbers using function

Private and protected members of a class can be accessed by,
(i) A friend function of the class.
(ii) A member function of the friend class.
(iii) A derived class member function.

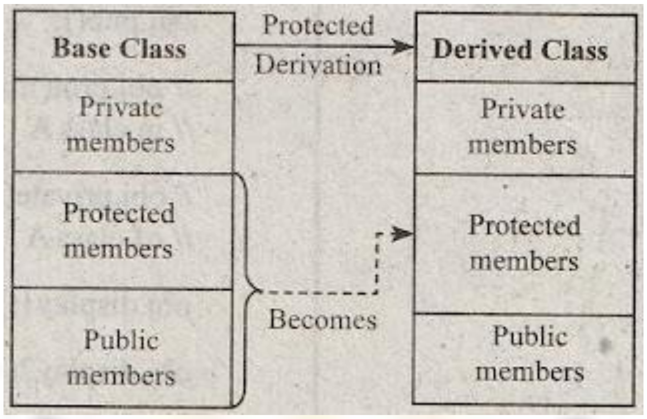


Figure: Protected Derivation of the Base Class

3. Private

When a base class contains members, that are declared as private, they cannot be accessed by the derived class objects. They can be accessed only by the class in which they are defined.

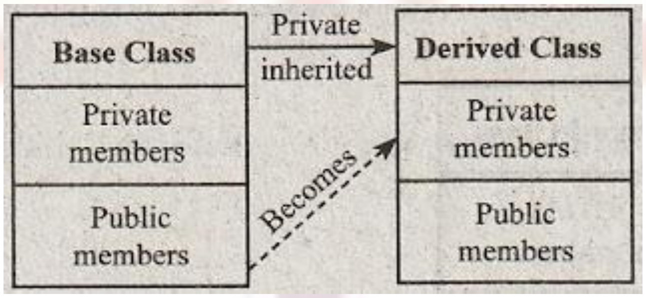


Figure: Privately inherited Base Class

Syntax

Also read :Explain the virtual functions call mechanism

```
class classname
{
private:
datatype variablename;
returntype functionname( );
};
```

Program

```
#include  <iostream>
using namespace std;
class A
{
public: pub()
{
cout << “In Public() \n”;
}
protected: prot()
{
cout << “In Protected() \n”;
}
private: priv()
{
cout << “In Pivate() \n”;
}
};
class C : public A
{
public:
void display1()
{
cout<< “In C::display1 call\n”;
pub();
}
void display2()
{
cout << “In C::display2 call\n”;
prot();
}
/*pri(Q is a private member of class A. Therefore it is an illegal access
void display3()
{
cout<<“In C::display3 call\n”;
priv();
} */
}
main()
{
C obj;
obj.pub();
// obj.prot(); illegal because it is declared as protected in class A
// obj.private(); illegal because pri() isa private member of class A
obj.display1();
obj.display2();
}
```

Constructor and Destructor in Derived class in C++

We know that constructor is invoked implicitly when an object is created. But do you think what will happen when we create object of derived class?

1. In inheritance, When an object of derived class is created then constructor of derived class get executed and then it calls the constructor of base class.

2. If there is no default constructor present in parent class then not only we have to create constructor in child class but also we will have to call the constructor of parent class.

3. In order to call parameterized constructor of parent class, we need to create a constructor in child class and also we will have to call parameterized constructor with the help of child class constructor

Syntax of calling parent class constructor using child class constructor

```
class A
{
 public:
  A() {
   //constructor body
     } };
  class B: public A {
  public:
   B(): A() {
   //constructor body
  }
 };
```

For Example:-

```
class A
{
  int a;
public:
  A (int k)          //parameterized constructor of parent class.
  {
    a = k;
  }
};
class B: public A
{
  int b;
  public:
  B(int x, int y):A(x)     //constructor of child class calling constructor of base class.
  {
    b = y;
  }
};
int main()
{
  B obj(2,3);
}
```

Points to Remember:-

1. In inheritance, the order of constructors calling is: from *child* class to *parent* class (*child* -> *parent*).

2. In inheritance, the order of constructors execution is: from *parent* class to *child* class (*parent* -> *class*).

3. In inheritance, the order of destructors calling is: from *child* class to *parent* class (*child* -> *parent*).

4. In inheritance, the order of destructors execution is: from *child* class to *parent* class (*child* **->** *parent*).

For Example:- Destructor Example

```
#include<iostream>

using namespace std;

class baseClass

{

public:

  baseClass()

  {

    cout << "I am baseClass constructor" << endl;

  }

  ~baseClass()

  {

    cout << "I am baseClass destructor" << endl;

  }

};


class derivedClass: public baseClass

{

public:

  derivedClass()

  {
    cout << "I am derivedClass constructor" << endl;

  }



  ~derivedClass()

  {

    cout <<" I am derivedClass destructor" << endl;

  }

};

int main()

{

  derivedClass D;

  return 0;

}
```

Data abstraction

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.

In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this −

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

int main() {
  cout << "Hello C++" <<endl;
  return 0;
}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of ‘cout’ is free to change.

Access Labels Enforce Abstraction

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels −

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction

Data abstraction provides two important advantages −

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

Data Abstraction Example

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example −

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

class Adder {
  public:
    // constructor
    Adder(int i = 0) {
      total = i;
    }

    // interface to outside world
    void addNum(int number) {
      total += number;
    }

    // interface to outside world
    int getTotal() {
      return total;
    };

  private:
    // hidden data from outside world
    int total;
};

int main() {
  Adder a;

  a.addNum(10);
  a.addNum(20);
  a.addNum(30);

  cout << "Total " << a.getTotal() <<endl;
  return 0;
}
```

Output:

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members - **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.

Abstract Data Types in C++

Now that we've seen the concept of abstract data types (ADTs), we proceed to examine the mechanisms C++ provides for defining an ADT. Unlike C, C++ allows the data and functions of an ADT to be defined together. It also enables an ADT to prevent access to internal implementation details, as well as to guarantee that an object is appropriately initialized when it is created.

Our convention in this course is to use the word *struct* to refer to C-style ADTs, as well as to use the struct keyword to define them. We use the word *class* to refer to C++ ADTs and use the class keyword to define them. We will discuss the technical difference between the two keywords momentarily.

A C++ class includes both member variables, which define the data representation, as well as member *functions* that operate on the data. The following is a Triangle class in the C++ style:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in);

  double perimeter() const {
    return this->a + this->b + this->c;
  }

  void scale(double s) {
    this->a *= s;
    this->b *= s;
    this->c *= s;
  }
};
```

The class has member variables for the length of each side, defining the data representation. We defer discussion of the `public:` and `Triangle(...)` lines for now. Below those lines are member functions for computing the perimeter of a triangle and scaling it by a factor.

The following is an example of creating and using a Triangle object:

```
int main() {
  Triangle t1(3, 4, 5);
  t1.scale(2);
  cout << t1.perimeter() << endl;
}
```

We initialize a triangle by passing in the side lengths as part of its declaration. We can then scale a triangle by using the same dot syntax we saw for accessing a member variable: <object>.<function>(<arguments>) .

Before we discuss the details of what the code is doing, let us compare elements of the C-style definition and use of the triangle ADT with the C++ version. The following contrasts the definition of an ADT function between the two styles:

C-Style Struct	C++ Class

void Triangle_scale(Triangle *tri,
double s) {
tri->a *= s;
tri->b *= s;
tri->c *= s;
}
	class Triangle {
	void scale(double s) {
	this->a *= s;
	this->b *= s;
	this->c *= s;
	}
	};

The following compares how objects are created and manipulated:

C-Style Struct	C++ Class

Triangle t1;
Triangle_init(&t1, 3, 4, 5);
Triangle_scale(&t1, 2);
	Triangle t1(3, 4, 5);
	
	t1.scale(2);

With the C-style struct, we defined a top-level Triangle_scale() function whose first argument is a pointer to the Triangle object we want to scale. With a C++ class, on the other hand, we define a scale() member function within the Triangle class itself. There is no need to prepend Triangle_ , since it is clear that scale() is a member of the Triangle class. The member function also does not explicitly declare a pointer parameter – instead, the C++ language adds an *implicit* this parameter that is a pointer to the Triangle object we are working on. We can then use the this pointer in the same way we used the explicit tri pointer in the C style.

As for using a `Triangle` object, in the C style, we had to separately create the `Triangle` object and then initialize it with a call to `Triangle_init()`. In the C++ style, object creation and initialization are combined – we will see how later. When invoking an ADT function, in the C case we have to explicitly pass the address of the object we are working on. With the C++ syntax, the object is part of the syntax – it appears on the left-hand side of the dot, so the compiler automatically passes its address as the `this` pointer of the `scale()` member function.

Member Accessibility

The data representation of an ADT is usually an implementation detail (plain old data being an exception). With C-style structs, however, we have to rely on programmers to respect convention and avoid accessing member variables directly. With C++ classes, the language provides us a mechanism for enforcing this convention: declaring members as *private* prevents access from outside the class, while declaring them as *public* allows outside access. [1] We give a set of members a particular access level by placing `private:` or `public:` before the members – that access level applies to subsequent members until a new access specifier is encountered, and any number of specifiers may appear in a class. The following is an example:

[1] Later, we will discuss *protected* members that are accessible to derived classes. This access level, however, is rarely used.

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in);

  double perimeter() const {
    return a + b + c;
  }

  void scale(double s) {
    a *= s;
    b *= s;
    c *= s;
  }
};
```

In this example, the members a , b, and c are declared as private, while Triangle() , perimeter() , and scale() are declared as public. Private members, whether variables or functions, can be accessed from within the class, even if they are members of a different object of that class. They cannot be accessed from outside the class:

```
int main() {
  Triangle t1(3, 4, 5);           // OK: Triangle() is public
  t1.scale(2);                    // OK: scale() is public
  cout << t1.perimeter() << endl; // OK: perimeter() is public

  // Die triangle! DIE!
  t1.a = -1;                      // ERROR: a is private
}
```

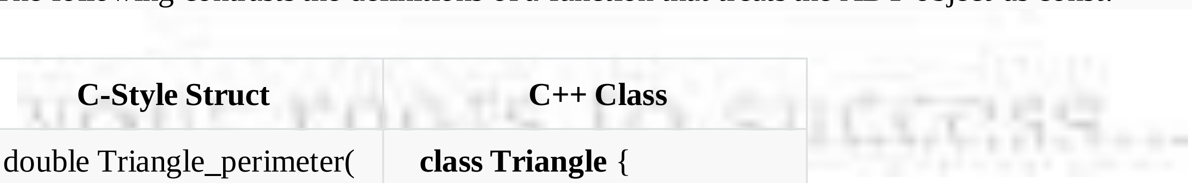
With the class keyword, the default access level is private. Thus, the private: at the beginning of the Triangle definition is redundant, and the following is equivalent:

```
class Triangle {
  double a;
  double b;
  double c;

public:
  Triangle(double a_in, double b_in, double c_in);

  ...
};
```

We have seen previously that members declared within a struct are accessible from outside the struct. In fact, the **only** difference between the struct and class keywords when defining a class type is the default access level: public for struct but private for class . 2 However, we use the two keywords for different conventions in this course.

Figure 43 Memory layout when scaling a triangle in the C and C++ styles.

The following contrasts the definitions of a function that treats the ADT object as const:



C-Style Struct	**C++ Class**

double Triangle_perimeter(
        const Triangle *tri) {
  return tri->a +
        tri->b +
	class Triangle {
	  double perimeter() **const** {
	    return this->a +
	        this->b +

C-Style Struct	C++ Class

tri->c;
}
	this->c;
	}
	};

In the C style, we add the `const` keyword to the left of the `*` when declaring the explicit pointer parameter, resulting in `tri` being a pointer to const. In the C++ style, we don't have an explicit parameter where we can add the `const` keyword. Instead, we place the keyword after the parameter list for the function. The compiler will then make the implicit `this` parameter a pointer to const, as if it were declared with the type `const Triangle *`. This allows us to call the member function on a const `Triangle` :

```
const Triangle t1(3, 4, 5);
cout << t1.perimeter() << endl;  // OK: this pointer is a pointer to const
t1.scale(2);                     // ERROR: conversion from const to non-const
```

As with accessing member variables, we can use the arrow operator to invoke a member function through a pointer:

```
Triangle t1(3, 4, 5);
const Triangle *ptr = &t1;
cout << ptr->perimeter() << endl;  // OK: this pointer is a pointer to const
ptr->scale(2);                     // ERROR: conversion from const to non-const
```

Implicit `this->`

Since member variables and member functions are both located within the scope of a class, C++ allows us to refer to members from within a member function without the explicit `this->` syntax. The compiler automatically inserts the member dereference for us:

```
class Triangle {
  double a;
  double b;
  double c;

  ...

  double perimeter() const {
    return a + b + c; // Equivalent to: this->a + this->b + this->c
  }
};
```

This is also the case for invoking other member functions. For instance, the following defines and uses functions to get each side length:

```
class Triangle {
  double a;
  double b;
  double c;

  ...

  double side1() const {
    return a;
  }

  double side2() const {
    return b;
  }

  double side3() const {
    return c;
  }

  double perimeter() const {
    return side1() + side2() + side3();
    // Equivalent to: this->side1() + this->side2() + this->side3()
  }
};
```

In both cases, the compiler can tell that we are referring to members of the class and therefore inserts the `this->`. However, if there are names in a closer scope that conflict with the member names, we must use `this->` ourselves. The following is an example:

```
class Triangle {
  double a;

  ...

  double set_side1(double a) {
    this->a = a;
  }
};
```

Here, the *unqualified* `a` refers to the parameter `a`, since it is declared in a narrower scope than the member variable. We can still refer to the member `a` by *qualifying* its name with `this->`.

In general, we should avoid declaring variables in a local scope that *hide* names in an outer scope. Doing so in a constructor or set function is often considered acceptable, but it should be avoided elsewhere.

Information hiding:

In simple words, data hiding is an object-oriented programming technique of hiding internal object details i.e. data members. Data hiding guarantees restricted data access to class members & maintain object integrity. In this blog, we will understand how data hiding works in C++. Following topics are covered in this tutorial:

- Encapsulation
- Abstraction
- Data Hiding

Encapsulation, abstraction & data hiding is closely related to each other. When we talk about any C++ program, it consists of two fundamental elements:

- **Program statements** – Part of the program (functions) that performs actions.
- **Program data** – Data in the program which is manipulated using functions.

Encapsulation

Encapsulation binds the data & functions together which keeps both safe from outside interference. Data encapsulation led to data hiding.

Let’s look at an example of encapsulation. Here, we are specifying the getter & setter function to get & set the value of variable *num* without accessing it directly.

Example:

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

class Encapsulation
{
private:
// data hidden from outside world
int num;

public:
// function to set value of
// variable x
void set(int a)
{
num =a;
}

// function to return value of
// variable x
int get()
{
return num;
}
};

// main function
int  main()
{
Encapsulation obj;

obj.set(5);

cout<<obj.get();
return 0;
}
```

Output:

Data Abstraction

Data Abstraction is a mechanism of hiding the implementation from the user & exposing the interface.

Example:

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

class Abstraction
{
private:
    int num1, num2;

public:

    void set(int a, int b)
    {
        num1 = a;
        num2 = b;
    }

void display()
{
cout<<"num1 = " <<num1 << endl;
cout<<"num2 = " << num2 << endl;
}
};

int main()
{
Abstraction obj;
obj.set(50, 100);
obj.display();
return 0;
}
```

Output:

Data Hiding in C++

Data hiding is a process of combining data and functions into a single unit. The ideology behind data hiding is to conceal data within a class, to prevent its direct access from outside the class. It helps programmers to create classes with unique data sets and functions, avoiding unnecessary penetration from other program classes.

Discussing data hiding & data encapsulation, data hiding only hides class data components, whereas data encapsulation hides class data parts and private methods.

Now you also need to know access specifier for understanding data hiding.

private, public & protected are three types of protection/ access specifiers available within a class. Usually, the data within a class is private & the functions are public. The data is hidden, so that it will be safe from accidental manipulation.

- **Private members/methods** can only be accessed by methods defined as part of the class. Data is most often defined as private to prevent direct outside access from other classes. Private members can be accessed by members of the class.

- **Public members/methods** can be accessed from anywhere in the program. Class methods are usually public which is used to manipulate the data present in the class. As a general rule, data should not be declared public. Public members can be accessed by members and objects of the class.
- **Protected member/methods** are private within a class and are available for private access in the derived class.

Now let’s look at a data hiding example.

Example: Data Hiding in C++

```
#include<iostream>
using namespace std;
class Base{

int num; //by default private
public:

void read();
void print();

};

void Base :: read(){
cout<<"Enter any Integer value"<<endl; cin>>num;

}

void Base :: print(){
cout<<"The value is "<<num<<endl;
}

int main(){
  Base obj;

  obj.read();
  obj.print();

  return 0;
}
```

Unit-3

Defining a class hierarchy:

Hierarchical Inheritance in C++ refers to the type of inheritance that has a hierarchical structure of classes. A single base class can have multiple derived classes, and other subclasses can further inherit these derived classes, forming a hierarchy of classes. The following diagram illustrates the structure of Hierarchical Inheritance in C++.

Now, understand Hierarchical Inheritance in C++ with the help of an example. There are 3 major branches derived from modern science. They are- Physics, Chemistry, and Biology. These 3 branches are further divided into sub-branches, which are further classified into other specialized disciplines. Here, Modern Science is the base class which is further inherited by 3 subclasses- Physics, Chemistry, and Biology. And these subclasses are further inherited by other derived classes, structuring into a hierarchy of classes.

Use of Hierarchical Inheritance in C++

Hierarchical Inheritance in C++ is useful in the cases where a hierarchy has to be maintained. Most of the schools and colleges maintain the data of their students in hierarchical form. For example, a college has to maintain the data of the engineering students and segregate them according to their branches such as the IT branch, mechanical branch, and so on. You can achieve such a scenario can by Hierarchical Inheritance in C++ easily. Similar is the case with the companies where they have to maintain the data of their employees according to the different departments.

Syntax to Implement Hierarchical Inheritance in C++

You can use the following syntax to achieve Hierarchical Inheritance in C++:

```
class base_class

{

   //data members

   //member functions

};

class derived_class1 : visibility_mode base_class
```

```
{

    //data members

    //member functions

};

class derived_class2 : visibility_mode base_class

{

    //data members

    //member functions

};
```

Different forms of Inheritance

Inheritance is one of four pillars of Object-Oriented Programming (OOPs). It is a feature that enables a class to acquire properties and characteristics of another class. Inheritance allows you to reuse your code since the derived class or the child class can reuse the members of the base class by inheriting them. Consider a real-life example to clearly understand the concept of inheritance. A child inherits some properties from his/her parents, such as the ability to speak, walk, eat, and so on. But these properties are not especially inherited in his parents only. His parents inherit these properties from another class called mammals. This mammal class again derives these characteristics from the animal class. Inheritance works in the same manner.

During inheritance, the data members of the base class get copied in the derived class and can be accessed depending upon the visibility mode used. The order of the accessibility is always in a decreasing order i.e., from public to protected. There are mainly five types of Inheritance in C++ that you will explore in this article. They are as follows:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

What Are Child and Parent classes?

To clearly understand the concept of Inheritance, you must learn about two terms on which the whole concept of inheritance is based - Child class and Parent class.

- Child class: The class that inherits the characteristics of another class is known as the child class or derived class. The number of child classes that can be inherited from a single parent class is based upon the type of inheritance. A child class will access the data members of the parent class according to the visibility mode specified during the declaration of the child class.

- Parent class: The class from which the child class inherits its properties is called the parent class or base class. A single parent class can derive multiple child classes (Hierarchical Inheritance) or multiple parent classes can inherit a single base class (Multiple Inheritance). This depends on the different types of inheritance in C++.

The syntax for defining the child class and parent class in all types of Inheritance in C++ is given below:

```
class parent_class

{

   //class definition of the parent class

};

class child_class : visibility_mode parent_class

{

  //class definition of the child class

};
```

Syntax Description

- parent_class: Name of the base class or the parent class.

- child_class: Name of the derived class or the child class.

- visibility_mode: Type of the visibility mode (i.e., private, protected, and public) that specifies how the data members of the child class inherit from the parent class.

Why and When to Use Inheritance?

Inheritance makes the programming more efficient and is used because of the benefits it provides. The most important usages of inheritance are discussed below:

1. Code reusability: One of the main reasons to use inheritance is that you can reuse the code. For example, consider a group of animals as separate classes - Tiger, Lion, and Panther. For these classes, you can create member functions like the predator() as they all are predators, canine() as they all have canine teeth to hunt, and claws() as all the three animals have big and sharp claws. Now, since all the three functions are the same for these classes, making separate functions for all of them will cause data redundancy and can increase the chances of error. So instead of this, you can use inheritance here. You can create a base class named carnivores and add these functions to it and inherit these functions to the tiger, lion, and panther classes.
2. Transitive nature: Inheritance is also used because of its transitive nature. For example, you have a derived class mammal that inherits its properties from the base class animal. Now, because of the transitive nature of the inheritance, all the child classes of 'mammal' will inherit the properties of the class 'animal' as well. This helps in debugging to a great extent. You can remove the bugs from your base class and all the inherited classes will automatically get debugged.

Types of inheritance in C++

There are five types of inheritance in C++ based upon how the derived class inherits its features from the base class. These five types are as follows:

- ### Single Inheritance

Single Inheritance is the most primitive among all the types of inheritance in C++. In this inheritance, a single class inherits the properties of a base class. All the data members of the base class are accessed by the derived class according to the visibility mode (i.e., private, protected, and public) that is specified during the inheritance.

Syntax

```
class base_class_1

{    // class definition

};

class derived_class: visibility_mode base_class_1

{    // class definition

};
```

Description

A single derived_class inherits a single base_class. The visibility_mode is specified while declaring the derived class to specify the control of base class members within the derived class.

Example

The following example illustrates Single Inheritance in C++:

```
#include <iostream>

using namespace std;

// base class

class electronicDevice

{

public:

   // constructor of the base class

   electronicDevice()

   {
        cout << "I am an electronic device.\n\n";

    }

};

 // derived class

class Computer: public electronicDevice

{

public:

    // constructor of the derived class

    Computer()

    {

        cout << "I am a computer.\n\n";

    }

};

int main()

{

    // create object of the derived class

    Computer obj; // constructor of base class and

                  // derived class will be called

    return 0;

}
```

In the above example, the subclass Computer inherits the base class electronicDevice in a public mode. So, all the public and protected member functions and data members of the class electronicDevice are directly accessible to the class Computer. Since there is a single derived class inheriting a single base class, this is Single Inheritance.

- **Multiple Inheritance**

The inheritance in which a class can inherit or derive the characteristics of multiple classes, or a derived class can have over one base class, is known as Multiple Inheritance. It specifies access specifiers separately for all the base classes at the time of inheritance. The derived class can derive the joint features of all these classes and the data members of all the base classes are accessed by the derived or child class according to the access specifiers.

Syntax

```
class base_class_1

{

   // class definition

};

class base_class_2

{

   // class definition

};

class derived_class: visibility_mode_1 base_class_1, visibility_mode_2 base_class_2

{

   // class definition

};
```

Description

The derived_class inherits the characteristics of two base classes, base_class_1 and base_class_2. The visibility_mode is specified for each base class while declaring a derived class. These modes can be different for every base class.

The following example illustrates Multiple Inheritance in C++:

```
#include <iostream>

using namespace std;

// class_A

class electronicDevice

{

    public:

        // constructor of the base class 1

        electronicDevice()

        {

            cout << "I am an electronic device.\n\n";

        }

};

// class_B

class Computer

{

    public:
```

```
    // constructor of the base class 2

    Computer()

    {

      cout << "I am a computer.\n\n";

    }

};

// class_C inheriting class_A and class_B

class Linux_based : public electronicDevice, public Computer

{};

int main()

{

  // create object of the derived class

    Linux_based obj; // constructor of base class A,

                // base class B and derived class

                // will be called

  return 0;

}
```

- **Multilevel Inheritance**

The inheritance in which a class can be derived from another derived class is known as Multilevel Inheritance. Suppose there are three classes A, B, and C. A is the base class that derives from class B. So, B is the derived class of A. Now, C is the class that is derived from class B. This makes class B, the base class for class C but is the derived class of class A. This scenario is known as the Multilevel Inheritance. The data members of each respective base class are accessed by their respective derived classes according to the specified visibility modes.

Syntax

```
class class_A

{

   // class definition

};

class class_B: visibility_mode class_A

{

   // class definition

};

class class_C: visibility_mode class_B

{

   // class definition

};
```

Description

The class_A is inherited by the sub-class class_B. The class_B is inherited by the subclass class_C. A subclass inherits a single class in each succeeding level.

Example

The following example illustrates Multilevel Inheritance in C++:

```
#include <iostream>
```

```
using namespace std;
// class_A
class electronicDevice
{
  public:
    // constructor of the base class 1
    electronicDevice()
    {
      cout << "I am an electronic device.\n\n";
    }
};
// class_B inheriting class_A
class Computer: public electronicDevice
{
  public:
    // constructor of the base class 2
    Computer()
    {
      cout << "I am a computer.\n\n";
    }
};

// class_C inheriting class_B

class Linux_based : public Computer

{

   public:

       // constructor of the derived class

       Linux_based()

       {

           cout << "I run on Linux.\n\n";;

       }

};

int main()

{

   // create object of the derived class

   Linux_based obj; // constructor of base class 1,

                   // base class 2, derived class will be called

   return 0;

}
```

In the above example, the base class electronicDevice is inherited by the subclass Computer which is further inherited by the subclass Linux_based. Since one class is inherited by a single class at each level, it is Multilevel Inheritance. The object of the derived class Linux_based can access the members of the class electronicDevice and Computer directly.

- **Hierarchical Inheritance**

The inheritance in which a single base class inherits multiple derived classes is known as the Hierarchical Inheritance. This inheritance has a tree-like structure since every class acts as a base class for one or more child classes. The visibility mode for each derived class is specified separately during the inheritance and it accesses the data members accordingly.

Syntax

```
class class_A

{

   // class definition

};

class class_B: visibility_mode class_A

{

   // class definition

};

class class_C : visibility_mode class_A

{

   // class definition

};

class class_D: visibility_mode class_B
```

```
{

    // class definition

};

class class_E: visibility_mode class_C

{

    // class definition

};
```

Description

The subclasses class_B and class_C inherit the attributes of the base class class_A. Further, these two subclasses are inherited by other subclasses class_D and class_E respectively.

Example

The following example illustrates Hierarchical Inheritance in C++:

```
#include <iostream>

using namespace std;

// base class

class electronicDevice

{

public:

    // constructor of the base class 1

    electronicDevice()

    {
        cout << "I am an electronic device.\n\n";

    }

};

// derived class inheriting base class

class Computer: public electronicDevice

{};

// derived class inheriting base class

class Linux_based : public electronicDevice

{};

int main()

{

    // create object of the derived classes

    Computer obj1;     // constructor of base class will be called

    Linux_based obj2;  // constructor of base class will be called

    return 0;

}
```

In the above example, the base class electronicDevice is inherited by two subclasses Computer and Linux_based. The class structure represents Hierarchical Inheritance. Both the derived classes can access the public members of the base class electronicDevice. When it creates objects of these two derived classes, it calls the constructor of the base class for both objects.

- **Hybrid Inheritance**

Hybrid Inheritance, as the name suggests, is the combination of two or over two types of inheritances. For example, the classes in a program are in such an arrangement that they show both single inheritance and hierarchical inheritance at the same time. Such an arrangement is known as the Hybrid Inheritance. This is arguably the most complex inheritance among all the types of inheritance in C++. The data members of the base class will be accessed according to the specified visibility mode.

Syntax

```
class class_A

{

    // class definition

};

class class_B

{

    // class definition

};

class class_C: visibility_mode class_A, visibility_mode class_B

{

    // class definition

};

class class_D: visibility_mode class_C

{
    // class definition

};

class class_E: visibility_mode class_C

{

    // class definition

};
```

Description

The derived class class_C inherits two base classes that are, class_A and class_B. This is the structure of Multiple Inheritance. And two subclasses class_D and class_E, further inherit class_C. This is the structure of Hierarchical Inheritance. The overall structure of Hybrid Inheritance includes more than one type of inheritance.

Example

The following example illustrates the Hybrid Inheritance in C++:

```
#include <iostream>

using namespace std;

// base class 1

class electronicDevice

{

public:

    // constructor of the base class 1

    electronicDevice()

{
        cout << "I am an electronic device.\n\n";
    }
};
// base class 2
class Computer
{
public:
    // constructor of the base class 2
    Computer()
    {
        cout << "I am a computer.\n\n";
    }
};
// derived class 1 inheriting base class 1 and base class 2
class Linux_based : public electronicDevice, public Computer
{};
// derived class 2 inheriting derived class 1
class Debian: public Linux_based
{};
int
```

```
main()

{

   // create an object of the derived class

   Debian obj; // constructor of base classes and

               // derived class will be called

   return 0;

}
```

Base Class:

In C++ or object-oriented programming, a base class is defined as any existing class from which other classes can be derived. A base class is sometimes alternately called a parent class or a superclass. The members and functions of a base class can be acquired by other non-base classes.

The syntax of the base class is the same as any other regular class syntax. The syntax is as follows:

```
class base_class

{

//class_members

//class_member_functions

}
```

Derived class

- A derived class is defined by specifying it relationship with the base class in addition to its own details.

Syntax:

```
class derived-class-name : visibility-mode base-class-name
{
          //members of derived class
}
```

Here,

- class is the required keyword,
- derived-class-name is the name given to the derived class,
- base-class-name is the name given to the base class,
- : (colon) indicates that the derived-class-name is derived from the base-class-name, visibility-mode is optional and, if present, may be either private or public.
- The default visibility-mode is private.
- Visibility mode specifies whether the features of the base class are privately derived or publicly derived.

Example:

```
class Shape
{
protected:
          float width, height;
public:
          void set_data (float a, float b)
          {
                    width = a;
                    height = b;
          }
};

class Rectangle: public Shape
{
public:
        float area ()
        {
                return (width * height);
        }
};

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

void main ()
{
        Rectangle rect;
        Triangle tri;
        rect.set_data (5,3);
        tri.set_data (2,5);
        cout << rect.area() << endl;
        cout << tri.area() << endl;
        getch();
}
```

Access to the base class members

The base class members can be accessed by its sub-classes through access specifiers. There are three types of access specifies. They are public, private and protected.

1. Public

When the base class is publicly inherited, the public members of the base class become the derived class public members.

Also read :Explain different datatypes in C++

They can be accessed by the objects of the derived class.

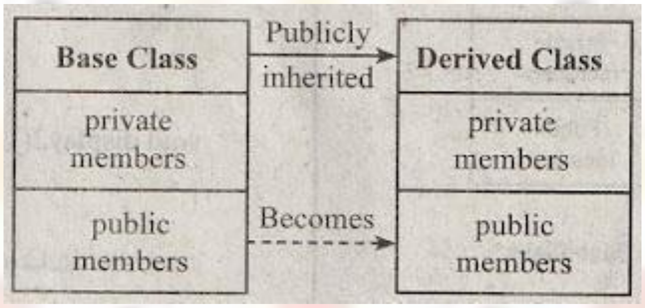


Figure: Publicly Inherited Base Class

Syntax

```
class classname

{

public:

datatype variablename;

returntype functionname();

};
```

2. Protected

When the base class is derived in protected mode, the 'protected' and 'public' members of the base class become the protected members of the derived class.

Also read :Structure of a C++ program

Private and protected members of a class can be accessed by,
(i) A friend function of the class.

(ii) A member function of the friend class.
(iii) A derived class member function.

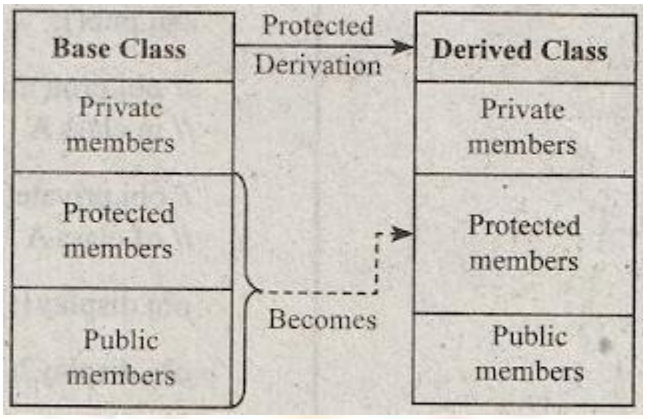


Figure: Protected Derivation of the Base Class

Also read :Abstraction, Encapsulation, Inheritance and Polymorphism

Syntax

class classname

{

protected: :

datatype variablename;

returntype functionname();

};

3.Private
When a base class contains members, that are declared as private, they cannot be accessed by the derived class objects. They can be accessed only by the class in which they are defined.

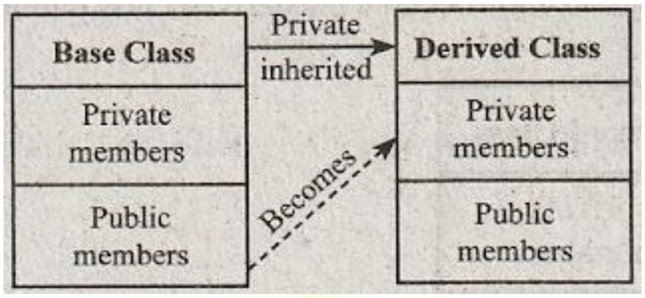


Figure: Privately inherited Base Class

Syntax

Also read :Differences between OOP and procedure oriented programming

```
class classname
{
private:
datatype variablename;
returntype functionname( );
};
```

Program

```
#include  <iostream>
using namespace std;
class A
{
public: pub()
{
cout << “In Public() \n”;
}
protected: prot()
```

```
{
cout << “In Protected() \n”;
}
private: priv()
{
cout << “In Pivate() \n”;
}
};
class C : public A
{
public:
void display1()
{
cout<< “In C::display1 call\n”;
pub();
}
void display2()
{
cout << “In C::display2 call\n”;
prot();
}
/*pri(Q is a private member of class A. Therefore it is an illegal access
void display3()
{
cout<<“In C::display3 call\n”;
priv();

} */

}

main()

{

C obj;

obj.pub();

// obj.prot(); illegal because it is declared as protected in class A

// obj.private(); illegal because pri() isa private member of class A

obj.display1();

obj.display2();

}
```

Base and Derived class constructor:

When we inherit class into another class then object of base class is initialized first. If a class do not have any constructor then default constructor will be called. But if we have created any parameterized constructor then we have to initialize base class constructor from derived class.

We have to call constructor from another constructor. It is also known as constructor chaining.

When we have to call same class constructor from another constructor then we use this keyword. In addition, when we have to call base class constructor from derived class then we use base keyword.

Example of Base Class Constructor Calling

In the following example we have created Abc is as a base class and inherit it into Pqr. We have created parameterized constructor in class Abc so we have to initialize this constructor from derived class constructor in the following manner:-

```
public class Abc

{

   public int p, q;
```

```
    public Abc(int p1, int p2)
    {
        p = p1;
        q = p2;
    }
    public int sum(int x, int y)
    {
        return (x + y);
    }


}
//derived class/ child class
public class Pqr : Abc
{
    public int a;
    public Pqr(int a1,int p1, int p2):base(p1,p2)
    {
        a = a1;
    }
    public int sub(int x, int y)
    {
        return (x - y);
    }
}
```

When we create the object of Pqr class then first it will call Pqr class constructor but Pqr class constructor first initialize the base class constructor then Pqr constructor will be initialized. It is very important point to note down the base class constructor initialized first.

Destructors:

Destructors in C++ are members functions in a class that delete an object. They are called when the class object goes out of scope such as when the function ends, the program ends, a delete variable is called etc.

Destructors are different from normal member functions as they don’t take any argument and don’t return anything. Also, destructors have the same name as their class and their name is preceded by a tilde(~).

A program that demonstrates destructors in C++ is given as follows.

Example

```
#include<iostream>
using namespace std;
class Demo {
  private:
  int num1, num2;
  public:
  Demo(int n1, int n2) {
    cout<<"Inside Constructor"<<endl;
    num1 = n1;
    num2 = n2;
  }
  void display() {
    cout<<"num1 = "<< num1 <<endl;
    cout<<"num2 = "<< num2 <<endl;
  }
  ~Demo() {
    cout<<"Inside Destructor";
  }
};
int main() {
  Demo obj1(10, 20);
  obj1.display();
  return 0;
}
```

Output

```
Inside Constructor
num1 = 10
num2  = 20
Inside Destructor
```

Virtual base class:

Virtual classes are primarily used during multiple inheritance. To avoid, multiple instances of the same class being taken to the same class which later causes ambiguity, virtual classes are used.

Example

```
#include  <iostream>
using namespace std;
class A {
  public:
  int a;
  A(){
    a = 10;
  }
};
class B : public virtual A {
};
class C : public virtual A {
};
class D : public B, public C {
};
int main(){
  //creating class D object
  D object;
  cout << "a = " << object.a << endl;
  return 0;
}
```

Output

a = 10

UNIT - IV

I/O using C functions:

When we say **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

The Standard Files

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen.

The getchar() and putchar() Functions

The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen. Check the following example –

```
#include <stdio.h>
int main( ) {

   int c;

   printf( "Enter a value :");
   c = getchar( );

   printf( "\nYou entered: ");
   putchar( c );

   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows −

```
$./a.out
Enter a value : this is test
You entered: t
```

The gets() and puts() Functions

The **char *gets(char *s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (End of File).

The **int puts(const char *s)** function writes the string 's' and 'a' trailing newline to **stdout**.

NOTE: Though it has been deprecated to use gets() function, Instead of using gets, you want to use fgets().

```
#include <stdio.h>
int main( ) {

   char str[100];

   printf( "Enter a value :");
   gets( str );

   printf( "\nYou entered: ");
   puts( str );

   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows −

```
$./a.out
```

```
Enter a value : this is test
You entered: this is test
```

The scanf() and printf() Functions

The **int scanf(const char *format, ...)** function reads the input from the standard input stream **stdin** and scans that input according to the **format** provided.

The **int printf(const char *format, ...)** function writes the output to the standard output stream **stdout** and produces the output according to the format provided.

The **format** can be a simple constant string, but you can specify %s, %d, %c, %f, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements. Let us now proceed with a simple example to understand the concepts better −

```
#include <stdio.h>
int main( ) {

  char str[100];
  int i;

  printf( "Enter a value :");
  scanf("%s %d", str, &i);

  printf( "\nYou entered: %s %d ", str, i);

  return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then program proceeds and reads the input and displays it as follows −

```
$./a.out
Enter a value : seven 7
You entered: seven 7
```

Here, it should be noted that scanf() expects input in the same format as you provided %s and %d, which means you have to provide valid inputs like "string integer". If you provide "string string" or "integer integer", then it will be assumed as wrong input. Secondly, while reading a string, scanf() stops reading as soon as it encounters a space, so "this is test" are three strings for scanf().

Stream classes hierarchy:

In C++ stream refers to the stream of characters that are transferred between the program thread and i/o.

Stream classes in C++ are used to input and output operations on files and io devices. These classes have specific features and to handle input and output of the program.

The **iostream.h** library holds all the stream classes in the C++ programming language.

Now, let's learn about the classes of the *iostream* library.

ios class − This class is the base class for all stream classes. The streams can be input or output streams. This class defines members that are independent of how the templates of the class are defined.

istream Class − The istream class handles the input stream in c++ programming language. These input stream objects are used to read and interpret the input as a sequence of characters. The cin handles the input.

ostream class − The ostream class handles the output stream in c++ programming language. These output stream objects are used to write data as a sequence of characters on the screen. cout and puts handle the out streams in c++ programming language.

Example

OUT STREAM

```
#include  <iostream>
using namespace std;
int main(){
   cout<<"This output is printed on screen";
}
```

Output

```
This output is printed on screen
```

PUTS

```
#include  <iostream>
using namespace std;
int main(){
   puts("This output is printed using puts");
}
```

Output

```
This output is printed using puts
```

IN STREAM

CIN

```
#include  <iostream>
using namespace std;
int main(){
   int no;
```

```
  cout<<"Enter a number ";
  cin>>no;
  cout<<"Number entered using cin is "<
```

Output

```
Enter a number 3453
Number entered using cin is 3453
```

gets

```
#include  <iostream>
using namespace std;
int main(){
  char ch[10];
  puts("Enter a character array");
  gets(ch);
  puts("The character array entered using gets is : ");
  puts(ch);
}
```

Output

```
Enter a character array
thdgf
The character array entered using gets is :
Thdgf
```

Stream I/O:

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters. This chapter will discuss very basic and most common I/O operations required for C++ programming.

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like 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 like 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 −

Sr.No	Header File & Function and Description
1	**<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.
2	**<iomanip>** This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as **setw** and **setprecision**.
3	**<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>

using namespace std;

int main() {
   char str[] = "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>

using namespace std;

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 following result −

```
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 −

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

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>

using namespace std;

int main() {
  char str[] = "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>

using namespace std;

int main() {
  char str[] = "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 the difference becomes obvious. So it is good practice to display error messages using cerr stream and while displaying other log messages then clog should be used.

File Streams:

File streams in C++ are basically the libraries that are used in the due course of programming. The programmers generally use the iostream standard library in the C++ programming as it provides the cin and cout methods that are used for reading from the input and writing to the output respectively.

In order to read and write from a file, the programmers are generally using the standard C++ library that is known as the fstream.

Here is the list of the data types that are defined in the fstream library:

Data Type	Description
fstream	This data types is generally used to create files, write information to files, and read information from files.
ifstream	This data types is generally used to read information from files.
ofstream	This data types is generally used to create files and write information to the files.

Example 1(Writing content to a file)

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
  ofstream filestream("test.txt");
  if (filestream.is_open())
  {
    filestream << "Welcome to the world of C++ Tutorial.\n";
    filestream << "Hello user.\n";
    filestream.close();
  }
  else
  {
  cout <<"No Such File created.";
  }
  return 0;
}
```

Output : This program create a **test.txt** write the info inside the file

```
Welcome to the world of C++ Tutorial.
Hello user.
```

String streams:

Here we will see the string stream in C++. The string stream associates a string object with a string. Using this we can read from string as if it were a stream like cin.

The Stringstream has different methods. These are like below −

clear(): Used to clear the stream

str(): To get and set the string object whose content is present in stream

operator << : This will add one string into the stringstream

operator >> : This is used to read from stringstream object.

Let us see two examples of string streams. In the first program we will divide the words into separate strings.

Example

```
#include <iostream>
#include <vector>
#include <string>
#include <sstream>
using namespace std;
int main() {
  string str("Hello from the dark side");
  string tmp; // A string to store the word on each iteration.
  stringstream str_strm(str);
  vector<string> words; // Create vector to hold our words
  while (str_strm >> tmp) {
    // Provide proper checks here for tmp like if empty
    // Also strip down symbols like !, ., ?, etc.
    // Finally push it.
    words.push_back(tmp);
  }
  for(int i = 0; i<words.size(); i++)
    cout << words[i] << endl;
}
```

Output

```
Hello
from
the
dark
side
```

Here we will convert Decimal to Hexadecimal using string stream.

Example

```
#include<iostream>
#include<sstream>
using namespace std;
main(){
  int decimal = 61;
  stringstream  my_ss;
  my_ss << hex << decimal;
  string res = my_ss.str();
  cout << "The hexadecimal value of 61 is: " << res;
}
```

Output

```
The hexadecimal value of 61 is: 3d
```

Example 1(Writing content to a file)

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
  ofstream filestream("test.txt");
  if (filestream.is_open())
  {
    filestream << "Welcome to the world of C++ Tutorial.\n";
    filestream << "Hello user.\n";
    filestream.close();
  }
  else
  {
  cout <<"No Such File created.";
  }
  return 0;
}
```

Output : This program create a **test.txt** write the info inside the file

```
Welcome to the world of C++ Tutorial.
Hello user.
```

Example 2(Reading Content from a file)

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
  string srg;
  ifstream filestream("test.txt");
  if (filestream.is_open())
  {
    while ( getline (filestream,srg) )
    {
      cout << srg <<endl;
    }
    filestream.close();
  }
  else
  {
      cout << "No such file found."<<endl;
    }
  return 0;
}
```

Operators Overloading in C++

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Box operator+(const Box&);

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows −

Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below –

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

class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }
      void setLength( double len ) {
         length = len;
      }
      void setBreadth( double bre ) {
         breadth = bre;
      }
      void setHeight( double hei ) {
         height = hei;
      }

      // Overload + operator to add two Box objects.
      Box operator+(const Box& b) {
         Box box;
         box.length = this->length + b.length;
         box.breadth = this->breadth + b.breadth;
         box.height = this->height + b.height;
         return box;
      }

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};

// Main function for the program
int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 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;

  // Add two object as follows:
  Box3 = Box1 + Box2;

  // volume of box 3
  volume = Box3.getVolume();
  cout << "Volume of Box3 : " << 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
Volume of Box3 : 5400
```

Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded −

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

Following is the list of operators, which can not be overloaded –

Error handling during file operations:

Sometimes during file operations, errors may also creep in. For example, a file being opened for reading might not exist. Or a file name used for a new file may already exist. Or an attempt could be made to read past the end-of-file. Or such as invalid operation may be performed. There might not be enough space in the disk for storing data.

To check for such errors and to ensure smooth processing, C++ file streams inherit 'stream-state' members from the ios class that store the information on the status of a file that is being currently used. The current state of the I/O system is held in an integer, in which the following flags are encoded :

Name	Meaning
eofbit	1 when end-of-file is encountered, 0 otherwise.
failbit	1 when a non-fatal I/O error has occurred, 0 otherwise
badbit	1 when a fatal I/O error has occurred, 0 otherwise
goodbit	0 value

C++ Error Handling Functions

There are several error handling functions supported by class ios that help you read and process the status recorded in a file stream.

Following table lists these error handling functions and their meaning :

Function	Meaning

int bad()	Returns a non-zero value if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is zero (false value), it may be possible to recover from any other error reported and continue operations.
int eof()	Returns non-zero (true value) if end-of-file is encountered while reading; otherwise returns zero (false value).
int fail()	Returns non-zero (true) when an input or output operation has failed.
int good()	Returns non-zero (true) if no error has occurred. This means, all the above functions are false. For example, if fin.good() is true, everything is okay with the stream named as fin and we can proceed to perform I/O operations. When it returns zero, no further operations can be carried out.
clear()	Resets the error state so that further operations can be attempted.

The above functions can be summarized as eof() returns true if eofbit is set; bad() returns true if badbit is set. The fail() function returns true if failbit is set; the good() returns true there are no errors. Otherwise, they return false.

These functions may be used in the appropriate places in a program to locate the status of a file stream and thereby take the necessary corrective measures. For example :

```
:
ifstream fin;
fin.open("master", ios::in);
while(!fin.fail())
{
   :       // process the file
}
if(fin.eof())
{
   :        // terminate the program
}
else if(fin.bad())
{
   :        // report fatal error
}
else
{
  fin.clear();     // clear error-state flags
  :
}
:
```

C++ Error Handling Example

Here is an example program, illustrating error handling during file operations in a C++ program:

```
/* C++ Error Handling During File Operations
 * This program demonstrates the concept of
 * handling the errors during file operations
 * in a C++ program */

#include<iostream.h>
#include<fstream.h>
#include<process.h>
#include<conio.h>
void main()
{
  clrscr();
  char fname[20];
  cout<<"Enter file name: ";
  cin.getline(fname, 20);
  ifstream fin(fname, ios::in);
  if(!fin)
  {
    cout<<"Error in opening the file\n";
    cout<<"Press a key to exit...\n";
    getch();
    exit(1);
  }
  int val1, val2;
  int res=0;
  char op;
  fin>>val1>>val2>>op;
  switch(op)
  {
    case '+':
      res = val1 + val2;
      cout<<"\n"<<val1<<" + "<<val2<<" = "<<res;
      break;
    case '-':
      res = val1 - val2;
      cout<<"\n"<<val1<<" - "<<val2<<" = "<<res;
      break;
```

```
          case '*':
            res = val1 * val2;
            cout<<"\n"<<val1<<" * "<<val2<<" = "<<res;
            break;
          case '/':
            if(val2==0)
            {
              cout<<"\nDivide by Zero Error..!!\n";
              cout<<"\nPress any key to exit...\n";
              getch();
              exit(2);
            }
            res = val1 / val2;
            cout<<"\n"<<val1<<" / "<<val2<<" = "<<res;
            break;

        }

        fin.close();

        cout<<"\n\nPress any key to exit...\n";
        getch();
      }
```

Let's suppose we have four files with the following names and data, shown in this table:

File Name	Data
myfile1.txt	10 5 /
myfile2.txt	10 0 /
myfile3.txt	10 5 +
myfile4.txt	10 0 +

Formatted I/O: The **iomanip.h** and **iostream.h** header files are used to perform the formatted IO operations in C++.
The C++ programming language provides the several built-in functions to display the output in formatted form. These built-in functions are available in the header file **iomanip.h** and **ios** class of header file **iostream.h**.

In C++, there are two ways to perform the formatted IO operations.

- Using the member functions of ios class.
- Using the special functions called manipulators defined in iomanip.h.

Formatted IO using ios class memebers
The **ios** class contains several member functions that are used to perform formmated IO operations.

The **ios** class also contains few format flags used to format the output. It has format flags like **showpos**, **showbase**, **oct**, **hex**, etc. The format flags are used by the function **setf( )**.

The following table provides the details of the functions of **ios** class used to perform formatted IO in C++.

Function	Description
width(int)	Used to set the width in number of character spaces for the immediate output data.
fill(char)	Used to fill the blank spaces in output with given character.
precision(int)	Used to set the number of the decimal point to a float value.
setf(format flags)	Used to set various flags for formatting output like showbase, showpos, oct, hex, etc.
unsetf(format flags)	Used to clear the format flag setting.

All the above functions are called using the built-in object **cout**.

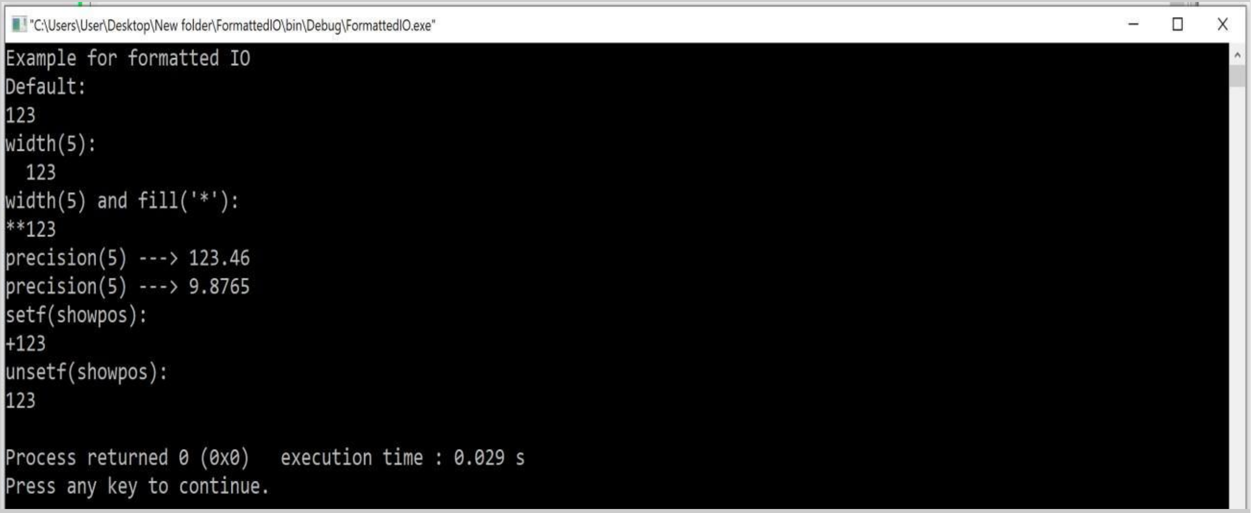
Lets look at the following code to illustrate the formatted IO operations using member functions of ios class.

Example - Code to illustrate the formatted IO using ios class

```
#include <iostream>
#include <fstream>


using namespace std;
```

```
int main()
{
   cout << "Example for formatted IO" << endl;

   cout << "Default: " << endl;
   cout << 123 << endl;

   cout << "width(5): " << endl;
   cout.width(5);
   cout << 123 << endl;

   cout << "width(5) and fill('*'): " << endl;
   cout.width(5);
   cout.fill('*');
   cout << 123 << endl;

   cout.precision(5);
   cout << "precision(5) ---> " << 123.4567890 << endl;
   cout << "precision(5) ---> " << 9.876543210 << endl;

   cout << "setf(showpos): " << endl;
   cout.setf(ios::showpos);
   cout << 123 << endl;

   cout << "unsetf(showpos): " << endl;
   cout.unsetf(ios::showpos);
   cout << 123 << endl;

   return 0;
}
```

Output


"C:\Users\User\Desktop\New folder\FormattedIO\bin\Debug\FormattedIO.exe"

```
Example for formatted IO
Default:
123
width(5):
  123
width(5) and fill('*'):
**123
precision(5) ---> 123.46
precision(5) ---> 9.8765
setf(showpos):
+123
unsetf(showpos):
123

Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.
```

Formatted IO using manipulators

The **iomanip.h** header file contains several special functions that are used to perform formmated IO operations.

The following table provides the details of the special manipulator functions used to perform formatted IO in C++.

Function	Description
setw(int)	Used to set the width in number of characters for the immediate output data.
setfill(char)	Used to fill the blank spaces in output with given character.
setprecision(int)	Used to set the number of digits of precision.
setbase(int)	Used to set the number base.
setiosflags(format flags)	Used to set the format flag.
resetiosflags(format flags)	Used to clear the format flag.

The **iomanip.h** also contains the following format flags using in formatted IO in C++.

Flag	Description
endl	Used to move the cursor position to a newline.

Flag	Description
ends	Used to print a blank space (null character).
Dec	Used to set the decimal flag.
Oct	Used to set the octal flag.
Hex	Used to set the hexadecimal flag.
Left	Used to set the left alignment flag.
right	Used to set the right alignment flag.
showbase	Used to set the showbase flag.
noshowbase	Used to set the noshowbase flag.
showpos	Used to set the showpos flag.
noshowpos	Used to set the noshowpos flag.
showpoit	Used to set the showpoit flag.
noshowpoint	Used to set the noshowpoint flag.

Lets look at the following code to illustrate the formatted IO operations using manipulators.

Example - Code to illustrate the formatted IO using manipulators

```
#include <iostream>
#include <fstream>

using namespace std;

void line() {
   cout << "----------------------------------------" << endl;
}

int main()
{
   cout << "Example for formatted IO" << endl;
```

```
    line();
    cout << "setw(10): " << endl;
    cout << setw(10) << 99 << endl;
    line();
    cout << "setw(10) and setfill('*'): " << endl;
    cout << setw(10) << setfill('*') << 99 << endl;
    line();
    cout << "setprecision(5): " << endl;
    cout << setprecision(5) << 123.4567890 << endl;
    line();
    cout << "showpos: " << endl;
    cout << showpos << 999 << endl;
    line();
    cout << "hex: " << endl;
    cout << hex << 100 << endl;
    line();
    cout << "hex and showbase: " << endl;
    cout << showbase << hex << 100 << endl;
    line();

    return 0;
}
```