Name: Kabir Dhruw
Reg No.: 20BCY10077

OOPC++

First Module

- Why object-oriented programming?
- Characteristics of object-oriented language: classes and objects
- Encapsulation
- Data abstraction
- Inheritance
- Polymorphism
- Merits and Demerits of object-oriented programming
- UML
- Class diagram of OOP
- Inline function
- Default argument function
- Exception handling (Standard)
- Reference: independent reference – function returning reference – pass by reference

Object Oriented Programming

Object Oriented Programming (OOP) is a programming pattern that allows you to package together data states and functionality to modify those data states, while keeping the details hidden away. As a result, code with OOP design is flexible, modular, and abstract. This makes it particularly useful when you create larger programs.

In C++, you can apply OOP in your code with classes and objects. And the C++ objects you create will have states and functionality.

There are four major benefits to object-oriented programming:

- Encapsulation: in OOP, you bundle code into a single unit where you can determine the scope of each piece of data.
- Abstraction: by using classes, you are able to generalize your object types, simplifying your program.
- Inheritance: because a class can inherit attributes and behaviours from another class, you are able to reuse more code.

- Polymorphism: one class can be used to create many objects, all from the same flexible piece of code.

Merits of OOP:

1. Improved software-development productivity. Object-oriented programming is modular, as it provides separation of duties in object-based program development. It is also extensible, as objects can be extended to include new attributes and behaviours. Objects can also be reused within an across applications.

2. Improved software maintainability. For the reasons mentioned above, object-oriented software is also easier to maintain. Since the design is modular, part of the system can be updated in case of issues without a need to make large changes.

3. Faster development, Reuse enables faster development. Object-oriented programming languages come with rich libraries of objects, and code developed during projects is also reusable in future projects.

Demerits of OOP:

1. Steep learning curve: The thought process involved in object-oriented programming may not be natural for

some people, and it can take time to get used to it. It is complex to create programs based on interaction of objects.
2. Larger program size, Object-oriented programs typically involve more lines of code than procedural programs.

3. Not suitable for all types of problems, there are problems that lend themselves well to functional-programming style, logic-programming style, or procedure-based programming style, and applying object-oriented programming in those situations will not result in efficient programs.

Class diagram

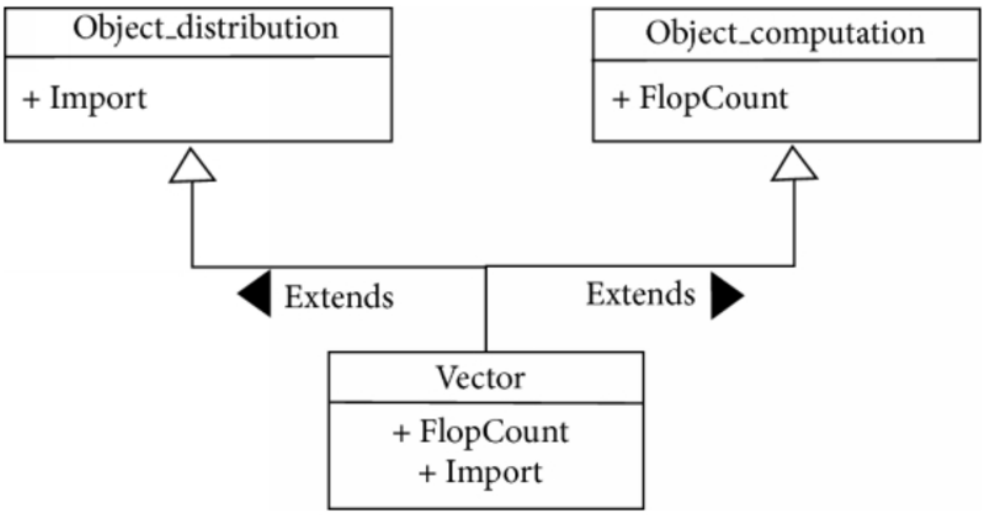
Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application. Class diagram describes the attributes and operations of a class and also the constraints imposed on the system.

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.

Class diagram is also considered as the foundation for component and deployment diagrams. Class diagrams are not only used to visualize the static view of the system but they are also used to construct the executable code for forward and reverse engineering of any system.

Or we can also conclude that, class diagram can be used to

- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object-oriented languages.



Inline Function

Inline function is used in C++ to reduce the function call overhead. It is expanded in line when called. It is only a request to the compiler and not a command. Some times compiler do not perform inlining because,

1) If a function is containing a loop
2) If a function contains switch statement.
3) If a function is recursive.

Advantages of inline functions are:

1) It saves return call from a function.
2) It is used in embedded system because inline can yield less code than the function call preamble and return.
3) Function call overhead doesn’t occur.

Disadvantages:

1) Due to high number of inline functions the size of binary file may become large.
2) It may cause thrashing.
3) It also consumes additional registers.

```cpp
#include <iostream>
using namespace std;
inline int cube(int s)
{
    return s*s*s;
}
int main()
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
}
```

Default Arguments

In C++ we are allowed to assign default values to a function's parameter which is useful in case a matching argument is not passed in the function call statement. It allows a function to be called without providing one or more trailing arguments. The default values are passed at the time of function declaration.

int hello(float a, int b, float c=0,10);

```cpp
#include<iostream>
using namespace std;
int sum(int x = 10, int y = 20, int z = 30){
    return x+y+z;
}
int main(){
    cout<<sum()<<"\n";
    cout<<sum(5)<<"\n";
    cout<< sum(5,10);
    return 0;
}
```

```
60
55
45

...Program finished with exit code 0
Press ENTER to exit console.
```

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

int sum(int x, int y, int z=0, int w=0)
{
    return (x + y + z + w);
}

int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```

```
25
50
80


...Program finished with exit code 0
Press ENTER to exit console.
```

Pass by reference

Pass by reference means to pass the reference of an argument in the calling function to the corresponding formal parameter of the called functions. The called function can modify the value of the argument by using its reference passed in.

Pass by reference is more efficient than pass-by-value, because it does not copy the arguments. The major difference between pass- by-value and pass-by-reference is that modifications made to arguments passed in by reference in the called function have effect in the calling functions, whereas modifications made to argument value passed in by value in the called function cannot affect the calling function.

```
1  #include<iostream>
2  using namespace std;
3
4  void swapnum(int &i, int &j) {
5    int temp = i;
6    i = j;
7    j = temp;
8  }
9
10 int main() {
11   int a = 10;
12   int b = 20;
13
14   swapnum(a, b);
15   cout << "A is" << a<< " and B is "<<b;
16   return 0;
17 }
```

```
A is20 and B is 10

...Program finished with exit code 0
Press ENTER to exit console.
```

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

void swapNums(int &x, int &y) {
  int z = x;
  x = y;
  y = z;
}

int main() {
  int firstNum = 10;
  int secondNum = 20;

  cout << "Before swap: " << "\n";
  cout << firstNum << "\n"<<secondNum << "\n";
  swapNums(firstNum, secondNum);
  cout << "After swap: " << "\n";
  cout << firstNum <<"\n"<< secondNum << "\n";

  return 0;
}
```

```
Before swap:
10
20
After swap:
20
10


...Program finished with exit code 0
Press ENTER to exit console.
```

Second module

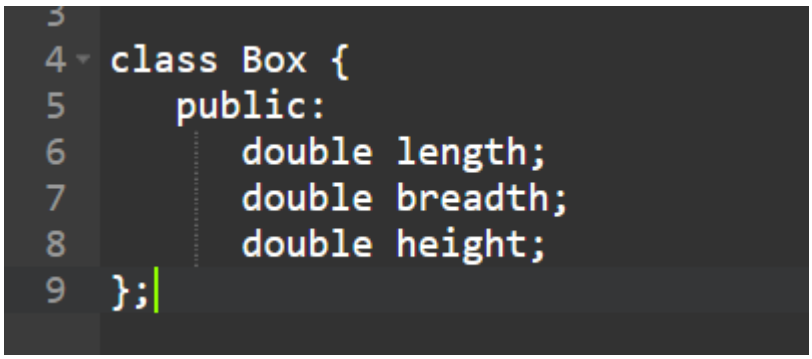
Topics:

- Definition of classes
- access specifier
- class versus structure
- constructor
- destructor
- copy constructor and its importance
- array of objects
- dynamic objects
- friend function
- friend class
- container class

Class

A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword class as follows:



Object

An object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated memory is allocated.

Box B1;
Box B2;

Here B1 and B2 are two Objects, both of them have their own copy of data members.

Access specifiers in C++

In C++, there are three access specifiers,

1) public: Members in this access specifier can be accessed from outside the class.

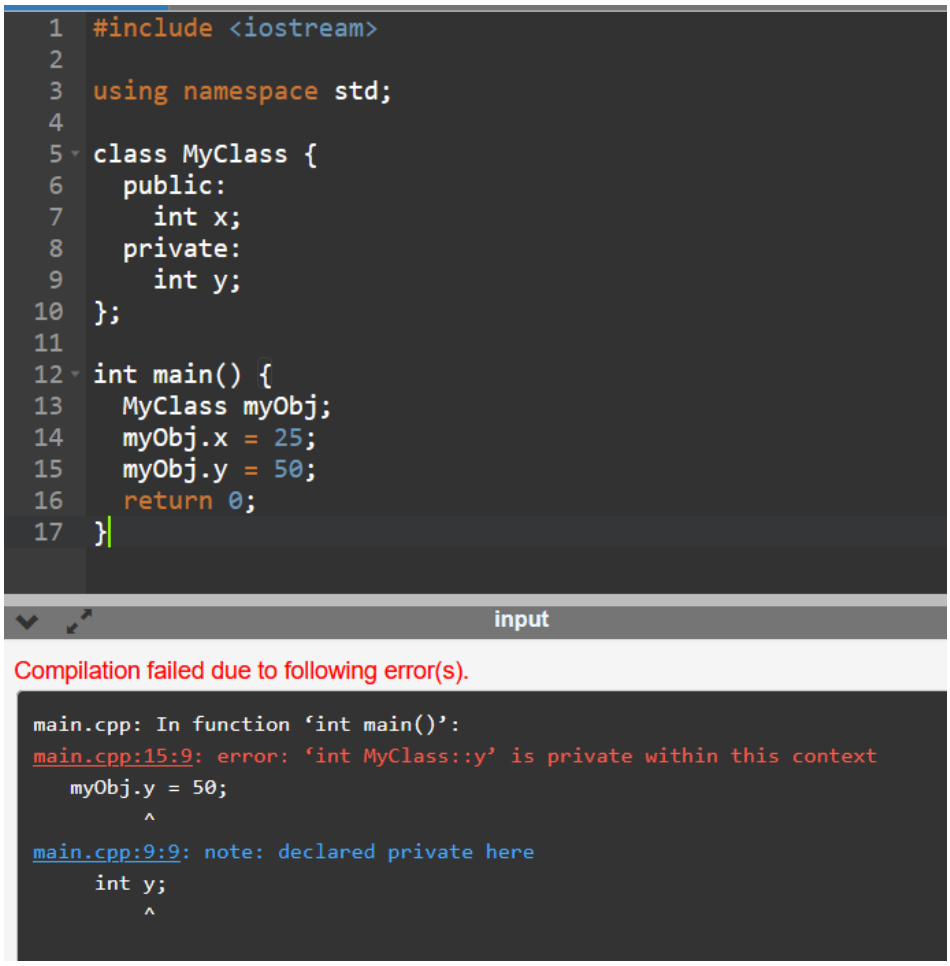
2) Private: Members cannot be accesses from outside the class.

3) Protected: Members cannot be accessed from outside the class, they can be accessed in inherited classes.

By default, all the members of a class are private if you don’t specify and access specify.

It is possible to access private members of a class using a public method inside the same class.
It is considered good practice to declare your class attribute as private as this will reduce the possibility of yourself to mess up the code.

In the example below, we can see that we cannot access the private member from a class.



class versus structure

Structure	Class
It is a value type.	It is a reference type.
Its object is created on the stack memory.	Its object is created on the heap memory.
It does not support inheritance.	It supports inheritance.
The member variable of structure cannot be initialized directly.	The member variable of class can be initialized directly.
It can have only parameterized constructor.	It can have all the types of constructor and destructor.

Constructor

To customize how class members are initialized, or to invoke functions when an object of your class is created, we can define a constructor. A constructor has the same name as the class and no return value. You can define as many overloaded constructors as needed to customize initialization in various ways. Typically, constructors have public accessibility so that code outside the class definition or inheritance hierarchy can create objects of the class. But you can also declare a constructor as private or protected.

Constructures are of 3 types:

- Default constructor
- Parametrized constructor
- Copy constructor

Default Constructor

Default Constructor typically have no parameters, but they can have parameters with default values. Default constructors are one of the special member function. If no constructors are declared in a class, the compiler provides an implicit inline default constructor. Or we can simply say that, Default constructor is the

constructor which doesn’t take any argument. It has no parameters.

main.cpp

```cpp
#include <iostream>
using namespace std;
class construct
{
public:
    int a, b;
    construct()
    {
        a = 10;
        b = 20;
    }
};
int main()
{
    construct c;
    cout << "a: " << c.a << endl
         << "b: " << c.b;
    return 1;
}
```

```
a: 10
b: 20

...Program finished with exit code 1
Press ENTER to exit console.
```

Parameterized Constructor

In C++, a constructor with parameters is known as a parameterized constructor. This is the preferred method to initialize member data.
It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

Uses of Parameterized constructor:

- It is used to initialize the various data elements of different objects with different values when they are created.
- It is used to overload constructors.

```cpp
#include <iostream>
using namespace std;
class Line {
   public:
      void setLength( double len );
      double getLength( void );
      Line(double len);
   private:
      double length;
};
Line::Line( double len) {
   cout << "Object is being created, length = " << len << endl;
   length = len;
}
void Line::setLength( double len ) {
   length = len;
}
double Line::getLength( void ) {
   return length;
}
int main() {
   Line line(10.0);
   cout << "Length of line : " << line.getLength() <<endl;
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;
   return 0;
}
```

input

```
Object is being created, length = 10
Length of line : 10
Length of line : 6


...Program finished with exit code 0
Press ENTER to exit console.
```

Copy Constructor

A copy constructor initializes an object by copying the member values from an object of the same type. If your class members are all simple types such as scalar values, the compiler-generated copy constructor is sufficient and you do not need to define your own. If your class requires more complex initialization, then you need to implement a custom copy constructor.

Different situations when a copy constructor may be called:

- When an object of the class is returned by value.
- When an object of the class is passed (to a function) by value as an argument.
- When an object is constructed based on another object of the same class.
- When the compiler generates a temporary object.
-

Whenever we define one or more non-default constructors (with parameters) for a class, a default constructor (without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it’s considered to be the best practice to always define a default constructor.

main.cpp

```cpp
#include "iostream"
using namespace std;
class point
{
    private:
    double x, y;
    public:
    point (double px, double py)
  {
    x = px, y = py;
  }
};

int main(void)
{
  point a[10];
  point b = point(5, 6);
}
```

input

Compilation failed due to following error(s).

```
main.cpp: In function ‘int main()’:
main.cpp:16:13: error: no matching function for call to ‘point::point()’
   point a[10];
             ^
```

Some important notes on copy constructor:

1. Can we make copy constructor private?

    Yes, a copy constructor can be made private. When we make a copy constructor private in a class, objects of that class become non-copyable. This is particularly useful when our class has pointers or dynamically allocated resources. In such situations, we can either write our own copy

constructor like above String example or make a private copy constructor so that users get compiler errors rather than surprises at runtime.

2. Why argument to a copy constructor must be passed as a reference?

A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So, if we pass an argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore, compiler doesn't allow parameters to be passed by value.

3. When no copy constructor is found by the compiler, compile creates its own copy constructor.

Constructor Overloading

We can have more than one constructor in the class with the same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to function overloading. It is when we have more than one

constructor present in the class, but they differ by the number of arguments present.

Properties of constructors to keep in mind while overloading a constructor:

1. Constructors essentially need to have the same name as of class.
2. Constructors don't have any return type.
3. Constructors are always declared as public.
4. While creating the object argument must be passed according to the requirement to let the compiler know which constructor to call.

When we add more than one constructor to a class, we call it multiple constructors.

Constructor overloading is very much similar to function overloading.

main.cpp

```cpp
#include <iostream>
using namespace std;
class construct
{
public:
    float area;
    construct()
    {
        area = 0;
    }
    construct(int a, int b)
    {
        area = a * b;
    }

    void disp()
    {
        cout<< area<< endl;
    }
};

int main()
{
    construct o;
    construct o2( 10, 20);
    o.disp();
    o2.disp();
```

Dynamic allocation of array

A dynamic array is quite similar to a regular array, but its size is modifiable during program runtime. Dynamic array elements occupy a contiguous block of memory.

Dynamic allocation of array: -

During runtime when user can allocate memory manually. As, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, dynamic allocation of array is used.

To allocate memory dynamically, following library functions are used:

- malloc (),
- calloc (),
- realloc () and
- free ()

These functions are defined in the <stdlib.h> header file.

main.cpp

```cpp
#include<iostream>
using namespace std;
int main() {
    int x, n;
    cout << "Enter the number of items:" << "\n";
    cin >>n;
    int *arr = new int(n);
    cout << "Enter " << n << " items" << endl;
    for (x = 0; x < n; x++) {
        cin >> arr[x];
    }
    cout << "You entered: ";
    for (x = 0; x < n; x++) {
        cout << arr[x] << " ";
    }
    return 0;
}
```

```
Enter the number of items:
2
Enter 2 items
5
9
You entered: 5 9

...Program finished with exit code 0
Press ENTER to exit console.
```

main.cpp

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

int main(void) {
    int x;
    int *array{ new int[5]{ 10, 7, 15, 3, 11 } };
    cout << "Array elements: " << endl;
    for (x = 0; x < 5; x++) {

        cout << array[x] << endl;
    }
    return 0;
}
```

```
Array elements:
10
7
15
3
11


...Program finished with exit code 0
Press ENTER to exit console.
```

Friend Function

A friend function in C++ is a function that is preceded by the keyword “friend”. When the function is declared as a friend, then it can access the private and protected data members of the class.

A friend function is declared inside the class with a friend keyword preceding as shown below.

main.cpp

```cpp
#include <iostream>
#include <string>
using namespace std;
class Area{
   int length,breadth,area;
   public:
   Area(int length,int breadth):length(length),breadth(breadth)
   {}
   void calcArea(){
      area = length * breadth;
   }
   friend class printClass;
};
class printClass{
   public:
   void printArea(Area a){
      cout<<"Area = "<<a.area;
   }
 };
int main(){
   Area a(10,15);
   a.calcArea();
   printClass p;
   p.printArea(a);
   return 0;
}
```

input

```
Area = 150

...Program finished with exit code 0
Press ENTER to exit console.
```

main.cpp

```cpp
#include <iostream>
#include <string>
using namespace std;
class sample{
   int length, breadth;
   public:
   sample(int length, int breadth):length(length),breadth(breadth)
   {}
   friend void calcArea(sample s);
};
void calcArea(sample s){
   cout<<"Area = "<<s.length * s.breadth;
   }
int main()
   {
      sample s(10,15);
      calcArea(s);
      return 0;
}
```

input

```
Area = 150

...Program finished with exit code 0
Press ENTER to exit console.
```

Third Module

- Polymorphism
- compile time polymorphism
- function overloading
- operator overloading
- Inheritance
- types of inheritance
- constructors and destructors in inheritance

Polymorphism

Polymorphism means having 'many forms'. It is considered to be one of the major aspects of Object-Oriented Programming in C++. A real-life example of polymorphism can be of a person who is a father, son, husband and employee at the same time. In C++, polymorphism is mainly divided into two categories -

1. Compile time polymorphism
2. Runtime polymorphism

Compile time polymorphism:

This type of polymorphism is achieved by function overloading or operator overloading.

Function Overloading

When there is more than one function with same name, they are said to be overloaded and the process is called function overloading. The functions are either given different number of parameters or different type of arguments or both.

main.cpp

```cpp
#include<iostream>
using namespace std ;
class Addition {
    public :
    void sum(int x, int y){
        cout<<"Sum = "<<x+y<<"\n";
    }
    void sum(int x, int y, int z){
        cout<<"Sum = "<<x+y+z<<"\n";
    }
};
int main() {
    Addition o1 ;
    o1.sum(2,4);
    o1.sum(6,8,10);
    return 0 ;
}
```

```
Sum = 6
Sum = 24


...Program finished with exit code 0
Press ENTER to exit console.
```

Operator Overloading

In C++, certain operators are also modified to make them work in desired manner. This is known an operator overloading. Thus, in C++ we can make operators to work for user defined classes.

Almost all operators can be overloaded except some.

1. .
2. ::
3. size of
4. ?;

main.cpp

```cpp
#include <iostream>
using namespace std ;
class complex {
    int real,imag ;
    public :
    int r,i ;
    void SetData() {
        cout<<"Enter real part : ";
        cin>>real ;
        cout<<"Enter imaginary part : ";
        cin>>imag ;
        r = real ;
        i = imag ;
    }
    void GetData() {
        cout<<"Sum = "<<r<<" + "<<i<<"i\n";
    }
    complex operator + (complex o1) {
        complex temp ;
        temp.r = r + o1.r ;
        temp.i = i + o1.i ;
        return temp ;
    }
};
int main() {
    complex c1, c2, C ;
    c1.SetData() ;
    c2.SetData() ;
    C = c1 + c2 ;
    C.GetData() ;
}
```

```
Enter real part : 3
Enter imaginary part : 7
Sum = 5 + 11i
```

Runtime polymorphism: -

This type of polymorphism is achieved by function overriding.

Function Overriding

In function overriding, a function is derived class has a definition for at least one of the member functions of the base class.

main.cpp

```cpp
#include <iostream>
using namespace std ;
class base {
    public:
    void display(){
        cout<<"You are in the base class.\n";
    }
};
class derived : public base {
  public :
  void display() {
      cout<<"Welcome to the derived class.\n";
  }
};
int main() {
    derived d ;
    d.display();
}
```

```
Welcome to the derived class.


...Program finished with exit code 0
Press ENTER to exit console.
```

Destructor

A destructor is a special member function that is called when the lifetime of an object ends. The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.

Properties of Destructor:

- Destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.

main.cpp

```
4
5   class String {
6   private:
7       char* s;
8       int size;
9
10  public:
11      String(char*);
12      ~String();
13  };
14
15  String::String(char* c)
16  {
17      size = strlen(c);
18      s = new char[size + 1];
19      strcpy(s, c);
20  }
21  String::~String() { delete[] s; }
22
23
```