Suggestions – OOP

What is object-oriented programming?

Object-oriented programming (OOP) is a popular technique to solve programming problems by creating objects.

Let's try to understand it with an example.

Suppose we need to store the name and the test score of university students. And based on the test score, we need to find if a student passed or failed the examination. Then, the structure of our code would look something like this.

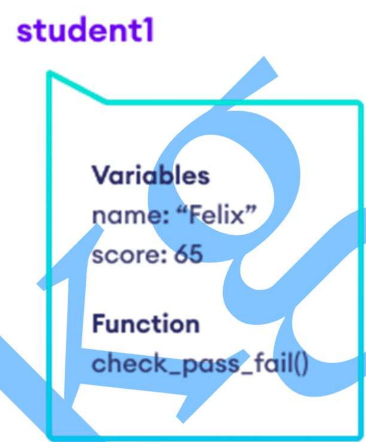


Figure: Code Structure

Now, imagine we have to store the name and the test score of multiple students instead of one student.

If we were to use the same approach, we can use the same `check_pass_fail()` function.

However, we would need to create multiple variables to store the `name` and the `score` for each student. This would make our code less organized and messy.

There are two steps involved in creating objects:

1. Define a class
2. Create objects from the class

Define a Class

To solve the problem, we will first define a class named Student.

Figure: The Student Class

This `Student` class has two variables `name` and `score`, and a function `check_pass_fail()`.

💡
Think of a class as a blueprint for a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions, we can build a house. The actual physical house is the object.

Now, let's see how we can create objects.

Creating Objects

Once we define a class, we can create as many objects as we want from the class.

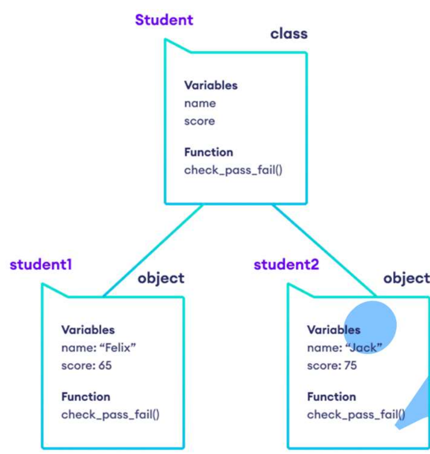


Figure: Classes and Objects

In the image, we have created objects `student1` and `student2` from the `Student` class.

All the objects of this `Student` class will have their own `name` and `score` variables and can use the `check_pass_fail()` function.

💡
Note: The variables and functions of a class are called class members. The variables are called member variables or data members, and the functions are called member functions.

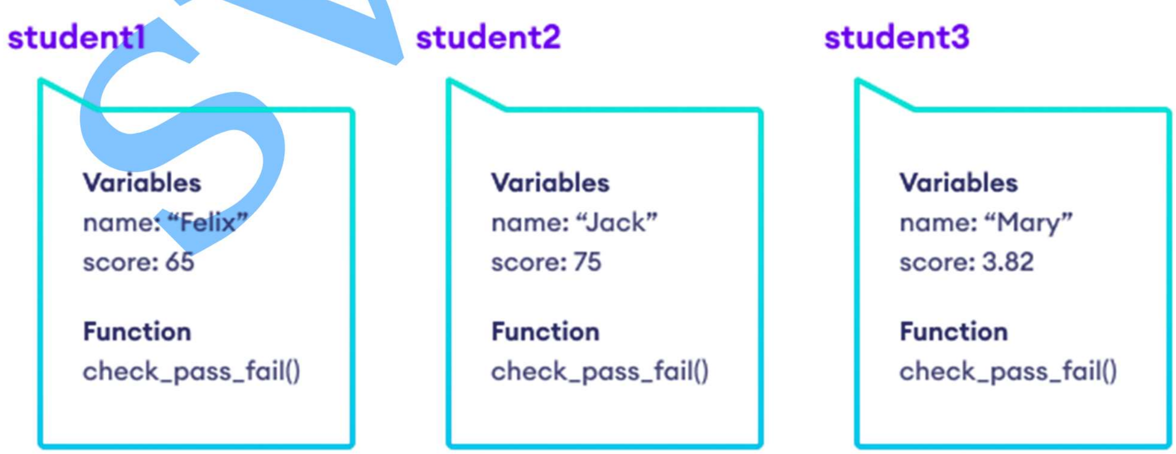


Figure: Code Structure

This approach to creating objects to solve problems is known as object-oriented programming.

Example

1. Create a Class

```
class Rectangle {
// code
};
```

Here, `Rectangle` is the name of the class. A class can contain data members such as variables (to store data) and member functions (to perform operations). Collectively, they are known as class members.

```
class Rectangle {
public:
// data members
int length, breadth;
// member function
void calculate_area(){
int area = length * breadth;
cout << "Area: " << area;
}
};
```

2. Create Objects

Here's how we create objects in C++.

```
Rectangle rectangle1;
```

Now we can use the `rectangle1` object to access the class members. For example,

```
#include <iostream>
using namespace std;
class Rectangle {
public:
// data members
int length, breadth;
// member function
void calculate_area(){
int area = length * breadth;
cout << "Area: " << area << endl;
}
};
int main() {
// create object of the Rectangle class
Rectangle rectangle1;
// assign values to length and breadth
rectangle1.length = 12;
rectangle1.breadth = 5;
// call the member function
```

```
rectangle1.calculate_area();
return 0;
}
```

Output

```
Area: 60
```

Basic Features of OOP

- Class
- Objects
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

Constructors

C++ Constructors

In C++, a constructor is similar to a member function, but it doesn't have a return type, and it has the same name as the class. For example,

```
class Student {

public:

// constructor

Student() {

...

}

// member function

void check_name() {

...

}
```

```
};
```

In the above example, `Student()` is a constructor and `check_name()` is a member function. You can see that the constructor doesn't have a return type, and it has the same name as the class (`Student`).

In C++, the constructor is called automatically when we create an object. Let's see an example,

```
#include <iostream>
using namespace std;
class Student {
public:
// constructor
Student() {
cout << "Calling Constructor" << endl;
}
};
int main() {
// create an object
Student student1;
return 0;
}
```

Output

```
Calling Constructor
```

Here, the code `Student student1;` calls the constructor. That's why we get the output.

Types of Constructors

There are broadly two types of constructors in C++. They are

- Default Constructors
- Parameterized Constructors

Default Constructors

In C++, a default constructor is a constructor that has no parameters, and thus takes no arguments. The constructors we've been dealing with so far are all default constructors.

Let's see an example,

```
#include <iostream>
using namespace std;
class Student {
public:
int marks;
// default constructor
Student() {
marks = 0;
}
};
int main() {
// create an object
Student student1;
// print the value of marks
cout << "Marks: " << student1.marks << endl;
return 0;
}
// Output: Marks: 0
```

Here, the `Student()` constructor doesn't take any argument. Hence, it's a default constructor.

Parameterized Constructors

As mentioned earlier, a parameterized constructor takes in arguments. We use this type of constructor to assign values to member variables for different objects.

Let's explore this with an example.

```
class Car {

public:

int gear;

// parameterized constructor

Car(int gear_no) {

gear = gear_no;

}

};
```

Here, `Car()` is a parameterized constructor that accepts a single parameter, `gear_no`.

Calling Parameterized Constructor

Just like any other constructor, a parameterized constructor is also called while creating objects. However, during the object creation, we pass arguments to the constructor. For example,

```
// call constructor

Car car1(5);

Car car2(6);
```

Here, the value of `gear_no` will be

- 5 for the object `car1`
- 6 for the object `car2`

Let's clarify this by writing a complete program.

```
#include <iostream>
using namespace std;
class Car {
public:
int gear;
// parameterized constructor to initialize gear
Car(int gear_no) {
gear = gear_no;
}
};
int main() {
// create objects of Car: car1 and car2
// pass 5 and 6 as arguments to constructors
// of car1 and car2 respectively
Car car1(5);
Car car2(6);
// print values of gear for car1 and car2
cout << "Gear for car1: " << car1.gear << endl;
cout << "Gear for car2: " << car2.gear << endl;
return 0;
}
```

Output

```
Gear for car1: 5
Gear for car2: 6
```

In the above example, we have used the parameterized constructor to assign the values of the `gear` data member.

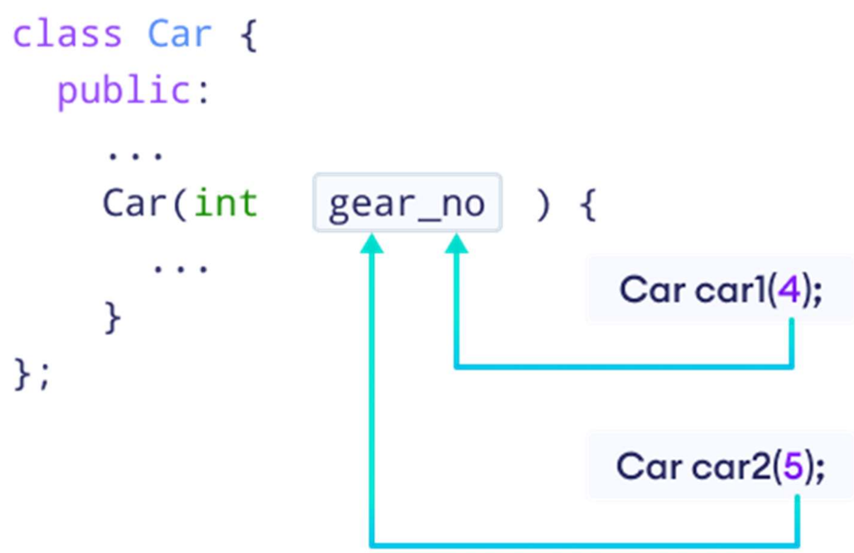


Figure: Passing different arguments to the constructor using different objects

💡

Going Forward: Because constructors are executed automatically when we create an object, they are thus excellent tools for initializing member variables.

Constructor Initializer List

In C++ constructors, we can also use an initialization list to initialize member variables. This will make our code look cleaner and more efficient. Let's see an example,

Suppose we are initializing the `name` and `score` variables using a constructor like this:

```
class Student {
    public:

        string name;
        int score;

       // constructor to initialize values
       Student (string sudent_name, int student_score) {
           name = student_name;
           score = student_score;
        }
};
```

Now let's see how we can do this using the initialization list.

```
class Student {
    public:

        string name;
        int score;

        // constructor to initialize values
        Student(string n, int s): name(n), score(s) {}
};
```

We can see this code now looks cleaner. Here,

- `n` and `s` are values passed to the constructor.
- `n` is assigned to the variable `name`.
- `s` is assigned to the variable `score`.

Copy Constructor

A **copy constructor** is a member function that initializes an object using another object of the same class.

Types of Copy Constructors:

1. Default Copy Constructor

2. User-defined Copy Constructor

1. Default Copy Constructor

When a copy constructor is not defined, the C++ compiler automatically supplies with its self-generated constructor that copies the values from the old object to the new object.

```
#include <iostream>

using namespace std;

 class A {

 int x, y;

public:

A(int i, int j){

x = i;

y = j;

}

int getX() {

return x;

}

int getY(){
```

```
return y;

}

};


int main() {

A ob1(10, 46);

A ob2 = ob1;

cout << "x = " << ob2.getX() << " y = " << ob2.getY();

 return 0;

 }
```

2. User-defined copy constructor

In case of a user-defined copy constructor, the values of the old object of the class are copied to the member variables of the newly created class object. The initialization or copying of the values to the member variables is done as per the definition of the copy constructor.

```
#include <iostream>

using namespace std;

class Example {

public:

int a;

Example(int x){              // parameterized constructor

a=x;

}

Example(Example &ob){        // copy constructor

a = ob.a;

}

};

int main(){

Example e1(36);      // Calling the parameterized constructor

Example e2(e1);     // Calling the user-defined copy constructor

cout<<e2.a;

return 0;
```

```
}
```

Inheritance

Inheritance is an important pillar of OOP (Object-Oriented Programming). The capability of a class to derive properties and characteristics from another class is called Inheritance. So, when we create a class, we do not need to write all the properties and functions again and again, as these can be inherited from another class that possesses it. Inheritance allows the user to reuse the code whenever possible and reduce its redundancy.

Why Inheritance?

Suppose we need to create a racing game with cars and motorcycles as vehicles.

To solve this problem, we can create two separate classes to handle each of their functionalities.

However, both cars and motorcycles are vehicles and they will share some common variables/arrays and functions.

So instead of creating two independent classes, we can create the Vehicle class that shares the common features of both cars and motorcycles. Then, we can derive the Car class from this Vehicle class.

In doing so, the `Car` class inherits all the variables and functions of the `Vehicle` class. And we can add car-specific features in the `Car` class.

Similarly, we can derive the `Motorcycle` class that inherits from the `Vehicle` class. Again, this `Motorcycle` class gets all vehicle-specific variables and functions from the `Vehicle` class, along with the unique features of motorcycles.

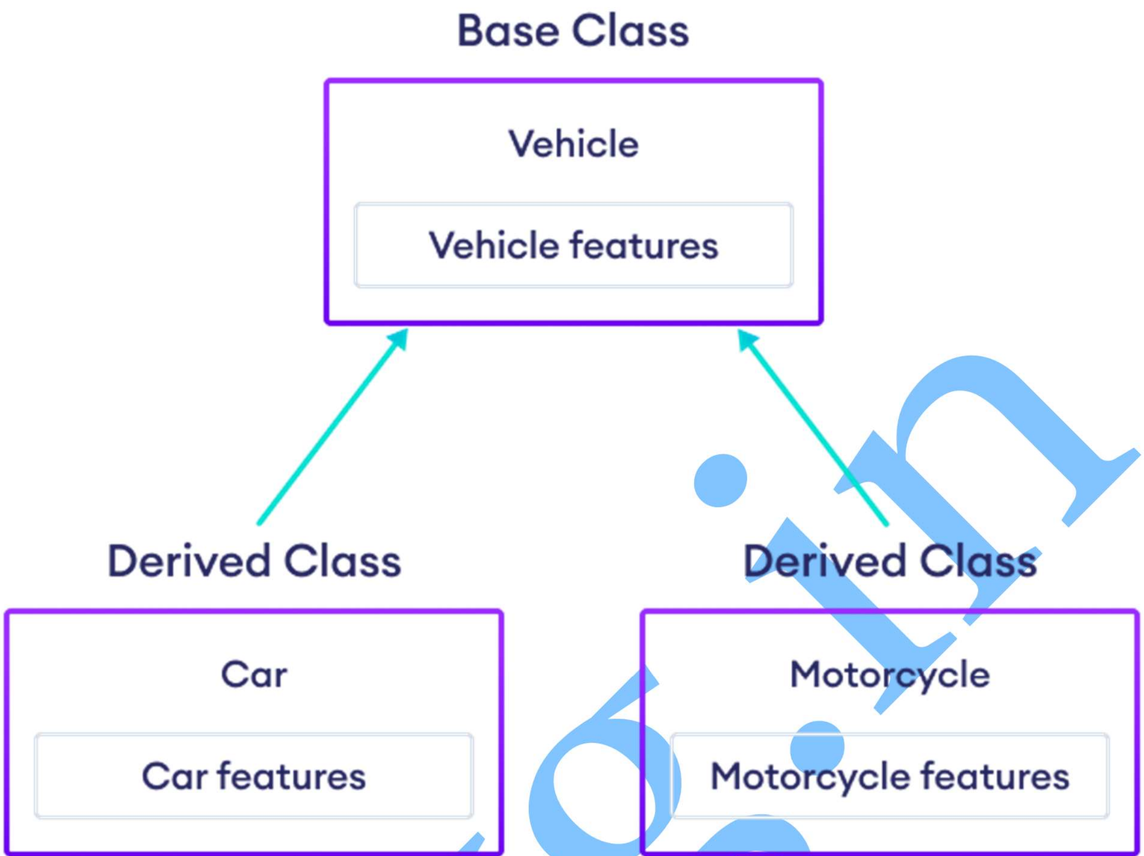


Figure: C++ Inheritance

This is the basic concept of inheritance. Inheritance allows a class (child or derived class) to inherit variables and functions from another class (parent or base class).

In our example, `Vehicle` is the superclass (also known as parent or base class) and `Car` and `Motorcycle` are subclasses (also known as child or derived classes).

Example: C++ Inheritance

Let's create an object of the `Dog` class and access the functions of `Animal`.

```
#include <iostream>
using namespace std;
// base class
class Animal {
public:
void eat() {
cout << "I can eat" << endl;
}
};
// the Dog class is derived from Animal
class Dog: public Animal {
public:
void bark() {
cout << "I can bark" << endl;
}
};
int main() {
// create object of Dog
Dog dog1;
// access the bark function of Dog
dog1.bark();
// access the eat() function of Animal
dog1.eat();
return 0;
}
```

Output

```
I can bark
I can eat
```

Here, `dog1` is an object of the `Dog` class. Hence,

- `dog1.bark()` calls the `bark()` function of the `Dog` class.
- `dog1.eat()` calls the `eat()` function of the `Animal` class. This can be done because `Dog` is derived from `Animal`, so the `Dog` class inherits all the variables and functions of `Animal`.

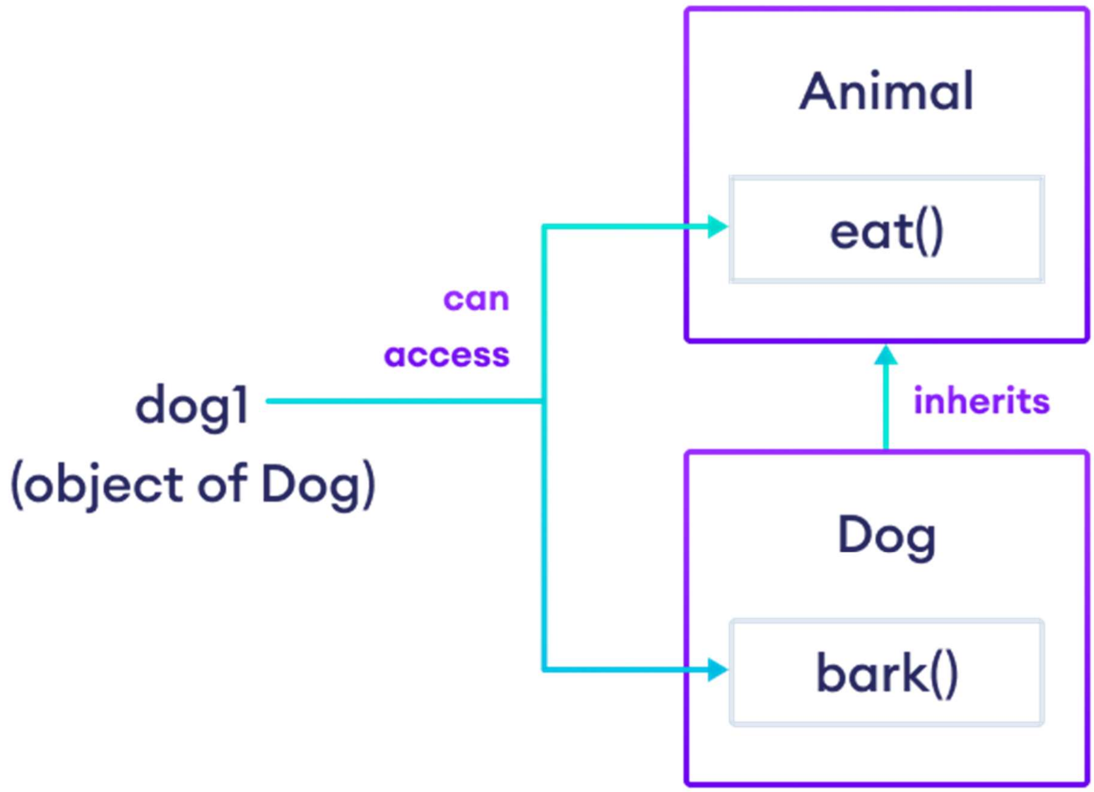


Figure: C++ Inheritance

Note: Objects of `Animal` can only access variables and functions of `Animal`. It's because `Dog` is derived from `Animal` and not the other way around.

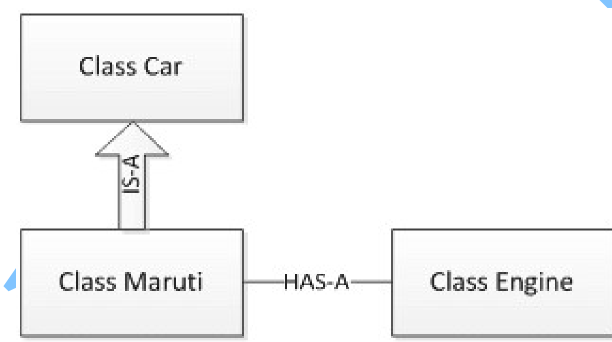
IS-A Relationship:

In object-oriented programming, the concept of IS-A is a totally based on Inheritance, it is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House.

HAS-A Relationship:

In object-oriented programming, the concept of HAS-A relationship is a totally based on the Inheritance, it is just like saying "A has a B type of thing". For example, House **HAS-A** Bathroom, Office **HAS-A** Bathroom, Ferrari **HAS-A** Engine, Lamborghini **HAS-A** Engine. Inheritance is uni-directional. For example, Ferrari **HAS-A** Engine. But Engine has not a Ferrari.

Let's understand these concepts with an example of Car class.



Destructor

What is a destructor?

Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

- Destructor is also a special member function like constructor. Destructor destroys the class objects created by constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object create by constructor. Hence destructor can-not be overloaded.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when object goes out of scope.
- Destructor release memory space occupied by the objects created by constructor.
- In destructor, objects are destroyed in the reverse of an object creation.

Example

```
// Example:

#include<iostream>
using namespace std;

class Test
{
   public:
          Test()
         {
                 cout<<"\n Constructor executed";
         }

         ~Test()
            {
                 cout<<"\n Destructor executed";
            }
};
main()
{
     Test t;

    return 0;
```

Diff between delete and free ()

delete and **free ()** have similar functionalities in programming languages but they are different. In C++, the delete operator should only be used either for the pointers pointing to the memory allocated using new operator or for a NULL pointer, and free () should only be used either for the pointers pointing to the memory allocated using malloc () or for a NULL pointer.

Delete	free ()
It is an operator.	It is a library function.
It de-allocates the memory dynamically.	It destroys the memory at the runtime.
It should only be used either for the pointers pointing to the memory allocated using the **new** operator or for a NULL pointer.	It should only be used either for the pointers pointing to the memory allocated using **malloc ()** or for a NULL pointer.
This operator calls the destructor after it destroys the allocated memory.	This function only frees the memory from the heap. It does not call the destructor.
It is faster.	It is comparatively slower than delete as it is a function.

Diff between new and malloc()

malloc () vs new:

malloc () is a C library function that can also be used in C++, while the **“new”** operator is specific for C++ only.
Both **malloc ()** and **new** are used to allocate the memory dynamically in heap. But **“new”** does call the constructor of a class whereas **“malloc ()”** does not.

new	malloc ()
calls constructor	does not call constructor
It is an operator	It is a function
Returns exact data type	Returns void *
on failure, Throws bad_alloc exception	On failure, returns NULL
size is calculated by compiler	size is calculated manually

Encapsulation

Encapsulation is another key feature of object-oriented programming. It means bundling variables and functions together inside a class.

Let's understand this with the help of an example.

Suppose we need to compute the area of a rectangle. We know that to compute the area, we need two data (variables) - `length` and `breadth` - and a function - `calculate_area()`.

Hence, we can bundle these variables and the function together inside a single class.

```
class Rectangle {
    public:

        // variables to store data
        int length;
        int breadth;

        // function to calculate area
        int calculate_area() {
            int area = length * breadth;
            return area;
        }
};
```

This is an example of encapsulation.

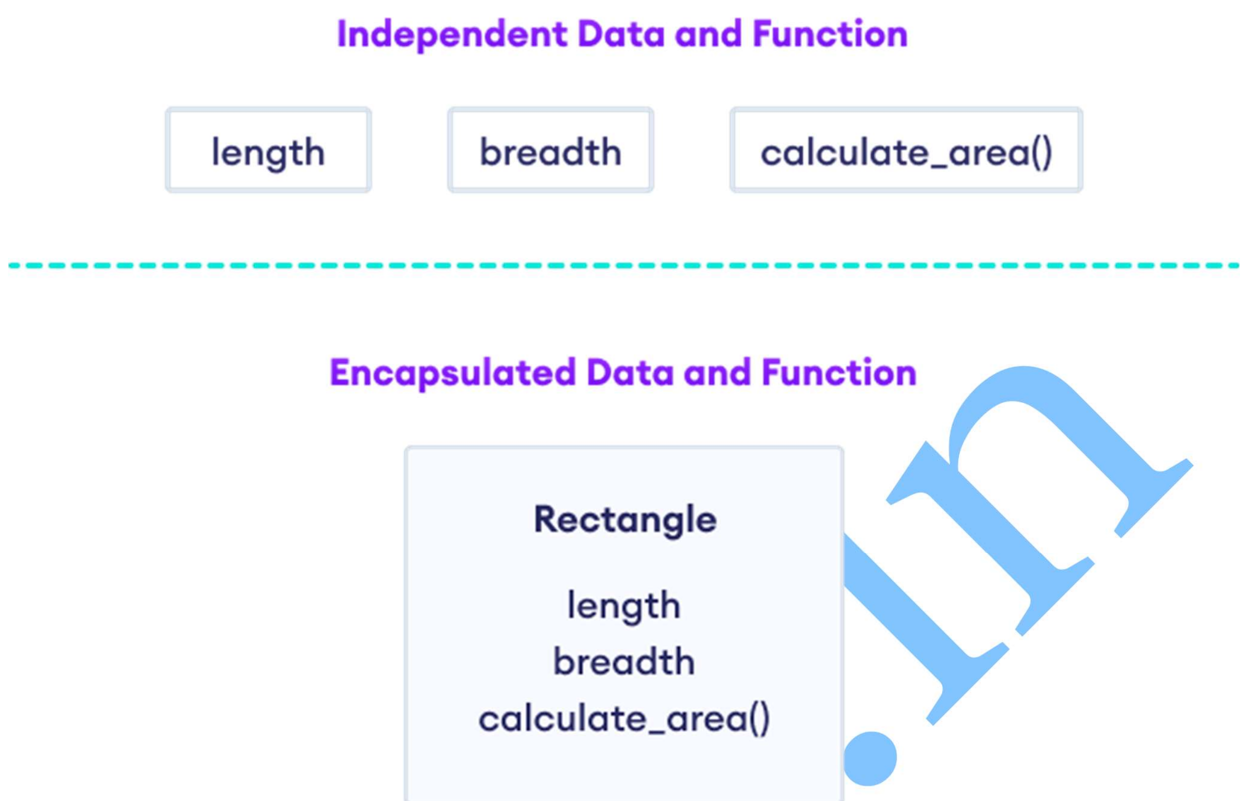


Figure: C++ Encapsulation

With this, we can now keep related variables and functions together, making our code clean and easy to understand.

Why Data Hiding?

Not all data inside a class are meant to be universally accessible. It is very important to hide some of the data from other functions and classes in our program.

For instance, consider a class called Bank_Account that allows the program to store the bank details of different people. Naturally, many of the details are confidential and should only be accessible to a select few.

But if our program gives public access to these crucial details, then anyone using our program can tamper with sensitive information.

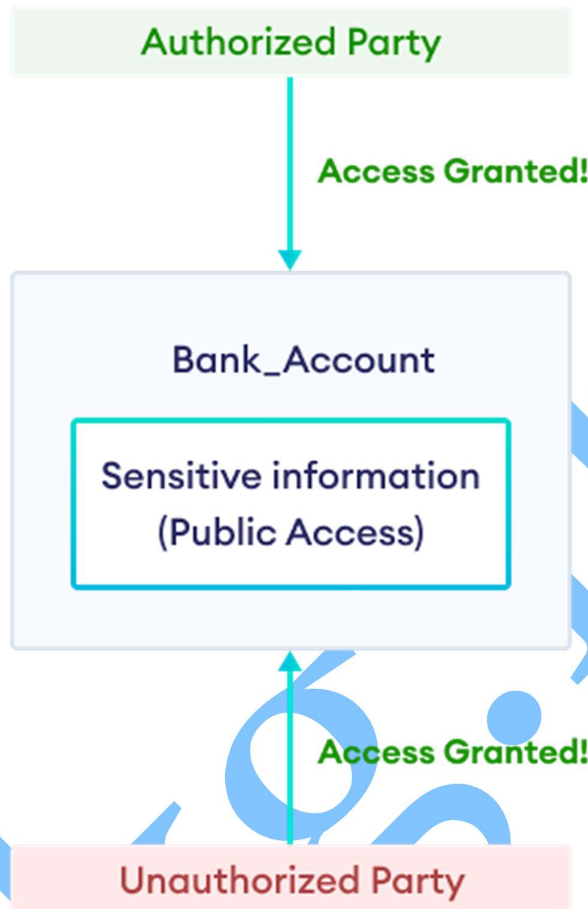


Figure: Public data can be accessed by unauthorized parties

To prevent this, object-oriented programming languages such as C++ have integrated a very crucial feature into their system: data hiding.

Data hiding refers to restricting access to data members of a class. As we have discussed earlier, this is to prevent other functions and classes from tampering with the class data.

That's why it is important to declare sensitive variables private so that unauthorized users don't get access to these variables.

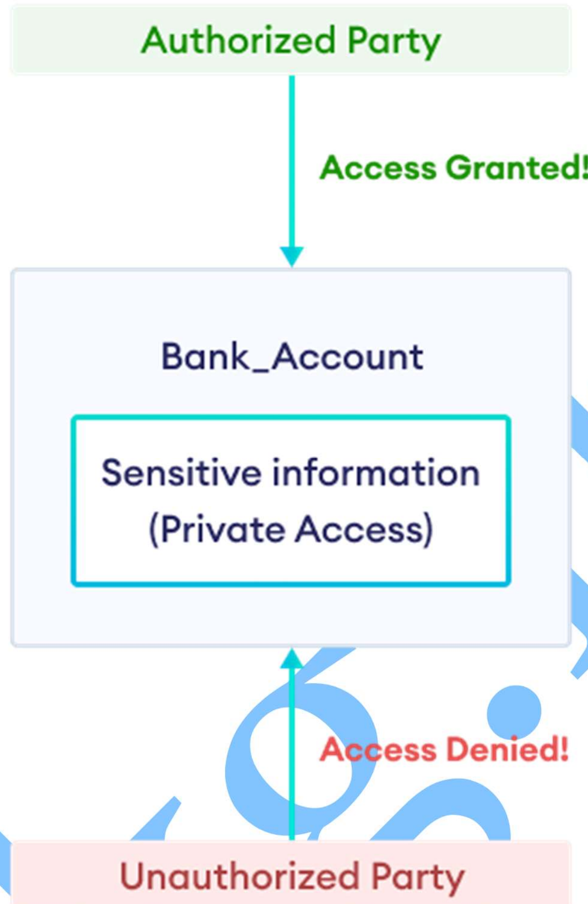


Figure: Private data cannot be accessed by unauthorized parties

Abstract Class

A class that contains a pure virtual function is known as an abstract class.

We cannot create objects of an abstract class. However, we can derive classes from them, and use their data members and member functions.

Normally, when we create a class, we can create objects from the class. For example,

```
class Animal {
// class body
};
// object of Animal
Animal obj;
```

Here, we are creating an object named `obj` of the `Animal` class.

In C++, we can also create abstract classes which contain pure virtual functions. For example,

```
// abstract class
class Polygon {
public:
// pure virtual function
virtual void get_area() = 0;
};
```

Here, `Polygon` is an abstract class because it includes the pure virtual function `get_area()`.

Unlike regular classes, we cannot create objects of an abstract class.

Example: Abstract Class

```
#include <iostream>
using namespace std;
// abstract class
class Polygon {
public:
// regular function
void print_sides() {
cout << "Print sides of Polygon." << endl;
}
// pure virtual function
virtual void get_area() = 0;
```

```
};
class Rectangle: public Polygon {
public:
// implementation of the pure virtual function
void get_area() {
cout << "Print the area of Rectangle." << endl;
}
};
int main() {
// create object of the child class
Rectangle rectangle1;
// access the regular function of Polygon
rectangle1.print_sides();
// access the implemented pure virtual function
rectangle1.get_area();
return 0;
}
```

Output

```
Print sides of Polygon.
Print the area of Rectangle.
```

In the above example, we have created the `Rectangle` class by inheriting the abstract class `Polygon`.

The `Rectangle` class now inherits both the regular and pure virtual functions, so we must provide the implementation for the pure virtual function `get_area()`.

We then used an object of `Rectangle` to access functions of the abstract class.

Why Abstract Classes?

Suppose there is a function that is common among multiple entities. For example, all polygons have an area, and the function for calculating area can be shared among different types of polygons (rectangle, triangle, etc.).

However, the process of calculating the area of each polygon is different from one another. So, we cannot provide one implementation of calculating area that will work for all the polygons.

Instead, we can create a function without any implementation and all the polygons will provide their own implementation for the function.

For this, we use abstract classes with pure virtual functions and all the polygons implementing the class will provide their own version of the pure virtual function.

Polymorphism

Polymorphism is another important concept in object-oriented programming. It simply means more than one form: the same entity (function or operator) can perform different operations in different scenarios.

For example, the `+` operator can be used to perform numeric addition as well as string concatenation.

```
#include <iostream>
using namespace std;
int main () {
// use + to add two numbers
int result = 4 + 8;
cout << "Sum: " << result << endl;
string str1 = "Hello ";
string str2 = "World";
// use + to join two strings
string new_string = str1 + str2;
cout << new_string << endl;
return 0;
}
```

Output

```
Sum: 12
Hello World
```

In the above example, we have used the same `+` operator to perform two different tasks:

- `4 + 8` - adds two numbers
- `str1 + str2` - joins two strings

Here, the `+` operator has two different forms. Thus, it is an example of C++ Polymorphism.

Function Overriding

In function overriding, the same function is present in both the base class and the derived class.

```
// base class
class Animal {
public:
// make_sound() in the base class
void make_sound() {
cout << "Making animal sound" << endl;
}
};
// derived class
class Dog: public Animal {
public:
// make_sound() in the base class
void make_sound() {
cout << "Woof Woof" << endl;
}
};
```

In this case, we can independently access functions of the base class and derived class by using their respective objects. For example,

```
#include <iostream>
using namespace std;
class Animal {
public:
// make_sound() function of base class
void make_sound() {
cout << "Making animal sound" << endl;
}
};
class Dog: public Animal {
public:
// make_sound() function of derived class
void make_sound() {
cout << "Woof Woof" << endl;
}
};
int main() {
// access function of derived class
Dog dog1;
dog1.make_sound();
// access function of base class
Animal animal1;
animal1.make_sound();
return 0;
```

}

Output

```
Woof Woof
Making animal sound
```

we are able to use the same function `make_sound()` to perform two different tasks.

Hence, we can say function overriding helps us achieve polymorphism in C++.

💡

Note: Because Polymorphism includes function overriding, the related concepts of virtual functions and pure virtual functions are also examples of Polymorphism.

Function overloading

In C++, two or more functions can have the same name if they have different numbers/types of parameters. Let's see an example.

```
// function with no parameter
void display() {
    ...
}
```

```
// function with an integer parameter
void display(int number) {
    ...
}
```

```
// function with string parameter
void display(string name) {
    ...
}
```

```
// function with two parameters
void display(string name, int age) {
    ...
}
```

Here, we have created 4 functions with the same name `display()`, but different parameters. These functions are called overloaded functions and the process is called function overloading.

From the above explanation, it's clear that there are two ways to perform function overloading.

- With different numbers of parameters
- With different types of parameters

Let's see an example of both.

Overloading With Different Number of Parameters

```
#include <iostream>
using namespace std;
class Addition {
public:
// function with 2 parameters
void add_numbers (int num1, int num2) {
int sum = num1 + num2;
cout << "Sum of 2 digits: " << sum << endl;
}
// function with 3 parameters
void add_numbers(int num1, int num2, int num3) {
int sum = num1 + num2 + num3;
cout << "Sum of 3 digits: " << sum << endl;
}
};
int main() {
// create an object of Addition
Addition addition;
// call function with 2 arguments
addition.add_numbers(3, 5);
// call function with 3 arguments
addition.add_numbers(7, 9, 4);
return 0;
}
```

Output

```
Sum of 2 digits: 8
Sum of 3 digits: 20
```

In the above example, we have overloaded the `add_numbers()` function with 2 and 3 parameters.

Here, based on the number of arguments passed during the function call, the corresponding function is executed.

You can see we are able to use the same function `add_numbers()` for two different tasks. Hence, this helps in achieving Polymorphism.

```
class Addition {
  public:

    void add_numbers(int num1, int num2) {
      // code
    }

    void add_numbers (int num1, int num2, int num3) {
      // code
    }
};

int main() {

  Addition addition;

  addition.add_numbers(3, 5);

  addition.add_numbers(7, 9, 4);

  return 0;
}
```

Overloading With Different Types of Parameters

Now, let's try function overloading with different parameter types. For example,

```
#include <iostream>

using namespace std;

class Addition {

public:

// function with integer parameters
```

```
int add_numbers (int number1, int number2) {
int sum = number1 + number2;
return sum;;
}
// function with double parameters
double add_numbers(double number1, double number2) {
double sum = number1 + number2;
return sum;
}
};
int main() {
// create an object of Addition
Addition addition;
// call function with integer arguments
int sum1 = addition.add_numbers(12, 9);
cout << "Sum of integers: " << sum1 << endl;
// call function with double arguments
double sum2 = addition.add_numbers(32.9, 43.7);
cout << "Sum of doubles: " << sum2 << endl;
return 0;
}
```

Output

```
Sum of integers: 21
Sum of doubles: 76.6
```

Here, we have overloaded the `add_numbers()` function with `int` and `double` parameters. Now, depending on the types of arguments passed during the function call, the corresponding function is executed.

As you can see, this example also uses the same function for two different purposes. Hence, this example is also an implementation of polymorphism.

Important! Function overloading is only associated with parameters, not their return types. Overloaded functions may have the same or different return types, as long as their parameters are different.

Virtual Function

A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call.
- They are mainly used to achieve Runtime polymorphism.
- Functions are declared with a virtual keyword in a base class.
- The resolving of a function call is done at runtime.

Rules for Virtual Functions

The rules for the virtual functions in C++ are as follows:

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as the derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.

6. A class may have a virtual destructor but it cannot have a virtual constructor.
7. Let's see an example.

```
#include <iostream>
using namespace std;
class Person {
public:
virtual void display_info() {
cout << "I am a person." << endl;
}
};
class Student : public Person {
public:
void display_info() {
cout << "I am a student." << endl;
}
};
int main() {
Student student1;
// create Person pointer that points to student object
Person* ptr = &student1;
ptr->display_info();
return 0;
}
// Output: I am a student.
```

C++ Pure Virtual Functions

Pure virtual functions are used

- if a function doesn't have any use in the base class
- but the function must be implemented by all its derived classes

Let's take an example,

Suppose, we have derived `Triangle`, `Square` and `Circle` classes from the `Shape` class, and we want to calculate the area of all these shapes.

In this case, we can create a pure virtual function named `calculateArea()` in the `Shape`.

Since it's a pure virtual function, all derived classes `Triangle`, `Square` and `Circle` must include the `calculateArea()` function with implementation.

A pure virtual function doesn't have the function body and it must end with `= 0`.

For example,

```
class Shape {
    public:

      // creating a pure virtual function
      virtual void calculateArea() = 0;
};
```

Note: The `= 0` syntax doesn't mean we are assigning 0 to the function. It's just the way we define pure virtual functions.

Abstract Class

Example: C++ Abstract Class and Pure Virtual Function

```
// C++ program to calculate the area of a square and a circle

#include <iostream>
using namespace std;

// Abstract class
class Shape {
   protected:
    float dimension;

   public:
    void getDimension() {
        cin >> dimension;
    }

    // pure virtual Function
    virtual float calculateArea() = 0;
};

// Derived class
class Square : public Shape {
   public:
    float calculateArea() {
        return dimension * dimension;
    }
};
```

```
// Derived class
class Circle : public Shape {
   public:
    float calculateArea() {
        return 3.14 * dimension * dimension;
    }
};

int main() {
    Square square;
    Circle circle;

    cout << "Enter the length of the square: ";
    square.getDimension();
    cout << "Area of square: " << square.calculateArea() << endl;

    cout << "\nEnter radius of the circle: ";
    circle.getDimension();
    cout << "Area of circle: " << circle.calculateArea() << endl;

    return 0;
}
```

Output

```
Enter the length of the square: 4
Area of square: 16

Enter radius of the circle: 5
Area of circle: 78.5
```

In this program, `virtual float calculateArea() = 0;` inside the `Shape` class is a pure virtual function.

That's why we must provide the implementation of `calculateArea()` in both of our derived classes, or else we will get an error.

Differences

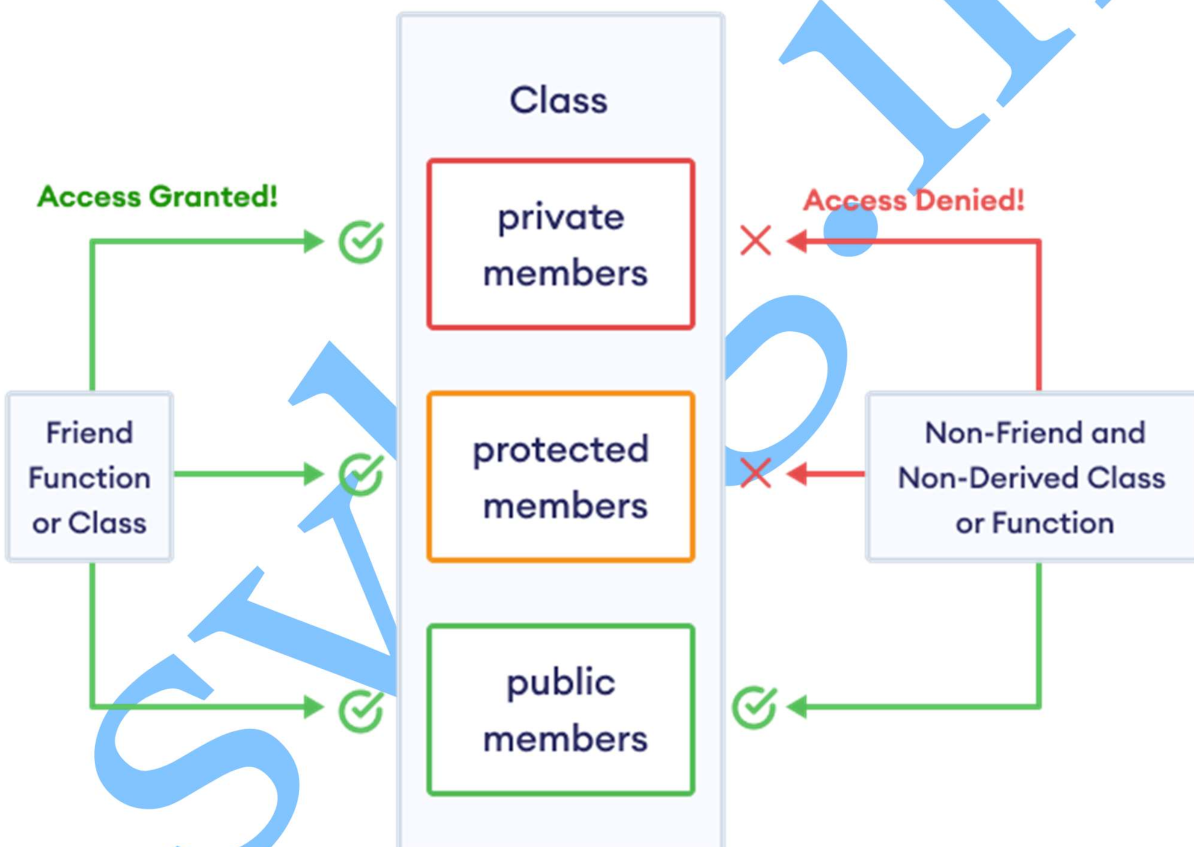
Virtual Function	Pure Virtual Function
In the virtual function, the derived class overrides the function of the base class; it is the case of function overriding.	In a pure virtual function, the derived class would not call the base class function as it has not defined instead it calls the derived function which implements that same pure virtual function in the derived call.
Class containing virtual function may or may not be an Abstract class.	If there is any pure virtual function in a class, then it becomes an "Abstract class".
Virtual function in the base does not enforce to derived for defining or redefining	In pure virtual function, the derived class must redefine the pure virtual sunction of the base class. Otherwise, that derived class will become abstract as well.

Friend Functions and classes

Private and protected class members cannot be accessed from outside of the class. The only way we have accessed private members so far is through getter and setter functions (and sometimes with constructors).

However, there is another way to access private members, known as friend functions and friend classes.

Friend functions and classes are exceptional cases using which we can access all class members from outside of the class, including private and protected members.



C++ Friend Function

As mentioned before, a friend function can access the private and protected members of a class. We use the `friend` keyword to declare a friend function. For example,

```
class Rectangle {
...
// friend function declaration
friend int find_area(Rectangle);
...
};
```

In the above code, we have declared a friend function `find_area()` inside the `Rectangle` class so that it can access all of the class members.

Let's explore further with an example.

```
#include <iostream>
using namespace std;
class Rectangle {
private:
int length, breadth;
public:
// constructor to initialize variables
Rectangle() : length(8), breadth(6) {}
// friend function declaration
friend int find_area(Rectangle);
};
// friend function definition
int find_area(Rectangle obj) {
// access private members
// from the friend function
int area = obj.length * obj.breadth;
return area;
}
int main() {
```

```
Rectangle obj;
// call find_area() by
// passing the object of Rectangle
cout << "Area = " << find_area(obj) << endl;
return 0;
}
// Output: Area = 48
```

In the above example, we have created the `Rectangle` class. It consists of two private members: `length` and `breadth`.

Notice that we have declared a friend function inside the `Rectangle` class and its definition is outside the class.

```
class Rectangle {
...
// friend function declaration
friend int find_area(Rectangle);
};
// friend function definition
int find_area(Rectangle obj) {
...
}
```

The function accepts an object of the `Rectangle` class as its parameter.

As you can see, we are able to access the private variables: `length` and `breadth` from the outer function (`find_area()`). It's possible because the outer function `find_area()` is declared as a friend function.

C++ Friend Class

Similar to friend functions, we can also create friend classes. A friend class can access the member variables and member functions of the class it is declared in. For example,

```
#include <iostream>

using namespace std;

class Animal {

private:

int legs_count;

public:

// constructor to initialize variable

Animal() : legs_count(4) {}

// declare friend class

friend class Dog;

};

// define friend class

class Dog {

public:

void count_legs() {

// create Animal object

Animal animal;

// access private variable of Animal class

cout << "Legs = " << animal.legs_count << endl;

}

};

int main() {

// create object of friend class

Dog dog;

dog.count_legs();

return 0;

}

// Output: Legs = 4
```

Here, the class `Dog` is a friend class of class `Animal`.

```
// inside Animal class
// declare friend class
friend class Dog;
```

That's why we are able to access the private variable `legs_count` from the `Dog` class.

```
// inside Dog class
void leg_count() {
Animal animal;
cout << "Legs = " << animal.legs_count << endl;
}
```

Classes and Object

we need to create a class first before we can create objects from it.

In C++, we use the `class` keyword to create a class. For example,

```
class Car {
  ...
};
```

Here, we have created a class named `Car`.

A class can contain:

- data members - variables/arrays to store data
- member functions - to perform tasks on data members

Note: A class ends with the code `};` we have ended loops and functions with the `}` symbol. For classes, however, we need to add a semicolon; after the closing brace '}' .

We will gradually add different functions and variables inside a class. But first, let's create objects from the class.

Creating Objects

Here's how we can create objects of a class.

```
// create a class
class Car {
    ...
};

// create object of the Car class
Car car1;
Car car2;
```

Here, `car1` and `car2` are objects of the `Car` class.

Next, we will learn how variables and functions are used with a class.

Access Modifiers

So far in our example, we have been using the public keyword along with our member variables and functions within the class.

```
class Car {
    public:
        // code
};
```

Here, public means these data members and functions can be accessed from anywhere in the program. Hence, we were able to access them from the main () function.

However, there might be situations where we wouldn't want our data members and functions to be accessed from outside. For this, we use access modifiers in C++.

Access modifiers are used to set the visibility of data members, functions, and even classes. For example, if we don't want our class members to be accessed from outside, we can mark them as private using the private access modifier.

```
class Car {
    private:
        // code
```

};

There are three types of access modifiers in C++.

- public - allows access from outside
- private - prevents access from outside
- Protected - prevents access from outside

Public Modifier

As the name suggests, variables and functions declared with the `public` access modifier can be accessed from any class. Let's see an example,

```
#include <iostream>
using namespace std;
class Student {
// public variable
public:
string name;
};
int main() {
// create object of Student
Student student1;
// access the public variable of the Student class
student1.name = "Rosie";
cout << "Student Name: " << student1.name << endl;
return 0;
}
// Output: Student Name: Rosie
```

In the above example, we have used the `public` access modifier with the `name` variable. That's why we are able to assign a new value and access its value from the `main()` function.

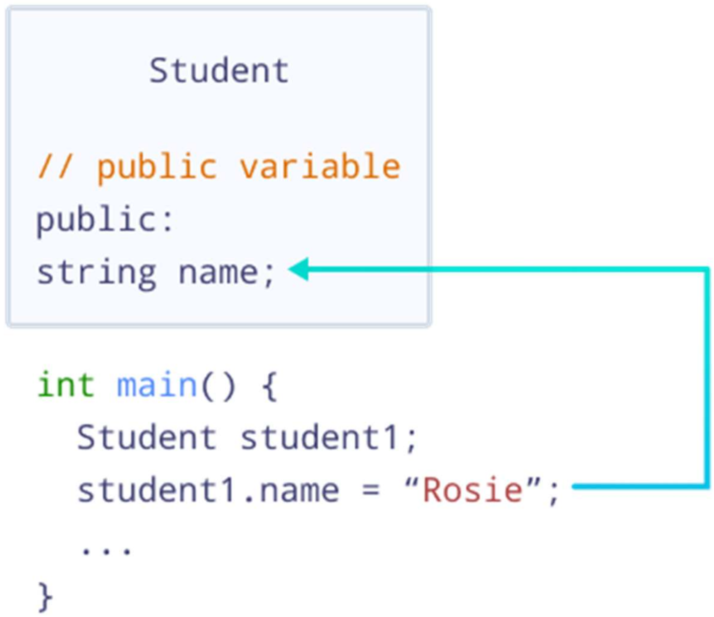


Figure: public Access Modifier

Private Modifier

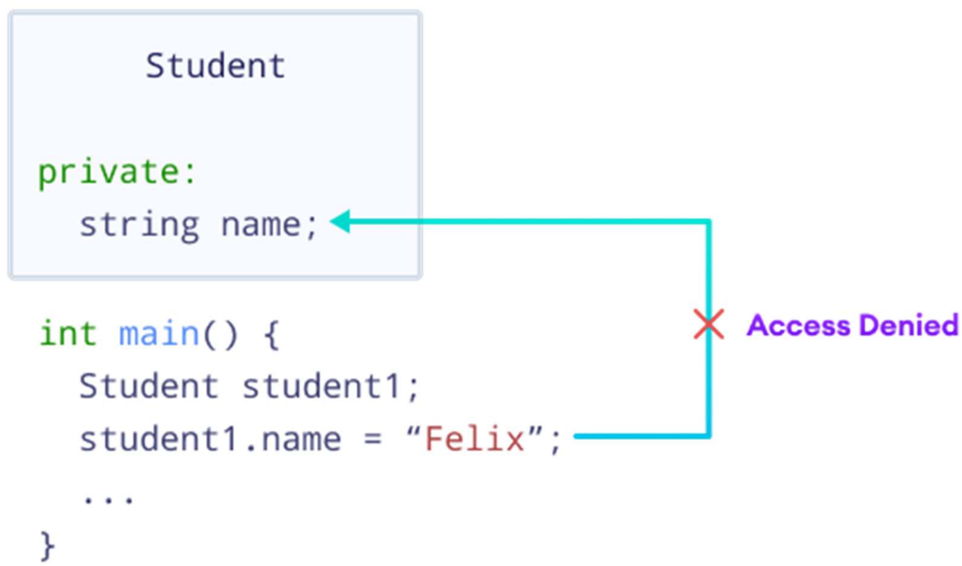
As mentioned earlier, if we create a variable with a `private` access modifier, it can't be accessed from outside. Let's see an example.

```
#include <iostream>

using namespace std;

class Student {

// create private variable

private:

string name;

};

int main() {

// create an object of Student

Student student1;

// try to access the private data member

student1.name = "Felix";

cout << "Name: " << student1.name << endl;

return 0;

}
```

When we run this code, we will get an error:

```
17:14: std::string Student::name' is private within this context
   17 |      student1.name = "Felix"
```

Here, you can see that we get an error when we try to access the `private` variable `name` from the `main()` function.



Getter and Setter Functions

We know that a private data member cannot be accessed from outside of a class. However, if we need to access them, we can use getter and setter functions.

- Setter Function - allows us to set the value of data members
- Getter Function - allows us to get the value of data members

Let's see an example.

```
#include <iostream>

using namespace std;

class Student {

private:

string name;

};

int main() {
```

```
// create an object of Student
Student student1;
// access the private name
student1.name = "Felix";
cout << "Name: " << student1.name << endl;
return 0;
}
```

We know this code will cause an error because we are trying to directly access the private variable from the `main()` function.

Now let's use the getter and setter functions to access the `name` variable.

```
#include <iostream>
using namespace std;
class Student {
private:
string name;
public:
// setter function
void set_name(string student_name) {
name = student_name;
}
// getter function
string get_name() {
return name;
}
};
int main() {
// create an object of Student
Student student1;
// assign value to name using setter function
student1.set_name("Felix");
// access value of name using getter function
cout << "Name: " << student1.get_name() << endl;
```

```
    return 0;
}
```

Output

```
Name: Felix
```

As you can see, we have successfully assigned a new value and accessed it using the getter and setter functions.

Student

```
// private variable
private:
  string name;

public:
```

```
// setter function
void set_name(string student_name) {
  name = student_name;
}
```

```
// getter function
string get_name() {
  return name;
}
```

Main

```
// create object of Student
Student student1;
```

```
// assign value using the setter
student1.set_name("Felix");
```

```
// access the value using getter
cout << "Name: " << student1.get_name();
```

C++ Protected Members

Similar to public and private, we use the protected keyword to declare protected class members in C++. For example,

class Person {

protected:

int id;

public:

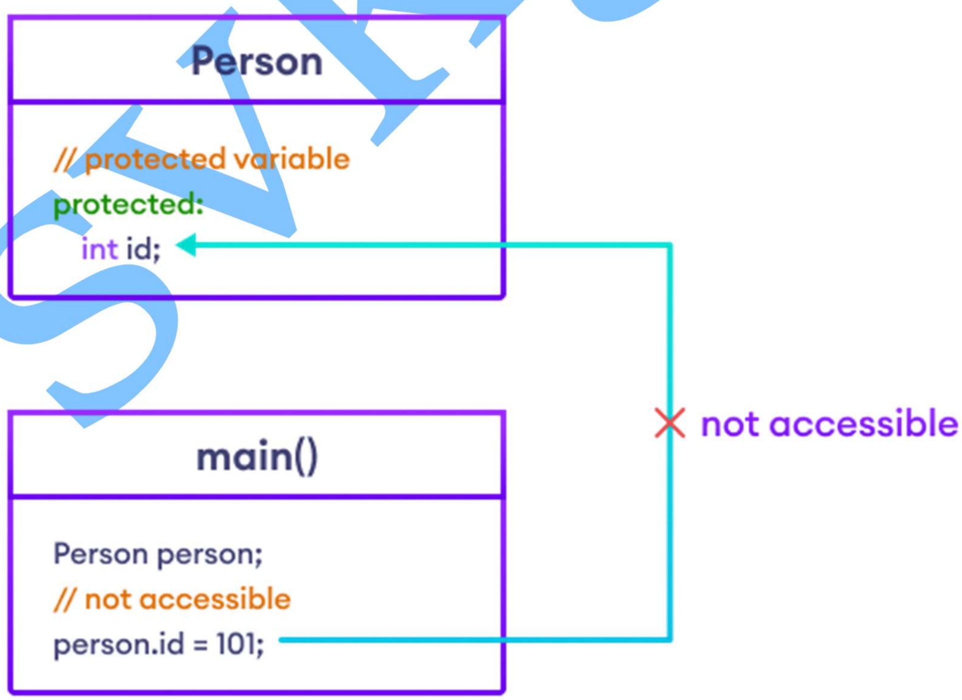
string name;

};

Here,

- id is protected
- name is public

Once we declare a variable/function protected, it can be only accessed from that class and its derived classes. If we try to access it from somewhere else, we will get an error.



Now let's see how we can access protected class members.

```
#include <iostream>
using namespace std;
class Person {
protected:
int id = 101;
public:
string name;
};
class Student: public Person {
public:
void access_protected() {
// access protected variable
cout << "ID: " << id << endl;
}
};
int main() {
// create an object of Student
Student student;
// access the public variable of the parent class
student.name = "Jon Snow";
cout << "Name: " << student.name << endl;
// call the access_protected() function
student.access_protected();
return 0;
```

}

Output

Name: Jon Snow

ID: 101

In the above example, we are accessing the protected variable inside the subclass Student.

```
void access_protected() {
cout << "ID: " << id << endl;
}
```

This is possible because protected variables can only be accessed by the same class or its subclasses.

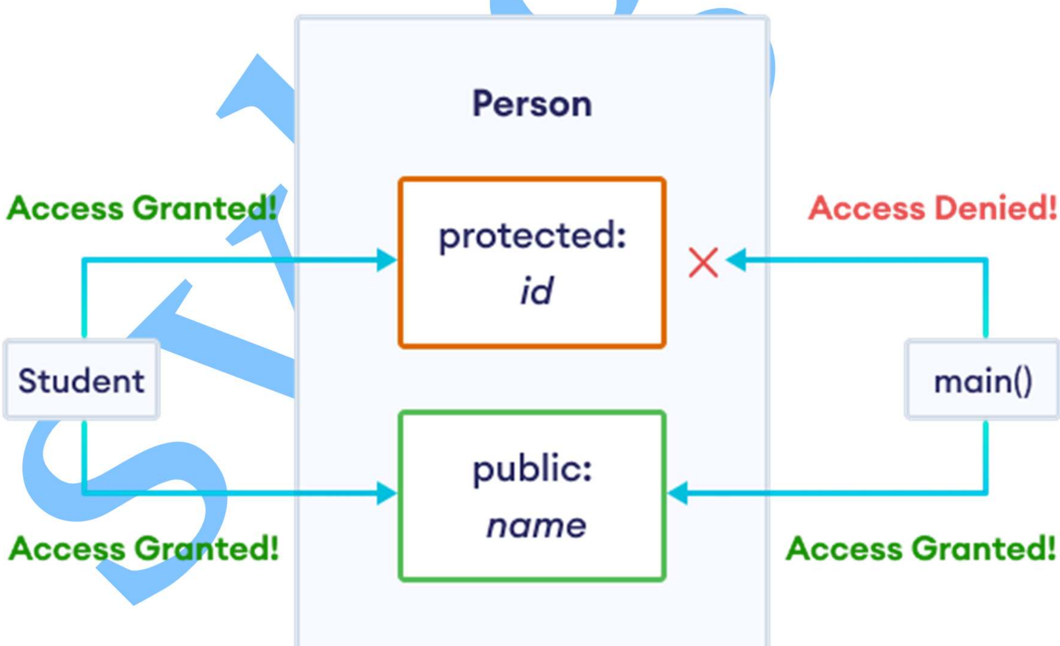


Figure: Protected Access Modifier

In order to access protected members outside the class and its subclasses, we must use public getter and setter functions (either inside the base class or inside the derived class).

Accessibility	Private Members	Protected Members	Public Members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes

We have already discussed how to access the private and protected variables from outside:

- private variables - use public getter and setter functions
- protected variables - access inside the subclass or use public getter and setter functions

Inheritance Access Control in C++

In C++, we can derive classes in 3 modes:

- Public Inheritance
- Protected Inheritance
- Private Inheritance

Properties of the Different Inheritance Modes

The following code specifies how members of the base class are inherited in the derived classes:

```
class Parent {
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};

// public inheritance
class Public_Child: public Parent {
    // x is public
    // y is protected
    // z is not accessible from Public_Child
};

// protected inheritance
class Protected_Child: protected Parent {
    // x is protected
    // y is protected
```

```
    // z is not accessible from Protected_Child
};
```

```
// private inheritance
class Private_Child: private Base {
    // x is private
    // y is private
    // z is not accessible from Private_Child
};
```

Access Members of the Base Class (Public Inheritance)

- private members - create public getter and setter functions in the base class to access
- protected members - create public getter and setter functions in either the base class or the derived class
- public members - can be accessed from outside the class

Access Members of the Base Class (Protected and Private Inheritance)

- protected and public members - create public getter and setter functions in the derived class
- private members - can't be accessed directly from the derived class

this pointer

Introduction to 'this' Pointer

In C++, we use this keyword to refer to the current object.

Let's see what that means.

```
#include <iostream>
using namespace std;
// define the student class
class Student {
public:
// public string variable to hold the student's name
string name;
// function that displays the student's name
void display_name() {
cout << "Student's name using this: " << this->name << endl;
```

```
}
};

Int main() {
// create a Student object and set the name variable
Student student;
student.name = "John Doe";
// call the display_name() function
student.display_name();
// print the student's name
cout << "Student's name using object: " << student.name << endl;
return 0;
}
```

Output

```
Student's name using this: John Doe
Student's name using object: John Doe
```

In the above example, you can see both `student.name` and `this->name` give the same result, `John Doe`.

Basically, what happens here is when we call the `display_name()` function using the `student` object, `this` will refer to the current object, which is `student`.

```
void display_name() {
cout << "Student's name using this: " << this->name << endl;
}
```

Hence, we get the output `John Doe` (value of `name` for `student`).

Similarly, if we call the function with another object (let's say `student2`), `this->name` will print the value of `name` for `student2`. For example,

```
#include <iostream>
using namespace std;
// define the student class
class Student {
```

```
public:
// public string variable to hold the student's name
string name;
// function that displays the student's name

void display_name() {
cout << "Student's name using this: " << this->name << endl;
}
};
int main() {
// create a Student object and set the name variable
Student student;
student.name = "John Doe";
// call the display_name() function
student.display_name();
// create a Student object and set the name variable
Student student2;
student2.name = "Lily Doe";
// call the display_name() function
student2.display_name();
return 0;
}
```


Output

```
Student's name using this: John Doe
Student's name using this: Lily Doe
```

Here, for the function call

- `student.display_name()` - `this` refers to the `student` object
- `student2.display_name()` - `this` refers to the `student2` object

Static

static Keyword

So far, we have been using an object of the class to access variables and functions of a class. For example,

```
#include <iostream>

using namespace std;

class Animal {

public:

void display() {

cout << "I am an animal." << endl;

}

};

int main() {

// object of the Animal class

Animal obj;

// access the function using the object

obj.display();

return 0;

}
```

Output

```
I am an animal.
```

Here, we have used the object `obj` of the `Animal` class to access the member function `display ()`.

However, there might be situations where we want to access variables and functions without creating the object. For this, we can use the `static` keyword.

Example: static Keyword

```
#include <iostream>
using namespace std;
class Animal {
public:
// static function
static void display() {
cout << "I am an animal." << endl;
}
};
int main() {
// access the function using class
Animal::display();
return 0;
}
// Output: I am an animal.
```

Here, you can see that we are able to directly access the `display()` function using the class name with the **scope resolution operator** `::`.

```
Animal::display();
```

Notice that we haven't created an object for this purpose. This is possible because we have declared the function as `static`.

static Member Variables

Unlike static functions, static member variables are declared inside the class and defined outside the class. For example,

```
class Student {
public:
// static variable declaration
static int subject_code;
};
// static variable definition
int Student::subject_code = 13;
```

In the above example, we have created the static variable `subject_code`.

Here, you can see we have declared the static variable inside the class; however, we have provided its definition outside the class.

Access static Variables

Like with static functions, we can use the class name with the scope resolution operator `::` to access static variables. For example,

```
#include <iostream>
using namespace std;
class Student {
public:
// static variable declaration
static int subject_code;
};
// static variable definition and initialization
int Student::subject_code = 13;
int main() {
// access static variable
cout << Student::subject_code << endl;
return 0;
}
```

```
// Output: 13
```

You can see that we have successfully accessed the static variable without creating an object of the class.

Why static?

While implementing OOP, we may be faced with situations where all the objects of a class need to share common data. In such cases, we store such data in static variables.

When we declare a static variable, all objects of the class share the same static variable. The static variables and functions belong to the class (rather than objects). And we don't need to create objects of the class to access the static variables and functions.

```cpp
#include <iostream>
using namespace std;
class Company {
public:
static string name;
};
// static variable definition
string Company::name;
int main() {
Company::name = "Programiz";
cout << "Name: " << Company::name << endl;
return 0;
}
```

Figure: Working of static variables

Here, the static variable `name` is common to all objects of the class `Company`.

However, when we declare a non-static variable, all objects will have separate copies of the variable.

```
#include <iostream>
using namespace std;
class Company {
public:
string name;
};
int main() {
Company object1;
Company object2;
object1.name = "Programiz";
object2.name = "Programiz PRO";
cout << "Name for object1: " << object1.name << endl;
cout << "Name for object2: " << object2.name << endl;
return 0;
}
```

Figure: Working of non-static variables

Here, both `object1` and `object2` will have separate copies of the variable `name`. And they are different from each other.

Constructor Overloading

C++ Constructors

Basically, a constructor is like a member function of a class that has the same name as the class but no return type. A constructor is automatically called when we create an object of the class. For example,

```
#include <iostream>
using namespace std;
class Sample {
public:
// default constructor with no arguments
Sample() {
cout << "Object created!" << endl;
}
};
int main() {
// create an object of the Sample class
Sample sample1;
return 0;
}
// Output: Object created!
```


Here, `Sample()` is a constructor of the `Sample` class and is called automatically the moment we create the `sample1` object. It is a default constructor since it takes no arguments.

Parameterized Constructor

Constructors can also take parameters. For example,

```
#include <iostream>

using namespace std;

class Sample {

public:

// constructor with integer parameter

Sample (int num) {

cout << "Constructor Parameter: " << num << endl;

}

};

int main() {

// create object of Sample

// supply 9 as argument to its constructor

Sample sample(9);

return 0;

}

// Output:

// Constructor Parameter: 9
```

Now, let's see how we can combine these two programs and overload these constructors.

Constructor Overloading

Similar to function overloading, overloaded constructors have the same name (name of the class) but different numbers or types of arguments.

Let's see an example.

```
#include <iostream>
using namespace std;
class Sample {
public:
// default constructor with no arguments
Sample() {
cout << "Default constructor!" << endl;
}
// parameterized constructor with an integer argument
Sample (int num) {
cout << "Second Constructor Parameter: " << num << endl;
}
// constructor with 2 parameters
Sample (int num1, double num2) {
cout << "Third Constructor Parameters: ";
cout << num1 << " and " << num2 << endl;
}
};
int main() {
// call the default constructor
Sample sample1;
// call the constructor with a single int argument
Sample sample2(9);
// call the constructor with two arguments
Sample sample3(9, 9.5);
return 0;
}
```

Output

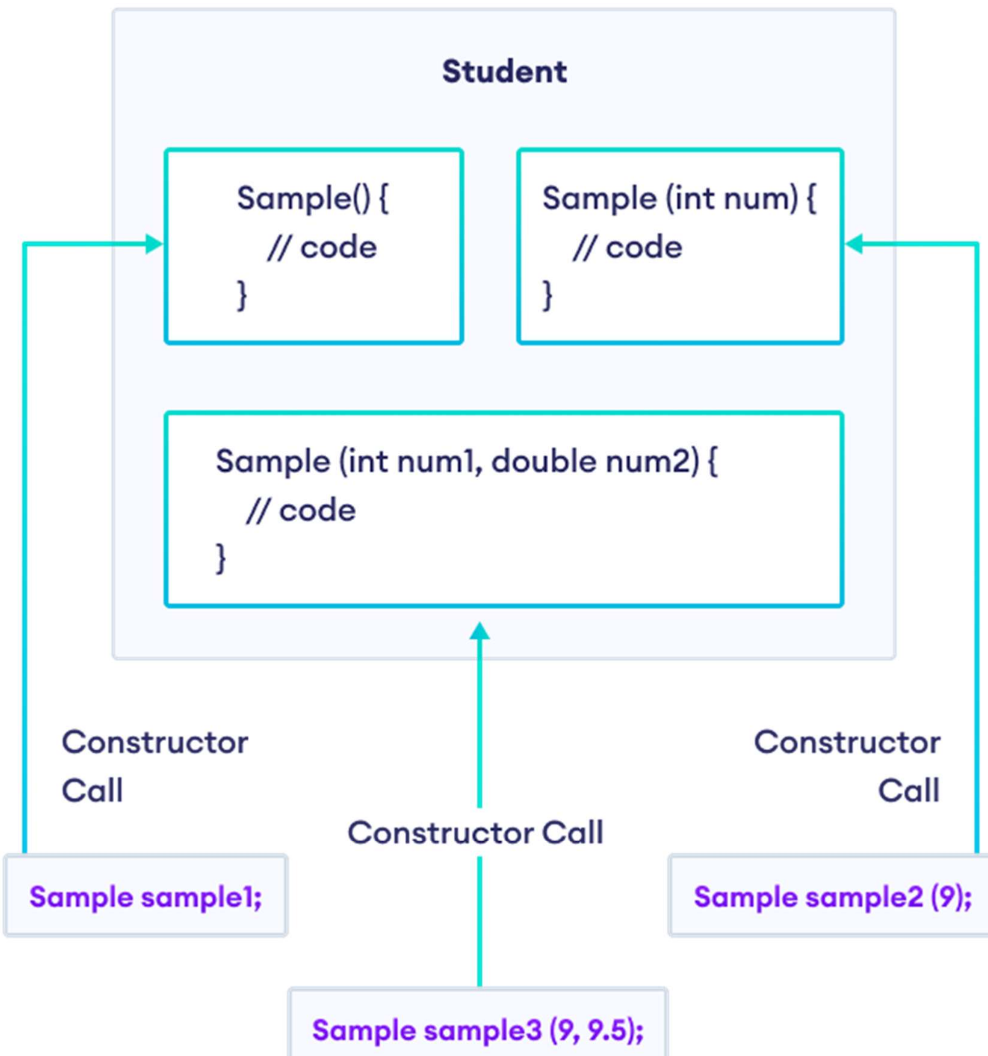
```
Default constructor!
Second Constructor Parameter: 9
Third Constructor Parameters: 9 and 9.5
```

Here, we have overloaded 3 constructors in the `Sample` class:

- `Sample()` - a default constructor with no parameters
- `Sample(int num)` - a parameterized constructor with an integer parameter
- `Sample(int num1, double num2)` - a parameterized constructor with two parameters: one integer and one `double`.

We can call the desired constructor by supplying the appropriate argument(s) when creating objects of the class.

The image below shows how:



Why Overload Constructors?

A lot of the times, we may want to initialize objects in different ways. Sometimes, we may want an object to have default values for its member variables.

At other times, we may want to initialize the members with different values. This can easily be achieved through constructor overloading.

So, with constructor overloading, we can make our classes and objects more dynamic and flexible. It can also make our code shorter and look cleaner.

Imagine having to assign custom values to different objects. Without constructor overloading, we'd have to either assign the values using the "." operator:

```
object1.variable1 = value1;
object1.variable2 = value2;

object2.variable1 = value3;
object2.variable2 = value4;
```

Or we'd have to rely on setter functions to assign those values:

```
object1.set_variable1(value1);
object1.set_variable2(value2);

object2.set_variable1(value3);
object2.set_variable2(value4);
```

With constructor overloading, we can condense these four lines of codes into two, while also having the freedom to initialize an object with default values:

```
// objects with custom values
Sample object1(value1, value2);
Sample_Class object2(value3, value4);

// objects with default values
Sample object3, object4;
```

As you can see, this process is far less tedious and is much easier on the eyes. So, it is always a good idea to overload constructors if our program demands flexibility with its classes.

Template

This is a powerful feature that allows us to write generic programs i.e., programs that include codes that can work with any data type.

There are two ways we can implement templates:

- Function Templates
- Class Templates

Function Templates

Function templates are generic functions that can work with multiple data types. For example,

```
template <typename T>
T add (T num1, T num2) {
    return (num1 + num2);
}
```

Here, we have created a function template named `add ()`. The template definition consists of the following parts:

- `template` - keyword used to declare a function template
- `typename` - keyword that is part of the function template syntax
- `T` - template argument that represents the data type

Now we can use this function with any type of data.

1. Working with int data

```
// call function template with int data
add <int> (2, 3);
```

Here, the template argument `T` will be `int` and `num1` and `num2` will be 2 and 3 respectively.

2. Working with double data

```
// call function template with double data
add <double> (5.56, 9.34);
```

In this case, the template argument `T` will be `double` and `num1` and `num2` will be 5.56 and 9.34 respectively.

Note: We can also omit the data type while calling a function template. For example, `add(2, 3)` and `add(5.56, 9.34)`. However, it is a good practice to include the data type during the function call.

Example: Function Template

```
#include <iostream>

using namespace std;

template <typename T>

T add (T num1, T num2) {

return num1 + num2;

}

int main() {

// call function template with int data

int result1 = add<int>(2, 3); // call function template with double data

double result2 = add<double>(5.56, 9.34);

cout << "2 + 3 = " << result1 << endl;

cout << "5.56 + 9.34 = " << result2 << endl;

return 0;

}
```

Output

```
2 + 3 = 5
5.56 + 9.34 = 14.9
```

As you can see, we are able to use the same function to work with both the integer data and `double` data.

Class Templates

Similar to functions, we can also create class templates to work with different types of data. For example,

```
template <class T>
class Number {
    public:
        T var1;
        T var2;
};
```

Notice that we have used the keyword `class` instead of `typename` in the syntax above. We can also use the keyword `typename` instead.

```
template <typename T>
class Number {...};
```

So don't get confused. We will be using class for all our examples.

Now, we can use this class to work with any type of data by creating objects with the appropriate data type. For example,

```
// object that works with integer data
Number<int> integer_object;
```

```
// object that works with double data
Number<double> double_object;
```

Note: Unlike with function templates, we must supply the data type of the parameters when creating objects of class templates.

```
// error: missing template arguments
Number integer_object;
```

Example: Class Templates

```
#include <iostream>

using namespace std;

// class template

template <class T>
```

```
class Multiplication {
public:
// variable of type T
T multiplier;
// constructor initializer list
Multiplication(T multi) : multiplier(multi) {}
// function that returns product of
// multiplier variable and the num argument
T multiply(T num) {
return num * multiplier;
}
};
int main() {
// create object with int data
Multiplication<int> num_int(3);
int result1 = num_int.multiply(9);
// create object with double data
Multiplication<double> num_double(5.7);
double result2 = num_double.multiply(13.2);
cout << "Product with int: " << result1 << endl;
cout << "Product with double: " << result2 << endl;
return 0;
}
```


Output

```
Product with int: 27
Product with double: 75.24
```

Why Templates?

1. Code Reusability

We can write code that will work with different types of data. For example,

```
int add(int num1, int num2) {
    return num + num2;
}
```

Here, the function only works if we pass `int` data to this. If we want to perform addition of `double` values, we have to create another function.

However, with templates, we can use one function and use it with any type of data.

```
template <typename T>
T add(T num1, T num2) {
    return num1 + num2;
}
```

2. Type Checking

The template parameter, `T`, provides information about the type of data used in the template code. For example,

```
Template_Class<string> obj("Hello");
```

Here, this object will only work with `string` data. Now, if we try to pass a value other than string, we will get an error.

Exception handling

https://www.programiz.com/cpp-programming/exception-handling