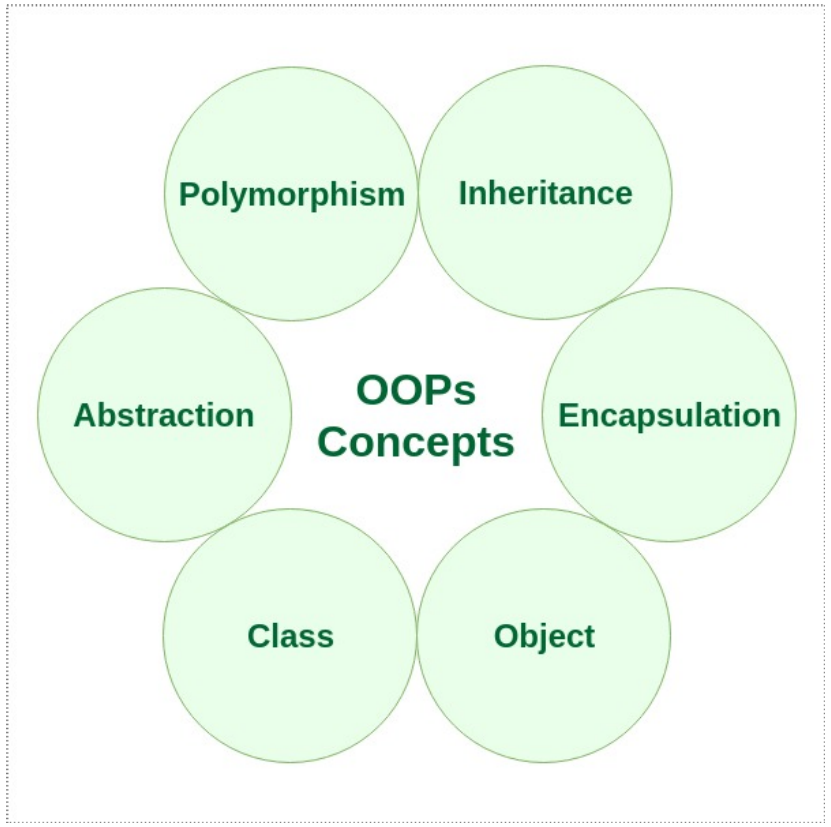
OOP

WEEK 4

Object-oriented programming

Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

OOP is an approach or a programming pattern where the programs are structured around objects rather than functions and logic. It makes the data partitioned into two memory areas, i.e., data and functions, and helps make the code flexible and modular.



Class

The building block of C++ that leads to Object-Oriented programming is a Class. 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 class is like a blueprint for an object.

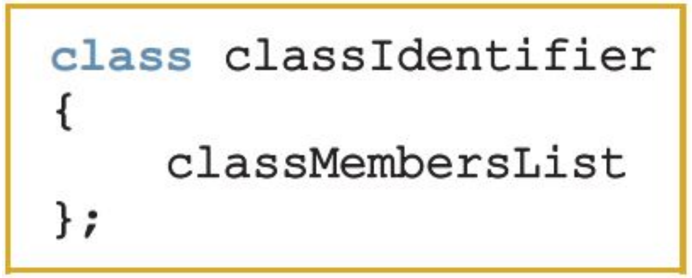
For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

Class

A Class is a user-defined data-type which has data members and member functions.

Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behavior of the objects in a Class.

The general syntax for defining a class is:



```
class classIdentifier
{
    classMembersList
};
```

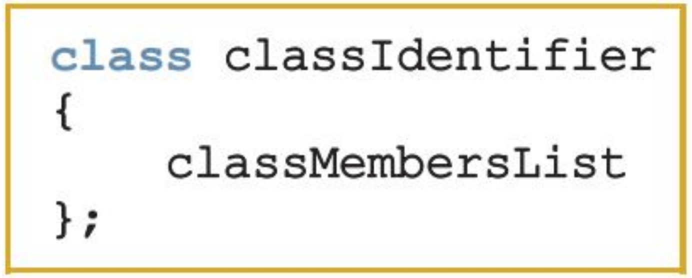
in which classMembersList consists of variable declarations and/or functions. That is, a member of a class can be either a variable (to store data) or a function.

Class

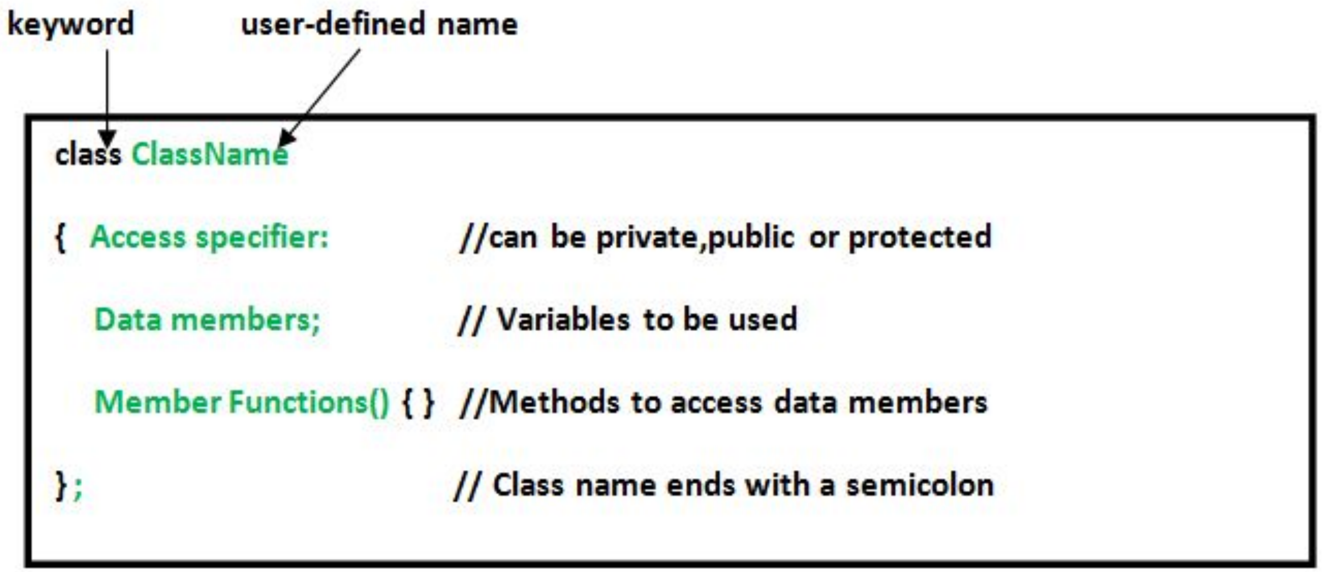
• If a member of a class is a variable, you declare it just like any other variable. Also, in the definition of the class, you cannot initialize a variable when you declare it.

• If a member of a class is a function, you typically use the function prototype to declare that member.

• If a member of a class is a function, it can (directly) access any member of the class—member variables and member functions. That is, when you write the definition of a member function, you can directly access any member variable of the class without passing it as a parameter. The only obvious condition is that you must declare an identifier before you can use it.



```
class classIdentifier
{
    classMembersList
};
```



```
class ClassName

{  Access specifier:          //can be private,public or protected

   Data members;              // Variables to be used

   Member Functions() { }     //Methods to access data members

};                            // Class name ends with a semicolon
```

In C++, class is a reserved word, and it defines only a data type; no memory is allocated.

It announces the declaration of a class.

Moreover, note the semicolon (;) after the right brace.

The semicolon is part of the syntax. A missing semicolon, therefore, will result in a syntax error

The members of a class are classified into three categories: **private, public,** and **protected**.

Access specifiers

Following are some facts about public and private members of a class:

- By default, all members of a class are **private.**
- If a member of a class is **private**, you cannot access it outside of the class.
- A public member is accessible outside of the class.
- To make a member of a class public, you use the member access specifier **public** with a colon, **:**

```
class person
{
    char name[20];
    int id;
public:
    void getdetails(){}
};
```

Object:

An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Variable (object) Declaration:

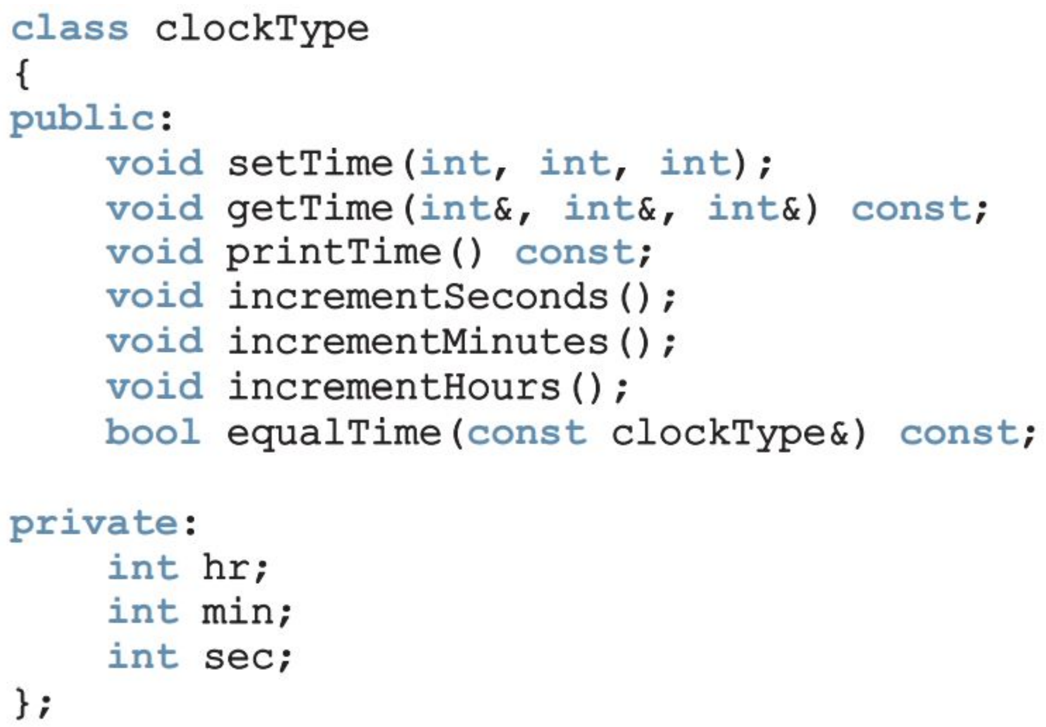
Once a class is defined, you can declare variables of that type. In C++ terminology, a class variable is called a class object or class instance

The syntax for declaring a class object is the same as that for declaring any other variable.

The following statements declare two objects of type clockType:

clockType myClock;

clockType yourClock;

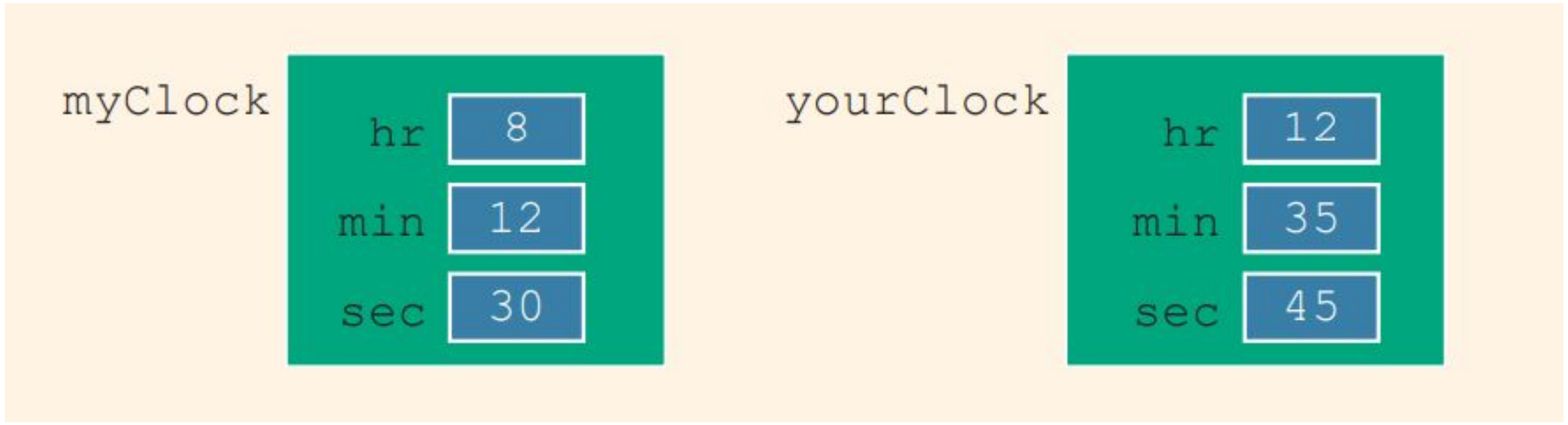


```
class clockType
{
public:
    void setTime(int, int, int);
    void getTime(int&, int&, int&) const;
    void printTime() const;
    void incrementSeconds();
    void incrementMinutes();
    void incrementHours();
    bool equalTime(const clockType&) const;

private:
    int hr;
    int min;
    int sec;
};
```

Object

In actuality, memory is allocated only for the member variables of each class object. The C++ compiler generates only one physical copy of a member function of a class, and each class object executes the same copy of the member function.



Accessing Class members

Once an object of a class is declared, it can access the members of the class. The general syntax for an object to access a member of a class is:

```
classObjectName.memberName
```

The class members that a class object can access depend on where the object is declared.

• If the object is declared in the definition of a member function of the class, then the object can access both the public and private members.

• If the object is declared elsewhere (for example, in a user’s program), then the object can access only the public members of the class.

Member Functions in Classes

There are 2 ways to define a member function:

Inside class definition

Outside class definition

To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

```
#include <iostream>
#include <string>
using namespace std;
class MyClass {       // The class
public:             // Access specifier
   int myNum;        // Attribute (int variable)
   string myString;  // Attribute (string variable)
   void func();
};
void MyClass::func()
{
   cout << "hello";
}
int main() {
   MyClass myObj;  // Create an object of MyClass
   myObj.myNum = 15;
   myObj.myString = "Some text";
   cout << myObj.myNum << "\n";
   cout << myObj.myString;
   return 0;
}
```

Builtin operations on classes

Most of C++’s built-in operations do not apply to classes. You cannot use arithmetic operators to perform arithmetic operations on class objects (unless they are overloaded; see Chapter 15). For example, you cannot use the operator + to add two class objects of, say, type clockType. Also, you cannot use relational operators to compare two class objects for equality (unless they are overloaded;

Accessor and Mutators

Accessor function: A member function of a class that only accesses (that is, does not

modify) the value(s) of the member variable(s).

Because an accessor function only accesses the values of the member variables, as a safeguard, we typically include the reserved word const at the end of the headings of these functions. Moreover, a constant member function of a class cannot modify the member variables of that class

Mutator function: A member function of a class that modifies the value(s) of the member

variable(s)

Utility Functions

Private function of a class

C++ Constructor

A constructor is a special type of member function that is called automatically when an object is created.

In C++, a constructor has the same name as that of the class and it does not have a return type. For example,

```
class  Wall {
  public:
    // create a constructor
    Wall() {
      // code
    }
};
```

1. has the same name as the class,
2. does not have a return type, and
3. is public
4. **Note:** If we have not defined a constructor in our class, then the C++ compiler will automatically create a default constructor with an empty code and no parameters.
5. **C++ Default Constructor**
   A constructor with no parameters is known as a default constructor. In the example above, Wall() is a default constructor.

C++ Constructor

A constructor, even though it is a function, has no type. That is, it is neither a value-returning function nor a void function.

• A class can have more than one constructor. However, all constructors of a class have the same name.

• If a class has more than one constructor, the constructors must have different formal parameter lists. That is, either they have a different number of formal parameters or, if the number of formal parameters is the same, then the data type of the formal parameters, in the order you list, must differ in at least one position.

• Constructors execute automatically when a class object enters its scope. Because they have no types, they cannot be called like other functions.

• Which constructor executes depends on the types of values passed to the class object when the class object is declared.

Parameterized constructors

In C++, a constructor with parameters is known as a parameterized constructor. This is the preferred method to initialize member data.

```
#include <iostream>
using namespace std;
// declare a class
class Wall {
  private:
    double length;
    double height;
  public:
    // parameterized constructor to initialize variables
    Wall(double len, double hgt) {
      length = len;
      height = hgt;
    }
    double calculateArea() {
      return length * height;
    }
};
int main() {
  // create object and initialize data members
  Wall wall1(10.5, 8.6);
  Wall wall2(8.5, 6.3);
  cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
  cout << "Area of Wall 2: " << wall2.calculateArea();
  return 0;
}
```

C++ Copy Constructor

The copy constructor in C++ is used to copy data of one object to another.

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

   Wall(double len, double hgt) {
      length = len;
      height = hgt;
   }
   Wall(Wall& obj) {
      length = obj.length;
      height = obj.height;
   }
   double calculateArea() {
      return length * height;
   }
};
int main() {
   Wall wall1(10.5, 8.6);
   Wall wall2 = wall1;
   cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
   cout << "Area of Wall 2: " << wall2.calculateArea();

   return 0;
}
```

```
#include <iostream>
using namespace std;
class Wall {
private:
   double length;
   double *height;
public:
   Wall(double len, double hgt) {
      length = len;
      height = new double;
      *height = hgt;
   }
  /* Wall(Wall& obj) {
      cout << "Copy constructor by me" << endl;
      length = obj.length;
      height = obj.height;
   }*/
   double calculateArea() {
      return length * *height;
   }
   void fun()
   {
      delete height;
   }
};
int main() {
   Wall wall1(10.5, 8.6);
   Wall wall2 = wall1;

   cout << "Area of Wall 1: " << wall2.calculateArea() << endl;
   wall2.fun();
   cout << "Area of Wall 2: " << wall1.calculateArea();

   return 0;
}
```

```
#include <iostream>
using namespace std;
class Wall {
private:
   double length;
   double *height;
public:

   Wall(double len, double hgt) {
      length = len;
      height = new double;
      *height = hgt;
   }
  Wall(Wall& obj) {
      cout << "Copy constructor by me" << endl;
      length = obj.length;
      height = new double;
      *height = *(obj.height);
   }
   double calculateArea() {
      return length * *height;
   }
   void fun()
   {
      delete height;
   }
};
int main() {
   Wall wall1(10.5, 8.6);
   Wall wall2 = wall1;

   cout << "Area of Wall 1: " << wall2.calculateArea() << endl;
   wall2.fun();
   cout << "Area of Wall 2: " << wall1.calculateArea();

   return 0;
}
```

DEFAULT PARAMETERS

A constructor can also have default parameters. In such cases, the rules for declaring formal parameters are the same as those for declaring default formal parameters in a function. Moreover, actual parameters to a constructor with default parameters are passed according to the rules for functions with default parameters.

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

  Wall(double len = 2, double hgt = 2) {
    length = len;
    height = hgt;
  }
  Wall(Wall& obj) {
    length = obj.length;
    height = obj.height;
  }
  double calculateArea() {
    return length * height;
  }
};
int main() {
  Wall wall1;
  Wall wall2I(wall1);
  cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
  cout << "Area of Wall 2: " << wall2I.calculateArea();

  return 0;
}
```

Destructors

Destructor is another special member function that is called by the compiler when the scope of the object ends.

Syntax of Destructor

```
~class_name()
{
  //Some code
}
```

```
class Geeks
{
   public:
   int id;

   //Definition for Destructor
   ~Geeks()
   {
      cout << "Destructor called for id: " << id <<endl;
   }
};

int main()
  {
   Geeks obj1;
   obj1.id=7;
   int i = 0;
   while ( i < 5 )
   {
      Geeks obj2;
      obj2.id=i;
      i++;
   } // Scope for obj2 ends here
   return 0;
  } // Scope for obj1 ends here
```

Destructor rules

1) Name should begin with tilde sign(~) and must match class name.
2) There cannot be more than one destructor in a class.
3) Unlike constructors that can have parameters, destructors do not allow any parameter.
4) They do not have any return type, just like constructors.
5) When you do not specify any destructor in a class, compiler generates a default destructor and inserts it into your code.

When does the destructor get called?

A destructor is **automatically called** when:
1) The program finished execution.
2) When a scope (the { } parenthesis) containing **local variable** ends.
3) When you call the delete operator.