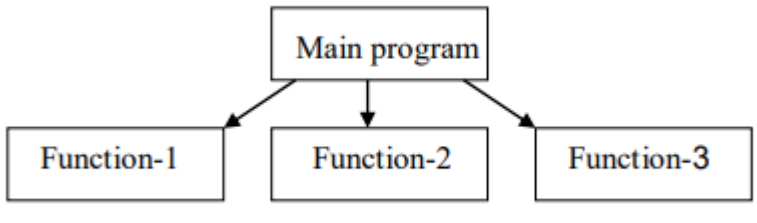
OOPS

Procedure Oriented Programming Language

In the procedure oriented approach, the problem is viewed as sequence of things to be done such as reading , calculation, and printing.

Procedure oriented programming basically consist of writing a list of instructions or actions for the computer to follow and organizing these instructions into groups known as functions.

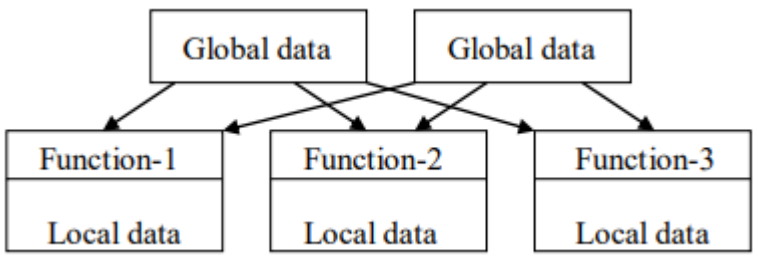


The disadvantages of the procedure oriented programming languages are:

Global data access

It does not model real word problem very well

No data hiding



Characteristics of procedure oriented programming:

Emphasis is on doing things (algorithms).

Large programs are divided into smaller programs known as ***functions.***

Most of the functions ***share global data.***

Data move openly around the system from function to function.

Function transforms data from one form to another.

Employs ***top-down approach*** in program design

NO NOTES TO SHOW

OOPS

Object Oriented Programming

In ***traditional programming***, programs are basically lists of instructions to the computer that define data and then work with that data (via statements and functions). Data and the functions that work on that data are ***separate entities*** that are combined together to produce the desired result. Because of this separation, traditional programming often does not provide a very intuitive representation of reality. It's up to the programmer to manage and connect the properties (variables) to the behaviors (functions) in an appropriate manner. This leads to code that looks like this:

```
driveTo(you, work);
```

So what is ***Object-Oriented Programming?*** Lets understand through an analogy. Take a look around you -- everywhere you look are objects: books and buildings and food and even you.

Objects have two major components to them:

1) A list of relevant properties (e.g. weight, color, size, solidity, shape)

2) Some number of behaviors that they can exhibit (e.g. being opened, making something else hot, etc.).

These properties and behaviors are inseparable.

Object-Oriented Programming (OOP) provides us with the ability to create objects that tie together both properties and behaviors into a self-contained, reusable package. This leads to code that looks more like this:

```
you.driveTo(work);
```

- This not only reads more clearly, it also makes it clearer who the subject is (you) and what behavior is being invoked (driving somewhere).
- Rather than being focused on writing functions, we're focused on defining objects that have a ***well-defined set of behaviors***. This is why the paradigm is called "object-oriented".
- This allows programs to be written in a more modular fashion, which makes them easier to write and understand, and also provides a higher degree of code-reusability.
- These objects also provide a more intuitive way to work with our data by allowing us to define how we interact with the objects, and how they interact with other objects.

Note that OOP doesn't replace traditional programming methods. Rather, it gives you additional tools in your programming tool belt to manage complexity when needed.

Object-oriented programming also brings several other useful concepts to the table: ***inheritance, encapsulation, abstraction, and polymorphism***.

Lets start learning them One by One!

NO NOTES TO SHOW

OOPS

Classes

In C++, a class is declared using the **class** keyword. The syntax of a class declaration is similar to that of a structure. Its general form is:

```
class class-name{
    // Private functions and Variables
    public:
    // Public functions and variables
} object-list;
```

In a class declaration, the object list is **optional**.

The class name is technically optional. From a practical point of view, it is virtually always needed. The reason is that the class name becomes a new type name that is used to declare objects of the class.

Functions and variables declared inside the class declaration are said to be **members** of the class.

By default, **all member functions and variables are private** to that class. This means that they are accessible only by other members of that class.

To declare public class members, the **public** keyword is used, followed by a colon. All functions and variables declared after the public specifier are accessible both by other members of the class and by any part of the program that contains the class.

```
#include <bits/stdc++.h>
using namespace std;
// Class Declartion
class myclass {
   // Private Members to myclass
   int a;

  public:
   // Public members to myclass
   void set_a(int num);
   int get_a();
};
int main() { return 0; }
```

This class has one private variable, called ***a***, and two public functions ***seta()*** *and* ***geta()***.

Notice that the functions are declared within a class using their prototype forms. The functions that are declared to be part of a class are called member functions.

Since ***a*** is private it is not accessible by any code outside ***myclass***. However, since ***seta()*** *and* ***geta()*** are members of ***myclass***, they have access to ***a*** and as they are declared as public member of ***myclass***, they can be called by any part of the program that contains ***myclass***.

The member functions need to be defined. You do this by preceding the function name with the class name followed by two colons (:: are called scope resolution operator). For example, after the class declaration, you can declare the member functions as

```
#include <bits/stdc++.h>
using namespace std;
// Class Declartion
class myclass {
   // Private Members to myclass
   int a;

  public:
   // Public members to myclass
   void set_a(int num);
   int get_a();
};

// Member Functions
```

```
void myclass::set_a(int num) { a = num; }
int myclass::get_a() { return a; }

int main() {
   myclass c;
   c.set_a(100);
   cout << c.get_a();
   return 0;
}
```

Output:

```
100
```

In general to declare a member function, you use this form:

```
return-type class-name::func-name(parameter-list){
    // Function Body
}
```

Here the class-name is the name of the class to which the function belongs.

The declaration of a class does not define any objects of the type ***myclass***. It only defines the type of object that will be created when one is actually declared. To create an object, use the class name as type specifier. For example,

```
int main() {
   myclass c;
   c.set_a(100);
   cout << c.get_a();
   return 0;
}
```

Remember that an object declaration creates a physical entity of that type. That is, **an object occupies memory space, but a type definition does not**.

Once an object of a class has been created, your program can reference its public members by using the dot operator in much the same way that structure members are accessed as shown above.

It is important to remember that although all objects of a class share their functions, each object creates and maintains its own data.

NO NOTES TO SHOW

OOPS

Constructors

When applied to real problems, virtually every object you create will require some sort of ***initialization***. C++ allows a constructor function to be included in a class declaration. A class's constructor is called each time an object of that class is created. Thus, any initialization to be performed on an object can be done automatically by the constructor function. A constructor function has the ***same name as the class*** of which it is a part and does not have return type. Here is a short example,

```
#include <iostream>
using namespace std;
// Class Declaration
class myclass {
   int a;

  public:
   // constructor
   myclass();
   void show();
};
myclass::myclass() {
   cout << "In constructor\n";
   a = 10;
}

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

int main() {
   // Automatic call to constructor
   myclass ob;
   ob.show();
   return 0;
}
```

Output

```
In constructor
10
```

In this simple example the constructor is called when the object is created, and the constructor initializes the private variable a to 10.

For a global object, its constructor is called once, when the program first begins execution. For local objects, the constructor is called each time the declaration statement is executed.

Destructors

The complement of a constructor is the ***destructor***. This function is called when an object is destroyed.

For example, an object that allocates memory when it is created will want to free that memory when it is destroyed. The name of a destructor is the **name of its class preceded by a** ~. For example,

```
#include <iostream>
using namespace std;
// Class Declaration
class myclass {
   int a;

  public:
   // constructor
   myclass();
   // Destructor
   ~myclass();
```

```
   void show();
};

myclass::myclass() {
   cout << "In constructor\n";
   a = 10;
}

myclass::~myclass() { cout << "In Destructor\n"; }

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

int main() {
   // Automatic call to constructor
   myclass ob;
   ob.show();
   return 0;
}
```

Output

```
In constructor
10
In Destructor
```

A class's destructor is called when an object is destroyed. Local objects are destroyed when they go ***out of scope***.

Global objects are destroyed when the ***program ends***. It is not possible to take the address of either a constructor or a destructor.

Constructors that take parameters. It is possible to pass one or more arguments to a constructor function. Simply add the appropriate parameters to the constructor function's declaration and definition. Then, when you declare an object, specify the arguments.

```
#include <iostream>
using namespace std;
// Class Declaration
class myclass {
   int a;

  public:
   // constructor
   myclass(int aa);
   // Destructor
   ~myclass();
   void show();
};

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

myclass::~myclass() { cout << "In Destructor\n"; }

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

int main() {
   // Automatic call to constructor
   myclass ob(100);
   ob.show();
   return 0;
}
```

Output:

```
In constructor
100
In Destructor
```

Pay particular attention to how ***ob*** is declared in ***main( )***. The value 100, specified in the parentheses following ***ob***, is the argument that is passed to myclass( )'s parameter x that is used to initialise a.

Actually, the syntax is shorthand for this longer form: ***myclass ob = myclass(4);***

Unlike constructor functions, ***destructors cannot have parameters***.

Although the previous example has used a constant value, you can pass an object's constructor any valid expression, including variables.

OOPS

Access Modifiers

One of the main features of object-oriented programming languages such as C++ is ***data hiding***.

Data hiding refers to restricting access to data members of a class. This is to prevent other functions and classes from tampering with the class data.

However, it is also important to make some member functions and member data ***accessible*** so that the hidden data can be manipulated indirectly.

The access modifiers of C++ allows us to determine which class members are accessible to other classes and functions, and which are not.

For example:

```
#include <bits/stdc++.h>
using namespace std;
class Course {
  private:
   int id;
   string name;

  public:
   void getUserCount() {
      // code
   }

   void enrollUser() {
      // code
   }
};

int main() {
   Course c;
   return 0;
}
```

Here, the variables ***id*** and ***name*** of the ***Course*** class are hidden using the ***private*** keyword, while the member functions are made accessible using the ***public*** keyword.

Types of C++ Access Modifiers

In C++, there are 3 access modifiers:

Public

Private

Protected

Public Access Modifier

The ***public*** keyword is used to create public members (data and functions).

The public members are accessible from any part of the program.

```
#include <bits/stdc++.h>
using namespace std;
class Course {
  public:
   int users = 100;
   int getUserCount() { return users; }
};
```

```
int main() {
   Course c;
   cout<<"Enter Number of users: ";
   cin >> c.users;
   cout << c.getUserCount();
   return 0;
}
```

Output

```
Enter Number of users: 3
3
```

In this program, we have created a class named Course, which contains a ***public*** variable age and a ***public*** function ***getUserCount()***

In ***main()***, we have created an object of the ***Course*** class named ***c***. We then access the ***public*** elements directly by using the codes ***c.users*** and ***c.getUserCount()***

Notice that the ***public*** elements are accessible from ***main()***. This is because ***public*** elements are accessible from all parts of the program.

Private Access Modifier

The ***private*** keyword is used to create private members (data and functions).

The private members can only be accessed from within the class.

However, friend classes and friend functions can access private members.

```
#include <bits/stdc++.h>
using namespace std;
class Course {
  private:
   int id;
   string name;
   int users = 7;

  public:
   int getUserCount() { return users; }
};

int main() {
   Course c;
   cout << c.getUserCount();
   return 0;
}
```

Output

```
7
```

In ***main()***, the object ***c*** cannot directly access the class variable ***users***.

```
// Compilation Error
cout << c.users;
```

Protected Access Modifier

The ***Protected*** keyword is used to create protected members (data and function).

The ***Protected*** members can be accessed within the class and from the ***derived*** class.

Derived Class and Inheritance are discussed further in this module.

```
#include <bits/stdc++.h>
using namespace std;
class Course {
```

```
  protected:
   int users = 12;
};

class Batch : public Course {
  public:
   int getUserCount() { return users; }
};

int main() {
   Batch b;
   cout << b.getUserCount();
   return 0;
}
```

Output

```
12
```

Here, ***Batch*** is an inherited class that is derived from ***Course***. The variable ***users*** is declared in ***Course*** with the ***protected*** keyword.

This means that ***Batch*** can access ***users*** since ***Course*** is its parent class.

Summary

public: elements can be accessed by ***all*** other classes and functions.

private: elements cannot be accessed outside the class in which they are declared, except by ***friend*** classes and functions.

protected: elements are just like the private, except they can be accessed by ***derived*** classes.

Specifiers	Same Class	Derived Class	Outside Class
public	**Yes**	**Yes**	**Yes**
private	**Yes**	**No**	**No**
protected	**Yes**	**Yes**	**No**

Note: By default, class members in C++ are ***private***, unless specified otherwise.

OOPS

Static Class Members

We can define class members static using ***static*** keyword.

Characteristics

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.

```
#include <iostream>
using namespace std;
class item {
   static int count;
   int number;

  public:
   void getdata(int a) {
      number = a;
      count++;
   }
   void getcount() { cout << "Count is : " << count << endl; }
};
int item::count = 7;
int main() {
   item a, b, c;
   a.getcount();
   b.getcount();
   c.getcount();
   a.getdata(100);
   b.getdata(200);
   c.getdata(300);
   cout << "After Reading Data \n";
   a.getcount();
   b.getcount();
   c.getcount();
   return 0;
}
```

Output:

```
Count is : 7
Count is : 7
Count is : 7
After Reading Data
Count is : 10
Count is : 10
Count is : 10
```

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.

```
#include <iostream>
using namespace std;
class Box {
  private:
   double length;
   double breadth;
   double height;

  public:
   static int objectCount;
   Box(double l, double b, double h) {
      cout << "Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
      objectCount++;
   }
   static int getCount() { return objectCount; }
};
// Initialize static member of class
int Box::objectCount = 0;
int main(void) {
   // Print total number of objects before creating object.
   cout << "Inital Count: " << Box::getCount() << endl;
   Box Box1(1, 2, 3);
   Box Box2(4, 5, 6);
   // Print total number of objects after creating object.
   cout << "Final Count: " << Box::getCount() << endl;
   return 0;
}
```

Output:

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

OOPS

Inline Functions

Definition

An inline function is a function that is expanded in line when it is invoked. Inline expansion makes a program run faster because the overhead of a function call and return is eliminated. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. ***This substitution is performed by the C++ compiler at compile time***. Inline function may increase efficiency if it is small. It is defined by using keyword "***inline***"

Why Inline Functions?

- One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times.
- Every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function.
- When a function is small, a substantial percentage of execution time may be spent in such overheads.
- One solution to this problem is to use macro definitions, known as macros. Preprocessor macros are popular in C.
- The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.
- C++ has different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called inline function.

Syntax

```
inline return-type function-name(parameters)
{
    // function code
}
```

Properties

- Inlining is only a request to the compiler, not a command.
- **Compiler can ignore the request for inlining**.
- Compiler may not perform inlining in such circumstances like:
    - If a function contains a **loop**. (for, while, do-while)
    - If a function is **recursive**.
    - If a function contains **static** variables.
    - If a function return type is other than **void**, and the return statement doesn’t exist in function body.
    - If a function contains **switch** or **goto** statement.

Example 1

```
#include <iostream>
using namespace std;
inline int square(int a) { return a * a; }
int main() {
   cout << "The Square of 5 is " << square(5) << endl;
   return 0;
}
```

Output

```
The Square of 5 is 25
```

Example 2 (Inline Functions with Classes)

- It is also possible to define the inline function inside the class.
- **All the functions defined inside the class are implicitly inline.**
- If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.

```
// Demonstrating Inline Functions
#include <iostream>
using namespace std;
class azcalc {
   int a, b, add, sub;
  public:
   void getInput();
   void addition();
   void subtract();
};
inline void azcalc ::getInput() {
   cout << "Enter first value : ";
   cin >> a;
   cout << "Enter second value : ";
   cin >> b;
}
inline void azcalc ::addition() {
   add = a + b;
   cout << "Addition of two numbers : " << a + b << "\n";
}
inline void azcalc ::subtract() {
   sub = a - b;
```

```
   cout << "Difference of two numbers : " << a - b << "\n";
}
int main() {
   azcalc s;
   s.getInput();
   s.addition();
   s.subtract();
   return 0;
}
```


NO NOTES TO SHOW

OOPS

Friend Functions

A ***friend function*** is a function that can ***access the private members*** of a class as though it were a member of that class. To declare a friend function, simply use the ***friend keyword*** in front of the prototype of the function you wish to be a friend of the class. It does not matter whether you declare the friend function in the private or public section of the class.

```
#include <iostream>
using namespace std;
class Adder {
  private:
   int val;

  public:
   Adder() { val = 0; }
   void add(int value) { val += value; }
   // Make the reset() function a friend of this class
   friend void reset(Adder &Adder);
};
// reset() is now a friend of the Adder class
void reset(Adder &Adder) {
   // And can access the private data of Adder objects
   Adder.val = 0;
}
int main() {
   Adder acc;
   acc.add(5);  // add 5 to the Adder
   reset(acc);  // reset the Adder to 0
   return 0;
}
```

Points to Note:

We've declared a function named reset() that takes an object of class Adder, and sets the value of ***val*** to 0.

Because reset() is not a member of the Adder class, normally reset() would not be able to access the private members of Adder.

However, because Adder has specifically declared this reset() function to be a friend of the class, the reset() function is given access to the private members of Adder.

Note that we have to pass an Adder object to reset(). This is ***because reset() is not a member function***. It does not have a *this pointer, nor does it have an Adder object to work with, unless given one.

Multiple friends

A function can be a friend of more than one class at the same time. For example, consider the following example:

```
#include <iostream>
using namespace std;
class Contest;
class Course {
  private:
   int progress_course;
  public:
   Course(int temp = 0) { progress_course = temp; }
   friend void printProgress(const Course &course, const Contest &contest);
};

class Contest {
  private:
   int progress_contest;
```

```
  public:
   Contest(int contest = 0) { progress_contest = contest; }
   friend void printProgress(const Course &course, const Contest &contest);
};

void printProgress(const Course &course, const Contest &contest) {
   cout << "Course Progress is: " << course.progress_course << "\n"
        << "Contest Progress is: " << contest.progress_contest << '\n';
}

int main() {
   Contest ct(60);
   Course co(80);
   printProgress(co, ct);
   return 0;
}
```

Output:

```
Course Progress is: 80
Contest Progress is: 60
```

Points to Note:

Because printProgress is a friend of both classes, it can access the private data from objects of both classes.

Note the following line at the top of the example:

```
class Contest;
```

This is a class prototype that tells the compiler that we are going to define a class called ***Contest*** in the future. Without this line, the compiler would tell us it doesn't know what a Contest is when parsing the prototype for ***printProgress()*** inside the ***Course*** class.

Class prototypes serve the same role as function prototypes - they tell the compiler what something looks like so it can be used now and defined later.

However, unlike functions, classes have no return types or parameters, so class prototypes are always simply ***class ClassName***, where ClassName is the name of the class.

Friend Classes

It is also possible to make an entire class a friend of another class. This gives all of the ***members of the friend class access to the private members of the other class***.

```
#include <iostream>
using namespace std;
class Contest;
class Course {
  private:
   int progress_course;
  public:
   Course(int temp = 0) { progress_course = temp; }
   // Make Contest Class Friend of Course Class
   friend class Contest;
};

class Contest {
  private:
   int progress_contest;
  public:
   Contest(int contest = 0) { progress_contest = contest; }
   void printProgress(const Course &course, const Contest &contest) {
      cout << "Course Progress is: " << course.progress_course << "\n"
           << "Contest Progress is: " << contest.progress_contest << '\n';
   }
```

```
};

int main() {
   Contest ct(60);
   Course co(80);
   ct.printProgress(co, ct);
   return 0;
}
```

Output:

```
Course Progress is: 80
Contest Progress is: 60
```

Points to Note:

Because the ***Contest*** class is a friend of ***Course***, any of Contest's members that use a Course class object can access the private members of Course directly.

Even though Contest is a friend of Course, Contest has no direct access to the *this pointer of Course objects.

Just because Contest is a friend of Course, that does not mean Course is also a friend of Contest.

If you want two classes to be friends of each other, both must declare the other as a friend.

Finally, if class A is a friend of B, and B is a friend of C, that does not mean A is a friend of C.

Be careful when using friend functions and classes, because it allows the friend function or class to violate encapsulation.

If the details of the class change, the details of the friend will also be forced to change. Consequently, limit your use of friend functions and classes to a minimum.

OOPS

Function Overloading

After classes, perhaps the next most important feature of C++ is function overloading.

Two or more functions can share the ***same name*** as long as ***either the type of their arguments differs or the number of their arguments differs - or both***.

When two or more functions share the same name, they are said ***overloaded***.

Overloaded functions can help reduce the complexity of a program by allowing related operations to be referred to by the same name.

To overload a function, simply declare and define all required versions. ***The compiler will automatically select the correct version based upon the number and/or type of the arguments*** used to call the function. The following example illustrates the overloading of the absolute value function:

```
#include <iostream>
using namespace std;
// overload myabs three ways
int myabs(int n);
long myabs(long n);
double myabs(double n);
int main() {
   cout << "Abs value of -10: " << myabs(-10) << "\n";
   cout << "Abs value of -10L: " << myabs(-10L) << "\n";
   cout << "Abs value of -10.01:" << myabs(-10.01) << "\n";
   return 0;
}

// myabs() for ints
int myabs(int n) {
   cout << "In integer myabs()\n";
   return n < 0 ? -n : n;
}

// myabs() for long
long myabs(long n) {
   cout << "In long myabs()\n";
   return n < 0 ? -n : n;
}


// myabs() for double
double myabs(double n) {
   cout << "In double myabs()\n";
   return n < 0 ? -n : n;
}
```

The compiler automatically calls the correct version of the function based upon the type of data used as an argument.

Overloaded functions can also differ in the number of arguments. But, you must ***remember that the return type alone is not sufficient to allow function overloading***. If two functions differ only in the type of data they return, the compiler will not always be able to select the proper one to call. For example, the following fragment is ***incorrect***,

```
// This is incorrect and will not compile
int f1 (int a);
double f1 (int a);
f1(10); // which function does the compiler call???
```

NO NOTES TO SHOW

OOPS

Encapsulation

Encapsulation is one of the key features of object-oriented programming. It involves the ***bundling*** of data ***members*** and ***functions*** inside a single class.

Bundling similar data members and functions inside a class together also helps in ***data hiding***.

In general, encapsulation is a process of ***wrapping similar code*** in one place.

In C++, we can bundle data members and functions that operate together inside a single class. For example,

```
#include <bits/stdc++.h>
using namespace std;
class Cuboid {
  public:
   int length, breadth, height;
   // Constructor to initialize Values
   Cuboid(int l, int b, int h) : length(l), breadth(b), height(h) {}

   // Function to Calculate Volume
   int getVolume() { return length * breadth * height; }
};

int main() {
   Cuboid c(1, 2, 3);
   cout << "Volume is: " << c.getVolume();
}
```

In the above example, we are calculating the volume of a cuboid.

To calculate the volume, we need three variables: ***length***, ***height*** and ***breadth*** and a function: ***getVolume()***. Hence, we ***bundled*** these variables and function inside a single class named Cuboid.

Here, the variables and functions can be accessed from other classes as well. Hence, this is ***not data hiding***.

This is only ***encapsulation***. We are just keeping similar codes together.

> **Note:** People often consider encapsulation as data hiding, but that's not entirely true.
>
> Encapsulation refers to the bundling of related fields and methods together. This can be used to achieve data hiding. ***Encapsulation in itself is not data hiding.***

Why Encapsulation?

In C++, encapsulation helps us keep ***related data and functions together***, which makes our code cleaner and easy to read.
It helps to control the modification of our data members.

Consider a situation where we want the length field in a class to be non-negative. Here we can make the length variable private and apply the logic inside the method ***setLength()***. For example,

```
#include <bits/stdc++.h>
using namespace std;
class Cuboid {
  private:
   int length;

  public:
   void setlength(int len) {
      if (len >= 0) length = len;
   }
};

int main() {
```

```
   Cuboid c;
   c.setlength(7);
   return 0;
}
```

The getter and setter functions provide read-only or write-only access to our class members. For example,

```
getLength()  // provides read-only access
setLength()  // provides write-only access
```

It helps to decouple components of a system. For example, we can encapsulate code into multiple bundles. These decoupled components (bundles) can be developed, tested, and debugged independently and concurrently. And any changes in a particular component do not have any effect on other components.

We can also ***achieve data hiding using encapsulation***. In Example 1, if we change the length, breadth and height variables into ***private*** or ***protected***, then the access to these variables is restricted. And, they are kept hidden from outer classes. This is called ***data hiding***.

Data Hiding

Data hiding is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding.

We can use access modifiers to achieve data hiding in C++. For example,

```
#include <bits/stdc++.h>
using namespace std;
class Cuboid {
   int length, breadth, height;

  public:
   // Setter functions
   void setLength(int l){length = l;}
   void setBreadth(int b){breadth = b;}
   void setHeight(int h){height = h;}

   // Getter Functions
   int getLength() { return length; }
   int getBreadth() { return breadth; }
   int getHeight() { return height; }

   // Function to Calculate Volume
   int getVolume() { return length * breadth * height; }
};

int main() {
   Cuboid c;
   c.setLength(1);
   c.setBreadth(2);
   c.setHeight(3);

   cout << "Length: " << c.getLength() << "\n";
   cout << "Breadth: " << c.getBreadth() << "\n";
   cout << "Height: " << c.getHeight() << "\n";
   cout << "Volume is: " << c.getVolume();
}
```

Output:

```
Length: 1
Breadth: 2
Height: 3
Volume is: 6
```

Here, we have made the length, breadth and height variables private.

This means that these variables cannot be directly accessed outside of the Cuboid class.

To access these private variables, we have used public functions ***setLength(), getLength(), setBreadth(), getBreadth(), setHeight(), getHeight()***. These are called ***getter and setter functions***.

Making the variables private allowed us to ***restrict unauthorized access from outside the class***. This is ***data hiding***.

If we try to access the variables from the ***main()*** class, we will get an error.

```
// error: c.length is inaccessible
c.length = 8;

// error: c.breadth is inaccessible
c.breadth = 6;
```

OOPS

Abstraction

Data abstraction refers to providing only ***essential*** information to the outside world and hiding their background details, 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 a 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 <bits/stdc++.h>
using namespace std;

int main() {
   cout << "Hello World!" << 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.

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.

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

```
#include <bits/stdc++.h>
using namespace std;
class Cuboid {
   int length, breadth, height;
```

```
  public:
   // Setter functions
   void setLength(int l){length = l;}
   void setBreadth(int b){breadth = b;}
   void setHeight(int h){height = h;}

   // Getter Functions
   int getLength() { return length; }
   int getBreadth() { return breadth; }
   int getHeight() { return height; }

   // Function to Calculate Volume
   int getVolume() { return length * breadth * height; }
};

int main() {
   Cuboid c;
   c.setLength(1);
   c.setBreadth(2);
   c.setHeight(3);

   cout << "Length: " << c.getLength() << "\n";
   cout << "Breadth: " << c.getBreadth() << "\n";
   cout << "Height: " << c.getHeight() << "\n";
   cout << "Volume is: " << c.getVolume();
}
```

Above class returns the volume. The public member - **getVolume()** is the interface to the outside world and a user needs to know it to use the class. The private members ***length, breadth and height*** are something that the user doesn't need to know about, but is needed for the class to operate properly.

Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.

In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.

OOPS

Inheritance

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 class access control

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

```
class derived-class-name : access base-class-name {
    // ...
}
```

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

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

When the access specifier for the inherited base class is public all public members of the base class become public members of the derived class.

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

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

> It is important to understand that if the access specifier is private public members of the base become private members of the derived class, but these are still accessible by member functions of the derived class.

If the access specifier is not present, it is private by default if the derived class is a class. If the derived class is a struct, public is the default access.

```
#include <iostream>

using namespace std;
class base {
   int x;

  public:
   void setx(int n) { x = n; }

   void showx() { cout << x << "\n"; }
};
// Inherit as public
class derived : public base {
   int y;

  public:
   void sety(int n) { y = n; }

   void showy() { cout << y << "\n"; }
};
int main() {
   derived ob;
   ob.setx(10);  // access member of base class
   ob.sety(20);  // access member of derived class
```

```
   ob.showx();   // access member of base class
   ob.showy();   // access member of derived class
   return 0;
}
```

Output

```
10
20
```

Here the variation of the program, this time derived inherits base as private, which causes error.

```
// This program contains an error
#include <iostream>

using namespace std;
class base {
   int x;

  public:
   void setx(int n) { x = n; }

   void showx() { cout << x << "\n"; }
};
// Inherit base as private
class derived : private base {
   int y;

  public:
   void sety(int n) { y = n; }

   void showy() { cout << y << "\n"; }
};
int main() {
   derived ob;
   ob.setx(10);  // ERROR now private to derived class
   ob.sety(20);  // access member of derived class - OK
   ob.showx();   // ERROR now private to derived class
   ob.showy();   // access member of derived class - OK
   return 0;
}
```

As the comments in this (incorrect) program illustrates, both showx() and setx() become private to derived and are not accessible outside it.

In other words, ***they are accessible from within the derived class***. Keep in mind that ***showx()*** and ***setx()*** are still public within base, no matter how they are inherited by some derived class.

This means that an object of type base could access these functions anywhere.

Using protected members : As you know from the preceding section, a derived class does not have access to the private members of the base class. However, there will be times when you want to keep a member of a base class private but still allow a derived class access to it. To accomplish this, C++ includes the protected access specifier. The protected access specifier is equivalent to the private specifier with the sole exception that protected members of a base class are accessible to members of any class derived from that base. Outside the base or derived classes, protected members are not accessible.

The protected access specifier can occur any where in the class declaration, although typically it occurs after the (default) private members are declared and before the public members.

The full general form of a class declaration is shown here:

```
class class-name {
    // private members
    protected: //optional
    // protected members
```

```
    public:
    //public members
};
```

When a protected member of a base class is inherited as public by the derived class, it becomes a protected member of the derived class.

If the base class is inherited as private, a protected member of the base becomes a private member of the derived class.

A base class can also be inherited as protected by a derived class. When this is the case, public and protected members of the base class become protected members of the derived class (of course, private members of the base remain private to it and are not accessible by the derived class).

The following program illustrates what occurs when protected members are inherited as public:

```
#include <iostream>

using namespace std;
class base {
  protected:  // private to base
   int a, b;  // but still accessible by derived
  public:
   void setab(int n, int m) {
      a = n;
      b = m;
   }
};
// Inherit as public
class derived : public base {
   int c;

  public:
   void setc(int n) { c = n; }
   // this function has access to a and b from base
   void showabc() { cout << a << " " << b << " " << c << "\n"; }
};
int main() {
   derived ob;
   // a and b are not accessible here because they
   // are
   // private to both base and derived.
   ob.setab(1, 2);
   ob.setc(3);
   ob.showabc();
   return 0;
}
```

Output:

```
1 2 3
```

If base is inherited as protected, its public and protected members become protected members of derived and therefore are not accessible within the main( ), i.e. the statement:

```
ob.setab(1, 2);// would create a compile-time error.
```

OOPS

Constructors, destructors, and inheritance

It is possible for the base class, the derived class, or both to have constructor and/or destructor functions. When a base class and a derived class both have constructor and destructor functions, the constructor functions are executed in order of derivation. **The destructor functions are executed in reverse order.**

That is, the base constructor is executed before the constructor in the derived class.

The reverse is true for destructor functions: **the destructor in the derived class is executed before the base class destructor**.

So far, you have passed arguments to either the derived class or base class constructor. When only the derived class takes an initialization, arguments are passed to the derived class constructor in the normal fashion.

However, if you need to pass an argument to the constructor of the base class, a little more effort is needed:

All necessary arguments to both the base class and derived class are passed to the derived class constructor.

Using an expanded form of the derived class' constructor declaration, you then pass the appropriate arguments along to the base class

The syntax for passing an argument from the derived class to the base class is as

```
derived-constructor(arg-list) : base(arg-list) {
     // body of the derived class constructor
}
```

Here base is the name of the base class. It is permissible for both the derived class and the base class to use the same argument. It is also possible for the derived class to ignore all arguments and just pass them along to the base.

```
// Illustrates when base class and derived class
// constructor and destructor functions are executed
#include <iostream>
using namespace std;
class base {
  public:
   base() { cout << "Constructing base\n"; }
   ~base() { cout << "Destructing base\n"; }
};
class derived : public base {
  public:
   derived() { cout << "Constructing derived\n"; }
   ~derived() { cout << "Destructing derived\n"; }
};
int main() {
   derived obj;
   return 0;
}
```

This program displays

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

In the following example both the derived class and the base class take arguments:

```
#include <iostream>
using namespace std;
class base {
   int i;

  public:
   base(int n) {
```

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

  public:
   derived(int n) : base(n) {
      // pass argument to the base class
      cout << "Constructing derived\n";
      j = n;
   }
   ~derived() { cout << "Destructing derived\n"; }
   void showj() { cout << j << "\n"; }
};
int main() {
   derived o(10);
   o.showi();
   o.showj();
   return 0;
}
```

Pay special attention to the declaration of the derived class constructor. Notice how the parameter n (which receives the initialization argument) is both used by derived() and base().

OOPS

Multiple Inheritance

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

First, a derived class can be used as a base class for another derived class, creating a multilevel class hierarchy. In this case, the original base class is said to be an indirect base class of the second derived class.

Second, a derived class can directly inherit more than one base class. In this situation, two or more base class are combined to help create the derived class.

There several issues that arise when multiple base classes are involved. Those will be discussed in this section. If a class B1 is inherited by a class D1, and D1 is inherited by a class D2, the constructor functions of all the three classes are called in order of derivation. Also the destructor functions are called in reverse order.

When a derived class inherits multiple base classes, it uses the expanded declaration:

```
class derived-class-name : access base1, access base2, ... , access baseN
{
     // ... body of class
};
```

Here **base1** through **baseN** are the base class names and access the access specifier, which can be different for each base class.

When multiple base classes are inherited, constructors are executed in the order, left to right that the base classes are specified. Destructors are executed in reverse order.

When a class inherits multiple base classes that have constructors that requires arguments, the derived class pass the necessary arguments to them by using this expanded form class constructor function:

```
derived-constructor(arg-list) : base1(arg-list) ,..., baseN(arg-list)
{
     // body of derived class constructor
}
```

Here base1 through baseN are the names of the classes.

Virtual base classes

A potential problem exists when multiple base classes are directly inherited by a derived class.

To understand what this problem is, consider the following class hierarchy:

Base class Base is inherited by both Derived1 and Derived2. Derived3 directly inherits both Derived1 and Derived2.

However, this implies that Base is actually **inherited twice** by Derived3. First it is inherited through Derived1, and then again through Derived2. This causes ambiguity when a member of Base is used by Derived3. Since two copies of Base are included in Derived3, is a reference to a member of Base referring to the Base inherited indirectly through Derived1 or to the Base inherited indirectly through Derived2?

To resolve this ambiguity, C++ includes a mechanism by which only one copy of Base will be included in Derived3. This feature is called a ***virtual base class***

In situations like this, in which a derived class indirectly inherits the same base class more than once, it is possible to prevent multiple copies of the base from being present in the derived class by having that base class inherited as virtual by any derived classes. Doing this prevents two or more copies of the base from being present in any subsequent derived class that inherits the base class indirectly. The virtual keyword precedes the base class access specifier when it is inherited by a derived class.

Diamond Problem - Programmer and Software Interview Questions and Answers

```
// This program uses a virtual base class.
#include <iostream>

using namespace std;
class Base {
  public:
   int i;
};
```

```
// Inherit Base as virtual
class Derived1 : virtual public Base {
  public:
   int j;
};

// Inherit Base as virtual here, too
class Derived2 : virtual public Base {
  public:
   int k;
};

// Here Derived3 inherits both Derived1 and Derived2.
// However, only one copy of base is inherited.
class Derived3 : public Derived1, public Derived2 {
  public:
   int product() { return i * j * k; }
};

int main() {
   Derived3 ob;
   ob.i = 10;  // unambiguous because virtual Base
   ob.j = 3;
   ob.k = 5;
   cout << "Product is: " << ob.product() << "\n";
   return 0;
}
```

Output:

```
Product is: 150
```

If Derived1 and Derived2 had not inherited Base as virtual, the statement ***ob.i=10*** would have been ambiguous and a compile-time error would have resulted.

OOPS

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 <bits/stdc++.h>
using namespace std;

class Base {
  protected:
   int a, b;

  public:
   Base(int _a, int _b) {
      a = _a;
      b = _b;
   }
   void func() { cout << "Inside Parent Class" << endl; }
};

class Derived1 : public Base {
  public:
   Derived1(int a, int b) : Base(a, b) {}

   void func() { cout << "Inside Derived1 Class" << endl; }
};

class Derived2 : public Base {
  public:
   Derived2(int a, int b) : Base(a, b) {}

   void func() { cout << "Inside Derived2 Class" << endl; }
};

// Main function for the program
int main() {
   Base *b;
   Derived1 d1(10, 7);
   Derived2 d2(10, 5);

   // Store the address of Derived1
   b = &d1;

   // Call Derived1 Func.
   b->func();

   // Store the address of Derived2
   b = &d2;

   // Call Derived2 Func.
   b->func();
```

```
   return 0;
}
```

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

```
Inside Parent Class
Inside Parent Class
```

The reason for the incorrect output is that the call of the function ***func()*** 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 ***func()*** 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 ***func()*** in the Base class with the keyword **virtual** so that it looks like below

```
class Base {
  protected:
   int a, b;

  public:
   Base(int _a, int _b) {
      a = _a;
      b = _b;
   }
   virtual void func() { cout << "Inside Parent Class" << endl; }
};
```

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

```
Inside Derived1 Class
Inside Derived2 Class
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of d1 and d2 classes are stored in *b the respective func() function is called.

As you can see, each of the child classes has a separate implementation for the function func(). 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.

Virtual Functions

A ***virtual*** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be ***based on the kind of object for which it is called***. This sort of operation is referred to as ***dynamic linkage***, or ***late binding***.

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function ***func()*** in the base class to the following

```
class Base {
  protected:
   int a, b;

  public:
   Base(int _a, int _b) {
      a = _a;
      b = _b;
   }
   virtual void func() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called ***pure virtual function***.

NO NOTES TO SHOW