CONCEPTS OF OOP

There are a few principle concepts that form the foundation of object-oriented programming −

## Object

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

## Class

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

## 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.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

## Encapsulation

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

## Inheritance

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

## Polymorphism

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

## Overloading

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

### Class and Objects

A class is like a blueprint of data member and functions and object is an instance of class. For example, lets say we have a class **Car** which has data members (variables) such as speed, weight, price and functions such as gearChange(), slowDown(), brake() etc. Now lets say I create a object of this class named FordFigo which uses these data members and functions and give them its own values. Similarly we can create as many objects as we want using the blueprint(class).

//Class name is Car

class Car

{

//Data members

char name[20];

int speed;

int weight;

public:

//Functions

void brake(){

}

void slowDown(){

}

};

int main()

{

//ford is an object

Car ford;

}

### Abstraction

[Abstraction](https://beginnersbook.com/2017/09/abstraction-in-c-with-example/) is a process of hiding irrelevant details from user. For example, When you send an sms you just type the message, select the contact and click send, the phone shows you that the message has been sent, what actually happens in background when you click send is hidden from you as it is not relevant to you.

# Abstraction in C++ with example

Abstraction is one of the feature of [Object Oriented Programming](https://beginnersbook.com/2017/08/cpp-oops-concepts/), where you show only relevant details to the user and hide irrelevant details. For example, when you send an email to someone you just click send and you get the success message, what actually happens when you click send, how data is transmitted over network to the recipient is hidden from you (because it is irrelevant to you).

Let’s see how this can be achieved in a C++ program using access specifiers:

## Abstraction Example

#include <iostream>

using namespace std;

class AbstractionExample{

private:

/\* By making these data members private, I have

\* hidden them from outside world.

\* These data members are not accessible outside

\* the class. The only way to set and get their

\* values is through the public functions.

\*/

int num;

   char ch;

public:

void setMyValues(int n, char c) {

num = n; ch = c;

}

void getMyValues() {

cout<<"Numbers is: "<<num<< endl;

cout<<"Char is: "<<ch<<endl;

}

};

int main(){

AbstractionExample obj;

obj.setMyValues(100, 'X');

obj.getMyValues();

return 0;

}

**Output:**

Numbers is: 100

Char is: X

## Advantage of data abstraction

The major advantage of using this feature is that when the code evolves and you need to make some adjustments in the code then you only need to modify the high level class where you have declared the members as private. Since none class is accessing these data members directly, you do not need to change the low level(user level) class code.  
Imagine if you had made these data members public, if at some point you want to change the code, you would have to make the necessary adjustments to all the classes that are accessing the members directly.

**Other advantages of data abstraction are:**  
1) Makes the application secure by making data private and avoiding the user level error that may corrupt the data.  
2) This avoids code duplication and increases the code reusability.

# Inheritance in C++

Inheritance is one of the feature of [Object Oriented Programming System(OOPs)](https://beginnersbook.com/2017/08/cpp-oops-concepts/), it allows the child class to acquire the properties (the data members) and functionality (the member functions) of parent class.

**What is child class?**  
A class that inherits another class is known as child class, it is also known as derived class or subclass.  
**What is parent class?**  
The class that is being inherited by other class is known as parent class, super class or base class.

#### Syntax of Inheritance

class parent\_class

{

//Body of parent class

};

class child\_class : access\_modifier parent\_class

{

//Body of child class

};

## What are the advantages of using inheritance in C++ Programming

The main advantages of inheritance are **code reusability** and **readability**. When child class inherits the properties and functionality of parent class, we need not to write the same code again in child class. This makes it easier to reuse the code, makes us write the less code and the code becomes much more readable.

Lets take a **real life example** to understand this: Lets assume that Human is a class that has properties such as height, weight, colour etc and functionality such as eating(), sleeping(), dreaming(), working() etc.  
Now we want to create Male and Female class, these classes are different but since both Male and Female are humans they share some common properties and behaviours (functionality) so they can inherit those properties and functionality from Human class and rest can be written in their class separately.  
This approach makes us write less code as both the classes inherited several properties and functions from base class thus we didn’t need to re-write them. Also, this makes it easier to read the code.

## Inheritance Example

Before we discuss the types of inheritance, lets take an example:  
Here we have two classes Teacher and MathTeacher, the MathTeacher class inherits the Teacher class which means Teacher is a parent class and MathTeacher is a child class. The child class can use the property collegeName of parent class.

Another important point to note is that when we create the object of child class it calls the constructor of child class and child class constructor automatically calls the constructor of base class.

#include <iostream>

using namespace std;

class Teacher {

public:

Teacher(){

cout<<"Hey Guys, I am a teacher"<<endl;

}

string collegeName = "Beginnersbook";

};

//This class inherits Teacher class

class MathTeacher: public Teacher {

public:

MathTeacher(){

cout<<"I am a Math Teacher"<<endl;

}

string mainSub = "Math";

string name = "Negan";

};

int main() {

MathTeacher obj;

cout<<"Name: "<<obj.name<<endl;

cout<<"College Name: "<<obj.collegeName<<endl;

cout<<"Main Subject: "<<obj.mainSub<<endl;

return 0;

}

**Output:**

Hey Guys, I am a teacher

I am a Math Teacher

Name: Negan

College Name: Beginnersbook

Main Subject: Math

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.

## Types of Inheritance in C++

1) Single inheritance  
2) Multilevel inheritance  
3) Multiple inheritance  
4) Hierarchical inheritance  
5) Hybrid inheritance

### Single inheritance

In Single inheritance one class inherits one class exactly.  
For example: Lets say we have class A and B

B inherits A

**Example of Single inheritance:**

#include <iostream>

using namespace std;

class A {

public:

A(){

cout<<"Constructor of A class"<<endl;

}

};

class B: public A {

public:

B(){

cout<<"Constructor of B class";

}

};

int main() {

//Creating object of class B

B obj;

return 0;

}

Output:

Constructor of A class

Constructor of B class

### 2)Multilevel Inheritance

In this type of inheritance one class inherits another child class.

C inherits B and B inherits A

**Example of Multilevel inheritance:**

#include <iostream>

using namespace std;

class A {

public:

A(){

cout<<"Constructor of A class"<<endl;

}

};

class B: public A {

public:

B(){

cout<<"Constructor of B class"<<endl;

}

};

class C: public B {

public:

C(){

cout<<"Constructor of C class"<<endl;

}

};

int main() {

//Creating object of class C

C obj;

return 0;

}

Output:

Constructor of A class

Constructor of B class

Constructor of C class

### Multiple Inheritance

In multiple inheritance, a class can inherit more than one class. This means that in this type of inheritance a single child class can have multiple parent classes.  
For example:

C inherits A and B both

**Example of Multiple Inheritance:**

#include <iostream>

using namespace std;

class A {

public:

A(){

cout<<"Constructor of A class"<<endl;

}

};

class B {

public:

B(){

cout<<"Constructor of B class"<<endl;

}

};

class C: public A, public B {

public:

C(){

cout<<"Constructor of C class"<<endl;

}

};

int main() {

//Creating object of class C

C obj;

return 0;

}

Constructor of A class

Constructor of B class

Constructor of C class

### 4)Hierarchical Inheritance

In this type of inheritance, one parent class has more than one child class. For example:

Class B and C inherits class A

**Example of Hierarchical inheritance:**

#include <iostream>

using namespace std;

class A {

public:

A(){

cout<<"Constructor of A class"<<endl;

}

};

class B: public A {

public:

B(){

cout<<"Constructor of B class"<<endl;

}

};

class C: public A{

public:

C(){

cout<<"Constructor of C class"<<endl;

}

};

int main() {

//Creating object of class C

C obj;

return 0;

}

Output:

Constructor of A class

Constructor of C class

### 5) Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance. For example, A child and parent class relationship that follows multiple and hierarchical inheritance both can be called hybrid inheritance.

**Abstraction in C++ with example**

Abstraction is one of the feature of [Object Oriented Programming](https://beginnersbook.com/2017/08/cpp-oops-concepts/), where you show only relevant details to the user and hide irrelevant details. For example, when you send an email to someone you just click send and you get the success message, what actually happens when you click send, how data is transmitted over network to the recipient is hidden from you (because it is irrelevant to you). Let’s see how this can be achieved in a C++ program using access specifiers:

**Abstraction Example**

#include <iostream>

using namespace std;

class AbstractionExample{

private:

/\* By making these data members private, I have

\* hidden them from outside world.

\* These data members are not accessible outside

\* the class. The only way to set and get their

\* values is through the public functions.

\*/

int num;

   char ch;

public:

void setMyValues(int n, char c) {

num = n; ch = c;

}

void getMyValues() {

cout<<"Numbers is: "<<num<< endl;

cout<<"Char is: "<<ch<<endl;

}

};

int main(){

AbstractionExample obj;

obj.setMyValues(100, 'X');

obj.getMyValues();

return 0;

}

**Output:**

Numbers is: 100

Char is: X

**Advantage of data abstraction**

The major advantage of using this feature is that when the code evolves and you need to make some adjustments in the code then you only need to modify the high level class where you have declared the members as private. Since none class is accessing these data members directly, you do not need to change the low level(user level) class code.  
Imagine if you had made these data members public, if at some point you want to change the code, you would have to make the necessary adjustments to all the classes that are accessing the members directly.

**Other advantages of data abstraction are:**  
1) Makes the application secure by making data private and avoiding the user level error that may corrupt the data.  
2) This avoids code duplication and increases the code reusability.

# Encapsulation in C++ with example

Encapsulation is a process of combining data members and functions in a single unit called class. This is to prevent the access to the data directly, the access to them is provided through the functions of the class. It is one of the popular feature of [Object Oriented Programming(OOPs)](https://beginnersbook.com/2017/08/cpp-oops-concepts/) that helps in **data hiding**.

## How Encapsulation is achieved in a class

To do this:  
1) Make all the data members private.  
2) Create public setter and getter functions for each data member in such a way that the set function set the value of data member and get function get the value of data member.

Let’s see this in an example Program:

## Encapsulation Example in C++

Here we have two data members num and ch, we have declared them as private so that they are not accessible outside the class, this way we are hiding the data. The only way to get and set the values of these data members is through the public getter and setter functions.

#include<iostream>

using namespace std;

class ExampleEncap{

private:

/\* Since we have marked these data members private,

\* any entity outside this class cannot access these

\* data members directly, they have to use getter and

\* setter functions.

\*/

int num;

char ch;

public:

/\* Getter functions to get the value of data members.

\* Since these functions are public, they can be accessed

\* outside the class, thus provide the access to data members

\* through them

\*/

int getNum() const {

return num;

}

char getCh() const {

return ch;

}

/\* Setter functions, they are called for assigning the values

\* to the private data members.

\*/

void setNum(int num) {

this->num = num;

}

void setCh(char ch) {

this->ch = ch;

}

};

int main(){

ExampleEncap obj;

obj.setNum(100);

obj.setCh('A');

cout<<obj.getNum()<<endl;

cout<<obj.getCh()<<endl;

return 0;

}

**Output:**

100

A

### Polymorphism

[Function overloading](https://beginnersbook.com/2017/08/cpp-function-overloading/) and Operator overloading are examples of polymorphism. Polymorphism is a feature using which an object behaves differently in different situation.  
In function overloading we can have more than one function with same name but different numbers, type or sequence of arguments.

#### Polymorphism Example

#include <iostream>

using namespace std;

class Sum {

public:

int add(int num1,int num2){

return num1 + num2;

}

int add(int num1, int num2, int num3){

return num1 + num2 + num3;

}

};

int main(void) {

//Object of class Sum

Sum obj;

//This will call the second add function

cout<<obj.add(10, 20, 30)<<endl;

//This will call the first add function

cout<<obj.add(11, 22);

return 0;

}

**Output:**

60

33

# Function overloading in C++

Function overloading is a [C++ programming](https://beginnersbook.com/2017/08/c-plus-plus-tutorial-for-beginners/) feature that allows us to have more than one function having same name but different parameter list, when I say parameter list, it means the data type and sequence of the parameters, for example the parameters list of a function myfuncn(int a, float b) is (int, float) which is different from the function myfuncn(float a, int b) parameter list (float, int). Function overloading is a [compile-time polymorphism](https://beginnersbook.com/2017/08/cpp-polymorphism/).  
Now that we know what is parameter list lets see the rules of overloading: we can have following functions in the same scope.

sum(int num1, int num2)

sum(int num1, int num2, int num3)

sum(int num1, double num2)

The easiest way to remember this rule is that the parameters should qualify any one or more of the following conditions, they should have different **type**, **number** or **sequence** of parameters.

**For example:**  
These two functions have different parameter **type**:

sum(int num1, int num2)

sum(double num1, double num2)

These two have different **number** of parameters:

sum(int num1, int num2)

sum(int num1, int num2, int num3)

These two have different **sequence** of parameters:

sum(int num1, double num2)

sum(double num1, int num2)

All of the above three cases are valid case of overloading. We can have any number of functions, just remember that the parameter list should be different. For example:

int sum(int, int)

double sum(int, int)

This is not allowed as the parameter list is same. Even though they have different return types, its not valid.

## Function overloading Example

Lets take an example to understand function overloading in C++.

#include <iostream>

using namespace std;

class Addition {

public:

    int sum(int num1,int num2) {

        return num1+num2;

    }

    int sum(int num1,int num2, int num3) {

    return num1+num2+num3;

    }

};

int main(void) {

Addition obj;

    cout<<obj.sum(20, 15)<<endl;

    cout<<obj.sum(81, 100, 10);

   return 0;

}

**Output:**

35

191

## Function overloading Example 2

As I mentioned in the beginning of this guide that functions having different return types and same parameter list cannot be overloaded. However if the functions have different parameter list then they can have same or different return types to be eligible for overloading. In short the return type of a function  
does not play any role in function overloading. All that matters is the parameter list of function.

#include <iostream>

using namespace std;

class DemoClass {

public:

    int demoFunction(int i) {

        return i;

    }

    double demoFunction(double d) {

    return d;

    }

};

int main(void) {

DemoClass obj;

    cout<<obj.demoFunction(100)<<endl;

    cout<<obj.demoFunction(5005.516);

   return 0;

}

**Output:**

100

5006.52

## Advantages of Function overloading

The main advantage of function overloading is to the improve the **code readability** and allows **code reusability**. In the example 1, we have seen how we were able to have more than one function for the same task(addition) with different parameters, this allowed us to add two integer numbers as well as three integer numbers, if we wanted we could have some more functions with same name and four or five arguments.  
Imagine if we didn’t have function overloading, we either have the limitation to add only two integers or we had to write different name functions for the same task addition, this would reduce the code readability and reusability.