Unit:4

Structure vs class: In C++, a structure is the same as a class except for a few differences. The most important of them is security. A Structure is not secure and cannot hide its implementation details from the end user while a class is secure and can hide its programming and designing details.

Structure: We often come around situations where we need to store a group of data whether of similar data types or non-similar data types. We have seen [Arrays in C++](https://www.geeksforgeeks.org/arrays-in-c-cpp/) which are used to store set of data of similar data types at contiguous memory locations.

Unlike Arrays, Structures in C++ are user defined data types which are used to store group of items of non-similar data types.

**What is a structure?**

A structure is a user-defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type. The ‘struct’ keyword is used to create a structure. The general syntax to create a structure is as shown below:

struct structureName{

member1;

member2;

member3;

.

.

.

memberN;

};

Structures in C++ can contain two types of members:

* **Data Member**: These members are normal C++ variables. We can create a structure with variables of different data types in C++.
* **Member Functions**: These members are normal C++ functions. Along with variables, we can also include functions inside a structure declaration.

struct Student

{

char stuName;

int stuRollNo;

int stuAge;

};

A program Explaining Structure

#include <iostream.h>

struct Person

{

int citizenship;

int age;

};

int main(void) {

struct Person p1,p2;

p1.citizenship = 1;

p1.age = 27;

p2.citizenship = 2;

p2.age = 37;

cout << "Person citizenship: " << p1.citizenship << endl;

cout << "Person age: " << p1.age << endl;

cout << "Person citizenship: " << p2.citizenship << endl;

cout << "Person age: " << p2.age << endl;

return 0;

}

Output

Person citizenship: 1

Person age: 27

Person citizenship: 2

Person age: 37

A program Explaining Array in Structure

#include <iostream>

struct Point {

    int x, y;

};

int main()

{

    // Create an array of structures

    struct Point arr[10];

    // Access array members

    arr[0].x = 10;

    arr[0].y = 20;

    cout << arr[0].x << " " << arr[0].y;

    return 0;

}

Output

10 20

# Access Specifiers

In C++, there are three access specifiers:

* public - members are accessible from outside the class
* private - members cannot be accessed (or viewed) from outside the class
* protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about [Inheritance](https://www.w3schools.com/cpp/cpp_inheritance.asp) later.

In the following example, we demonstrate the differences between public and private members:

“Members of a class are private by default and members of a struct are public by default.”

#include <stdio.h>

class Test {

private:

    int x; // x is private

};

int main()

{

  Test t;

  t.x = 20; // compiler error because x is private

  getchar();

  return 0;

}

Above Program fails in compilation. But when we was using Struct it was public as shown in the bellow program.

#include <stdio.h>

 struct Test {

    int x; // x is public

};

int main()

{

  Test t;

  t.x = 20; // works fine because x is public

  getchar();

  return 0;

}

Difference between Public and Private

All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

#include <iostream>

// class definition

class Circle {

public:

    double radius;

    double compute\_area()

    {

        return 3.14 \* radius \* radius;

    }

};

// main function

int main()

{

    Circle obj;

    // accessing public data member outside class

    obj.radius = 5.5;

    cout << "Radius is: " << obj.radius << "\n";

    cout << "Area is: " << obj.compute\_area();

    return 0;

}

**Output:**

Radius is: 5.5

Area is: 94.985

In the above program, the data member radius is public so we are allowed to access it outside the class.

The class members declared as private can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

#include <iostream>

class Circle {

    // private data member

private:

    double radius;

    // public member function

public:

    void compute\_area(double r)

    {

        // member function can access private

        // data member radius

        radius = r;

        double area = 3.14 \* radius \* radius;

        cout << "Radius is: " << radius << endl;

        cout << "Area is: " << area;

    }

};

// main function

int main()

{

    // creating object of the class

    Circle obj;

    // trying to access private data member

    // directly outside the class

    obj.compute\_area(1.5);

     return 0;

}

**Output:**

Radius is: 1.5

Area is: 7.065

|  |  |
| --- | --- |
| **PUBLIC** | **PRIVATE** |
| All the class members declared under public will be available to everyone. | The class members declared as private can be accessed only by the functions inside the class. |
| The data members and member functions declared public can be accessed by other classes too. | Only the member functions or the friend functions are allowed to access the private data members of a class. |
| The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class. | They are not allowed to be accessed directly by any object or function outside the class. |

**Friend Functions**

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows −

class Box

{

private:

double width;

public:

double length;

friend void printWidth( Box box );

void setWidth( double wid );

};

#include <iostream>

class Box {

double width=10.0;

public:

friend void printWidth( Box box );

void setWidth( double wid );

};

// Member function definition

void Box::setWidth( double wid )

{

width = wid;

}

// Note: printWidth() is not a member function of any class.

void printWidth( Box box )

{

/\* Because printWidth() is a friend of Box, it can

directly access any member of this class \*/

cout << "Width of box : " << box.width <<endl;

}

// Main function for the program

int main()

{

Box box;

// set box width without member function

box.setWidth(10.0);

// Use friend function to print the wdith.

printWidth( box );

return 0;

}

Output:

Width of box : 10

OOPs Concepts 🡪 Polymorphism 🡪 (1) Function Overloading (2) Operator Overloading

**Operator Overloading**

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special feature which is known as Operator function.

#include <iostream.h>

Class space

{

Private:

int x;

int y;

int z;

Public:

void getdata(int a , int b , int c);

void display(void);

void operator – ();

};

void space : : getdata (int a , int b , int c)

{

x=a;

y=b;

z=c;

}

void space : : display (void)

{

cout<<x;

cout<<y;

cout<<z;

}

void space : : operator – ()

{

x= - x;

y= - y;

z= - z;

} Space🡪 s

X,y,z

void main()

{

space s;

s.getdata(10,-20,30);

cout<<”s:”;

s.display();

- s;

Getdata()

cout<<”- s:”;

s.display();

}

**OUTPUT**

s: -- s

10,-20,30

-s:

-10,20,-30

**Constructors**

C++ provides a special member function called **constructor** which enables an object to initialize itself when it is created. This is known as automatic initialization of objects. It also provides another member function called **destructor** that destroys the objects when they are no longer required.

A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the value of data members of the class.

Syntax of using a constructor is given bellow:

class **bca**

{

int m, n;

public:

**bca**(void);

--------------;

\_\_\_\_\_\_\_\_;

};

bca : : bca(void)

{

m=0,n=0;

}

When a class contains a constructor it is guaranteed that an object created by the class will be initialed automatically. For example:

bca x1; // object x1 created

The declaration not only creates the object x1 of type bca but also initialize its data members m and n to zero. There is no need to write any statement to invoke the constructor function.

class Car {        // The class  
  public:          // Access specifier  
    string brand;  // Attribute  
    string model;  // Attribute  
    int year;      // Attribute  
    Car(string x, string y, int z)

{ // Constructor with parameters  
      brand = x;  
    model = y;  
       year = z;  
    }  
};

int main() {

// Create Car objects and call the constructor with different values  
  Car carObj1("BMW", "X5", 1999);  
  Car carObj2("Ford", "Mustang", 1969);  
  // Print values

  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";  
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";  
  return 0;  
}

Just like functions, constructors can also be defined outside the class. First, declare the constructor inside the class, and then define it outside of the class by specifying the name of the class, followed by the scope resolution :: operator, followed by the name of the constructor (which is the same as the class):

class Car {        // The class  
  public:          // Access specifier  
    string brand;  // Attribute  
    string model;  // Attribute  
    int year;      // Attribute  
    Car(string x, string y, int z); // Constructor declaration  
};  
// Constructor definition outside the class  
Car::Car(string x, string y, int z)

{  
  brand = x;  
  model = y;  
  year = z;  
}  
int main() {  
  // Create Car objects and call the constructor with different values  
  Car carObj1("BMW", "X5", 1999);  
  Car carObj2("Ford", "Mustang", 1969);  
  // Print values  
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";  
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";  
  return 0;  
}

**Inheritance**: When we are sharing the properties from one class to another class than we use inheritance. It helps in reusability. Parent class is known as Base class while child class is known as derived class. It may be of 5 types.

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance

B

A

The above diagram shows the single inheritance. A derived class can be defined by specifying its relationship with the base class in addition to its own detail. The general form of defining a derived class is given bellow:

**Class** derived-class-name : visibility-mode base class-class-name

{

………………//

………………//

………………//

}

When we use the syntax for above figure:

**Class** B : public A

{

………………//

………………//

………………//

}

// Single Inheritance : public

#include<iostream.h>

class B

{

private:

int a;

public:

int b;

void get\_ab();

int get\_a(void);

void show\_a(void);

};

class D : public B

{

int c;

public:

void mul(void)

void display(void)

};

void B : : get\_ab(void)

{

a=5;

b=10;

}

void B : : get\_a()

{

return a;

}

void B : : show\_a()

{

Cout<<”a=”<<a;

}

Void D :: mul()

{

c=b\*get\_a();

}

Void D :: display()

{

cout<<”a=”<<get\_a();

cout<<”b=”<<b;

cout<<”c=”<<c;

}

Int main()

{

D d;

d.get\_ab();

d.mul();

d.show\_a();

d.display();

d.b=20;

d.mul();

d.display();

return 0;

}

**Output**

a=5

a=5

b=10;

c=50

a=5

b=20

c=100