AMBITION GURU

Object-Oriented programming in C++ (CSIT 202)

II semester, BScCSIT

Compiled by Ankit Bhattarai
Ambition Guru College

Compiled by ab

Syllabus

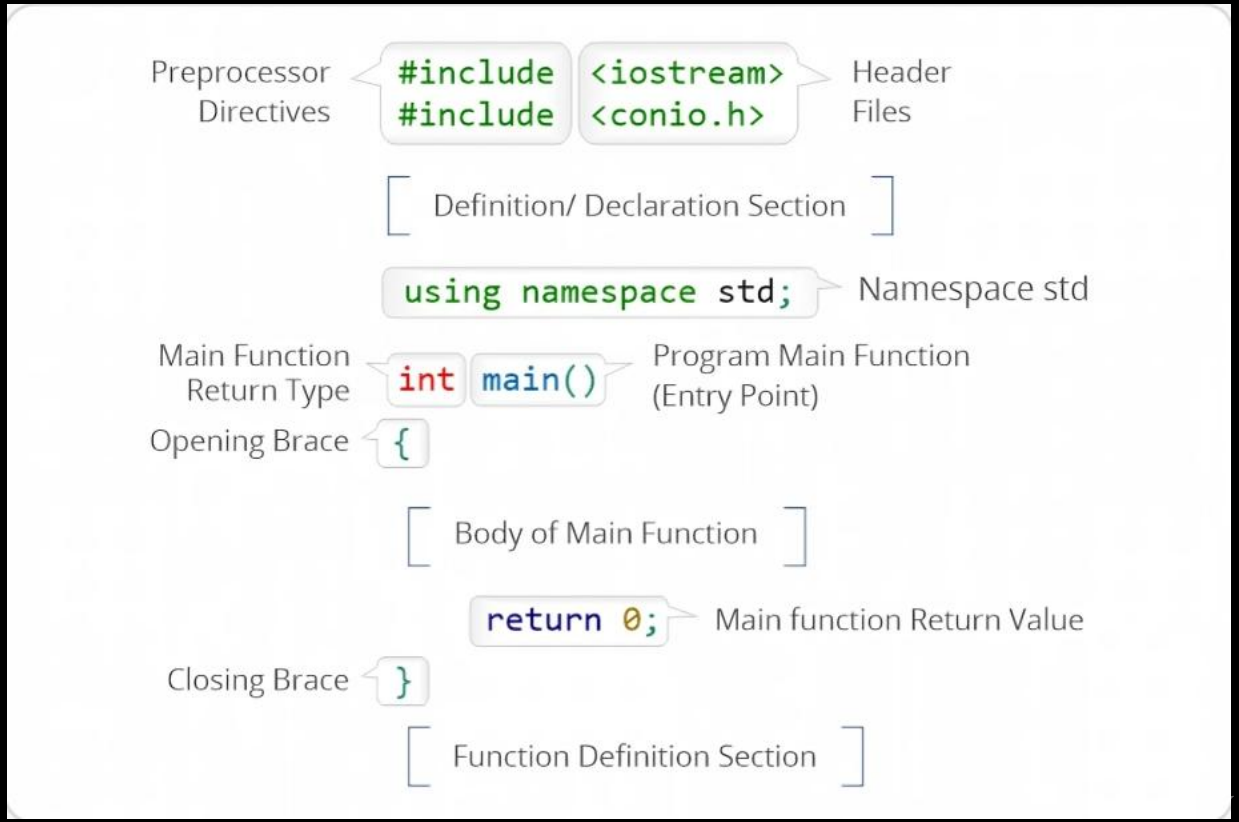
Unit	Contents	Hours	Remarks
1.	Introduction to C++ and OOP	3	
2.	**Classes and objects**	**7**	
3.	Operator Overloading & Type Conversion	7	
4.	Inheritance	7	
5.	Polymorphism and Virtual Functions	7	
6.	Templates and Exception Handling	7	
7.	I/O Stream	8	

Practical Works

Credit hours : 3

Format for C++ program

AMBITION GURU



Unit 2

Classes and Objects

(7 hrs.)

Defining classes and creating objects, Access specifiers: public, private, protected, Namespace, scope resolution operator, member functions (inside and outside class), inline function, storage classes: (automatic, external, static, register)., Static data members and functions, this pointer , Constructors and destructors, types of constructor (default, parameterized), Dynamic constructor, copy constructor, constructor overloading, manipulating private data members.

Defining classes and creating objects, Access specifiers: public, private, protected, Namespace, scope resolution operator, member functions (inside and outside class), inline function, storage classes: (automatic, external, static, register)., Static data members and functions.

C++ class

- C++ incorporates all these extensions in another user-defined type known as class.
- The only difference between a structure and a class in C++ is that,
  - By default, the members of a class are private.
  - By default, the members of a structure are public.

Specifying a Class

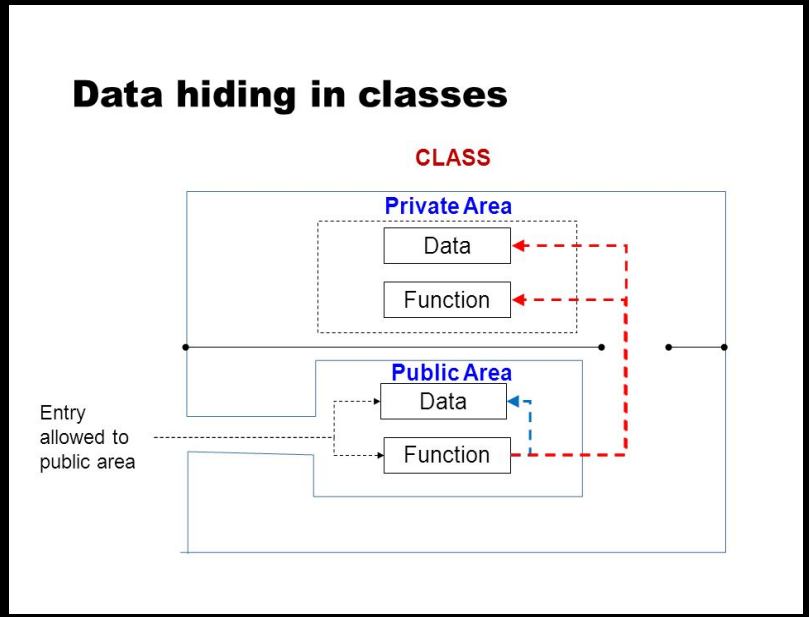
- A class is a way to bind the data and its associated functions together(encapsulation).
- Class is a combination of data members and member function collectively called as members of the class. Class is an example of encapsulation.
- It allows the data(and function) to be hidden, if necessary, from external use.
- A class specification has two parts:
  - Class declaration
  - Class function definitions

Class declaration

AMBITION GURU

Syntax:

```
class class_name{
    private:
        variable declarations;
        function declarations;
    public:
        variable declarations;
        function declarations;
};
```

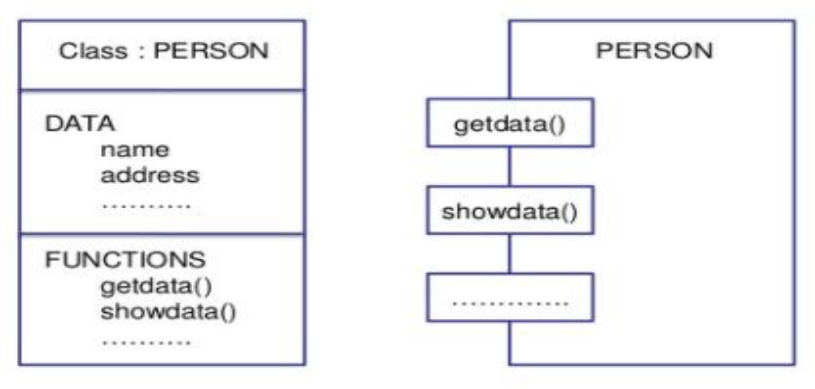


Class declaration

Example:

```
class person
{
        char name[20];     //Variables declaration
        char address[25]; //private by default
    public:
        void getdata();    //function declaration
        void showdata();
};
```

Representation of a class



Access Specifiers

Access Specifiers:

Keywords **public, private and protected** are know as access specifiers or sometimes they are called as visibility modes in C++.

These access specifiers define how the members of the class can be accessed.

- **public:** The members declared as public are accessible from outside the class through an object of the class.

- **private:** These members are accessible from within the class. No outside access is allowed. Keyword private is optional.

- **protected:** The members declared as protected are accessible from outside the class **BUT** only in a class derived from it.

(Note: protected and private access specifiers have the same effect in the class. However their difference is seen in the case of inheritance. That is to say protected information can be inherited but the private information cannot be inherited in C++.)

Data Members

- The variables which are declared in any class by using any fundamental data types (like int, char, float etc.) or derived data type (like class, structure, pointer etc.) are known as **Data Members**.
- There are two **types of data members/member functions in C++**:

1. Private members
2. Public members

Private members

- The members which are declared in private section of the class (using private access modifier) are known as private members.
- Private members can be accessible within the same class in which they are declared.

Public members

- The members which are declared in public section of the class (using public access modifier) are known as public members.
- Public members can access within the class and outside of the class by using the object name of the class in which they are declared.

Creating Objects

Once a class has been declared, we can *create variables(**objects**) of that type by using the class name*(like any other built-in type variable)

```
person x;  //memory for x is created

person x,y,z;

class person
{
   ......
   ......
}x,y,z;
```

Accessing Class Members

AMBITION GURU

Syntax:

```
object-name.function-name(actual-arguments);
```

Example:

```
x.getdata();
x.showdata();
```

```
getdata(); //Error
x.name;    //Error
```

```
class xyz
{
    int x;
    int y;
public:
    int z;
};

int main()
{
    xyz p;
    p.x=0;  //error, x is private
    p.z=10; //OK, z is public
}
```

Defining Member Functions

- A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.
- It operates on any object of the class of which it is a member, and has access to all the members of a class for that object
- Member functions are created and stored in memory only once when a class specification is declared
- All of the objects of that class have access to the same area in the memory where the member functions are stored.
- **Member functions can be defined in two places:**
  - **Outside the class definition**
  - **Inside the class definition**

Outside the class definition

AMBITION GURU

Syntax:

```
return-type class-name::function-name(argument declaration)
{
    //Function body
}
```

- The membership label (**class-name::**) tells the compiler that the function function-name belongs to the class class-name.
- That is, the **scope** of the function is **restricted to the class-name** specified in the header line.

Example:

```
void person::getdata()
{
    ......
    ......
}
```

```
void person::showdata()
{
    ......
    ......
}
```

Outside the class definition

Characteristics of Member Function:

- Several different classes can use the same function name. The '**membership label**' will resolve their scope.
- Member functions can access the **private** data of the class. A non-member function cannot do so.(However, an exception to this rule is a *friend function*)
- A member function can **call another member function** directly, without using the dot operator.

Inside the class definition

AMBITION GURU

- Replace the function declaration by the actual function definition inside the class.

```
class person
{
    char name[20];
    char address[25];
public:
    void getdata()   //inline function
    {
        ......
        ......
    }
    void showdata()   //inline function
    {
        ......
        ......
    }
};
```

NOTE:

- Function defined inside a class is treated as an **inline function**.
- All restrictions that apply to an inline function are also applicable here.
- Normally, only **small functions** are defined inside the class definition.

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

class item {
    int number;       // private by default
    float cost;       // private by default

public:
    void getdata(int a, float b);  // function declaration
    void putdata() {               // inline function definition
        cout << "number: " << number << endl;
        cout << "cost: " << cost << endl;
    }
};
```

In C++, endl is a stream manipulator used with cout to:

- Insert a newline character (\n)
- Flush the output buffer

```
// Member function definition outside the class
void item::getdata(int a, float b) {
    number = a;
    cost = b;
}

int main() {
    item i1;                        // Create object
    i1.getdata(101, 55.5);          // Set data
    i1.putdata();                   // Display data
    return 0;
}
```

Making an outside Function Inline

- One of the objective of OOP is to separate the details of implementation from the class definition.
- It is therefore good practice to define the member **functions outside the class**.

```
class item
{
     .....
     .....
public:
     void getdata(int a,float b); //declaration
};

inline void item::getdata(int a,float b) //definition
{
     number=a;
     cost=b;
}
```

C++ Class

- Create a C++ Class **Weight** having private data members: float **kg, grams**. A member function **getvalue ()** should enter their values from the user. Another member function **putvalue ()** should display their values. Define both functions **inside the class.** Member function defined inside the class behaves like an inline function.

Code Solution:

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

class Weight {
private:
    float kg;
    float grams;

public:
    void getvalue() {
        cout << "Enter weight in kilograms: ";
        cin >> kg;
        cout << "Enter weight in grams: ";
        cin >> grams;
    }

      void putvalue() {
        cout << "Weight = " << kg << " KILO AND " << grams << " GRAMS" << endl;
    }
};
```

Code Solution:

```
int main() {
    Weight w;

    w.getvalue();    // input weight
    w.putvalue();    // display weight

    return 0;
}
```

C++ Class

- Create a C++ class Student having private data members: string name, int rollNo, and float marks. A member function getData() should take input for these values from the user. Another member function putData() should display the values. Define both functions inside the class.

Example of class and multiple objects:

AMBITION GURU

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

class student
{
     char name[20];
     int roll_no;
     float marks[5];

     public:
     void input( );
     void display( )
     {
          cout<<"The name of the student="<<name<<endl;
          cout<<"The roll number of the student="<<roll_no<<endl;
          cout<<"The marks of the five subjects are:"<<endl;
          for(int i=0;i<5;i++)
          {
               cout<<marks[i]<<endl;
          }
     }
};
```

Example of class and multiple objects:

AMBITION GURU

```
void student :: input( )
{
     cout<<"Enter the name of the student:"<<endl;
     cin>>name;
     cout<<"Enter the roll of the student:"<<endl;
     cin>>roll_no;
     cout<<"Enter the marks obtained in five subjects:"<<endl;
     for(int i=0;i<5;i++)
     {
     cin>>marks[i];
     }
}

int main( )
{
     student s1,s2;
     s1.input( );
     s2.input( );
     s1.display( );
     s2.display( );
     return 0;
}
```

Static Data Members

Static Data Members

AMBITION GURU

- A data member of a class can be qualified as **static**.

Static data member has certain special characteristics:

- It is ***initialized to zero*** when the first object of its class is created. No other initialization is permitted.
- ***Only one copy*** of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its ***lifetime is the entire program*****.**

Static Data Members

AMBITION GURU

- A static variable is normally used to maintain value common to the entire class.
  - ***For e.g., to hold the count of objects created*.**
- Note that the type and scope of each static member variable **must be declared outside the class definition.**
- This is necessary because the static data members are stored separately rather than as a part of an object.
- While defining a static variable, some initial value can also be assigned

```
int item::count=10;
```

- Since they are associated with the class itself rather than with any class objects, they are also known as **class variables.**

Static Data Members

AMBITION GURU

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

class item
{
     int number;
     static int count;
public:
     void getdata(int a)
     {
          number =a;
          count++;
     }

     void getcount()
     {
      cout<<"count:";
      cout<<count<<"\n";
     }

};
```

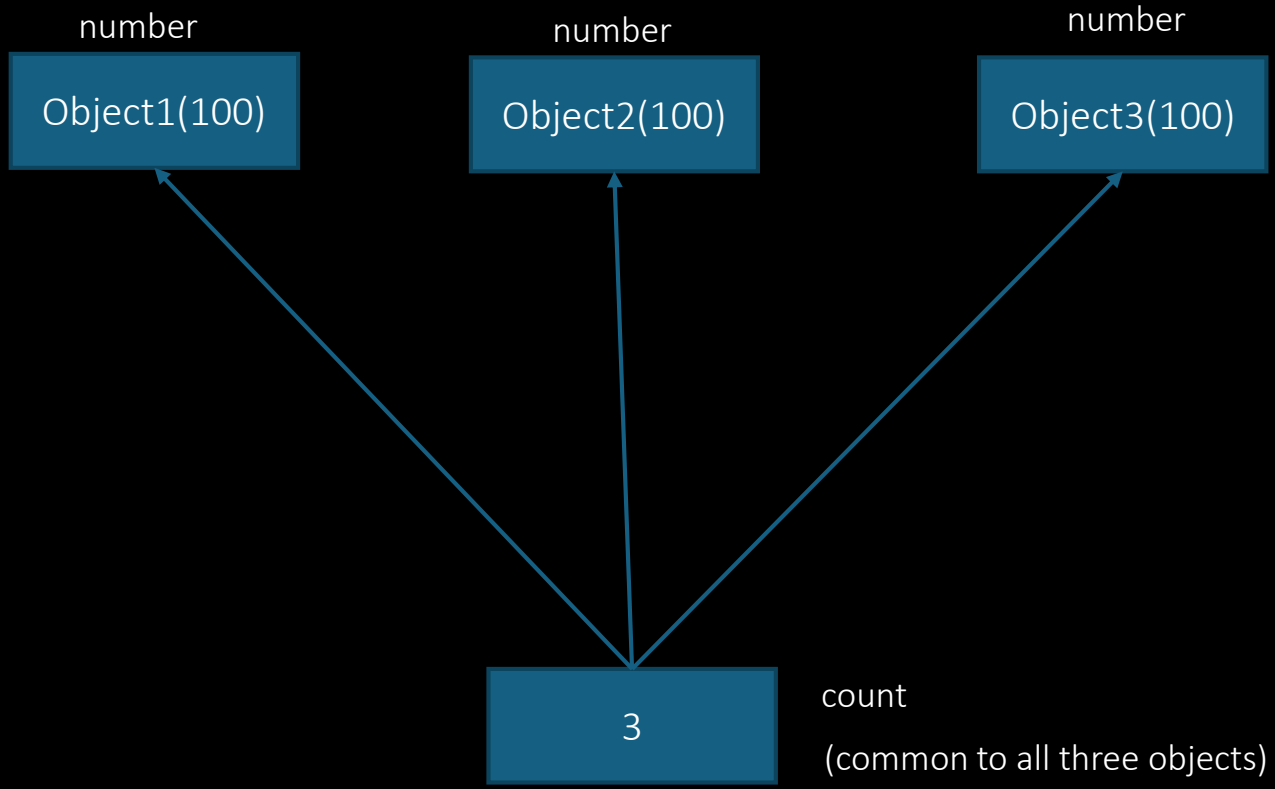
Static Data Members

AMBITION GURU

```
int item::count;

int main()
{
     item a,b,c;

     a.getcount();
     b.getcount();
     c.getcount();

     a.getdata(100);
     b.getdata(200);
     c.getdata(300);

     cout<<"After reading data"<<"\n";

     a.getcount();
     b.getcount();
     c.getcount();
     return 0;
}
```

```
count:0
count:0
count:0
After reading data
count:3
count:3
count:3
```

Static Data Members



Static Member Functions

- Like a static member variable, we can also have static member functions.

Static member function has certain special characteristics:

- A static function can have access to only **other static members (function or variable)** declared in the same class.
- A static member function can be called using the **class name (instead of its object)** as follows:

```
class_name:: function_name();
```

Static Member Functions


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

class test {
    int code;
    static int count;  // static data member

public:
    void setcode() {
        code = ++count;
    }

    void showcode() {
        cout << "object number: " << code << "\n";
    }

    static void showcount() {
        cout << "count: " << count << "\n";
        // cout << code; // ERROR: code is not static
    }
};
```

Static Member Functions

AMBITION GURU

```
// Definition of static member outside the class
int test::count;

int main() {
    test t1, t2;

    t1.setcode();  // count = 1
    t2.setcode();  // count = 2

    test::showcount();  // Output: count: 2

    test t3;
    t3.setcode();       // count = 3

    test::showcount();  // Output: count: 3

    t1.showcode();      // Output: object number: 1
    t2.showcode();      // Output: object number: 2
    t3.showcode();      // Output: object number: 3

    return 0;
}
```

Output

```
count: 2
count: 3
object number: 1
object number: 2
object number: 3
```

Objects as Function Arguments

AMBITION GURU

Like any other data type, an object may be used as a function argument.

- **This can be done in two ways:**

Pass/Call By Value

- A ***copy of the entire object*** is passed to the function
- Any changes made to object inside function **don't affect original object.**

Pass/Call By Reference

- Only the ***address of the object*** is transferred to the function.
- Any changes made to object inside function are **reflected in the original object.**

Objects as Function Arguments: Example

Part 1

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

class Number {
public:
    int value;

    Number(int v) {
        value = v;
    }

    void display() {
        cout << "Value: " << value << endl;
    }
};
```

Objects as Function Arguments: Example

Part 2

```
// Pass by Value
void modifyByValue(Number obj) {
    obj.value += 10;
    cout << "Inside modifyByValue(): ";
    obj.display();
}

// Pass by Reference
void modifyByReference(Number &obj) {
    obj.value += 20;
    cout << "Inside modifyByReference(): ";
    obj.display();
}
```

Objects as Function Arguments: Example

Part 3

```
int main() {
    Number n(100);

    cout << "Original Object: ";
    n.display();

    modifyByValue(n);       // Pass by value (copy)
    cout << "After modifyByValue(): ";
    n.display();

    modifyByReference(n);   // Pass by reference (original)
    cout << "After modifyByReference(): ";
    n.display();

    return 0;
}
```

Objects as Function Arguments: Example

Part 4

Output:

```
Original Object: Value: 100
Inside modifyByValue(): Value: 110
After modifyByValue(): Value: 100
Inside modifyByReference(): Value: 120
After modifyByReference(): Value: 120
```

WAP in C++ to add two time objects using a member function.

- Define a *class time* with data members: hr, min, and sec.
- Create three member functions:
  - *get()* to input time,
  - *disp()* to display time in [hr:min:sec] format,
  - *sum(time, time)* to add two time objects and store the result in the current object.
- Perform proper adjustment for minutes and seconds (i.e., 60 seconds = 1 minute, 60 minutes = 1 hour).

WAP in C++ to add two time objects using a member function.

Part 1

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

class time {
    int hr, min, sec;

public:
    void get() {
        cout << "Enter Hour :: ";
        cin >> hr;
        cout << "Enter Minutes :: ";
        cin >> min;
        cout << "Enter Seconds :: ";
        cin >> sec;
    }

    void disp() {
        cout << "[ " << hr << ":" << min << ":" << sec << " ]\n";
    }
```

WAP in C++ to add two time objects using a member function.

Part 2

```
void sum(time t1, time t2) {
        sec = t1.sec + t2.sec;
        min = sec / 60;
        sec = sec % 60;

        min = min + t1.min + t2.min;
        hr = min / 60;
        min = min % 60;

        hr = hr + t1.hr + t2.hr;
    }
};
```

WAP in C++ to add two time objects using a member function.

Part 3

```
int main() {
    time t1, t2, t3;
    cout << "Enter first time:\n";
    t1.get();

    cout << "\nEnter second time:\n";
    t2.get();

    t3.sum(t1, t2);

    cout << "\nFirst Time: ";
    t1.disp();
    cout << "Second Time: ";
    t2.disp();
    cout << "Sum of Time: ";
    t3.disp();

    return 0;
}
```

WAP in C++ to add two time objects using a member function.

Output

```
Enter first time:
Enter Hour :: 1
Enter Minutes :: 45
Enter Seconds :: 50

Enter second time:
Enter Hour :: 2
Enter Minutes :: 30
Enter Seconds :: 30

First Time: [ 1:45:50 ]
Second Time: [ 2:30:30 ]
Sum of Time: [ 4:16:20 ]
```

Objects as Function Arguments

AMBITION GURU

Write a C++ program having **class Dist** with private data member **int feet** and **float inches**. Define following public member functions for it.

1) **getdata( )** to take feet and inches as input

2) **putdata( )** to display distance in 1’2.5” format

3**) add( )** to do addition of two distances such that it can handle function call **d1.add(d2)** where d1, d2 and d3 are objects of class.

Use Concept of **Object as Function Arguments**.

Storage Classes in C++

AMBITION GURU

Storage Classes in C++

- Storage Classes are used to describe the features of a variable/function.
- These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.
- To specify the storage class for a variable, the following syntax is to be followed:

Syntax:

```
storage_class var_data_type var_name;
```

C++ uses 5 storage classes, namely:

1. auto
2. register
3. extern
4. static
5. mutable

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extern	Whole program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage
Mutable	Mutable	Class	Local	garbage

Auto:

- Unless specified otherwise, a variable declared in a function is of type *auto* (an *automatic variable*). **For example**: *int linecount;*
- Storage is allocated to the variables each time the function is called and is released when the function returns.
- There is no connection between the value left by a previous call and the initial value of the next call.
- When a value is not given explicitly to the variable, we can assume that it contains garbage.
- The scope of an automatic variable is the function in which it is declared. Another function may use a variable of the same name without conflict.

Example:


```
#include <iostream>
using namespace std;
void autoStorageClass()
{
    cout << "Demonstrating auto class\n";
    // Declaring an auto variable
    // No data-type declaration needed
    auto a = 32;
    auto b = 3.2;
    auto c = "KICEM";
    auto d = 'K';
```

```
// printing the auto variables
    cout << a << " \n";
    cout << b << " \n";
    cout << c << " \n";
    cout << d << " \n";
}

int main()
{
    // To demonstrate auto Storage Class
    autoStorageClass();
    return 0;
}
```

Output:

```
Demonstrating auto class
32
3.2
KICEM
K
```

extern:

- Extern storage *class simply tells us that the variable is defined elsewhere and not within the same block where it is used.*
- Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere.
- It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block.
- This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only.
- The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program

Example:

AMBITION GURU

```
#include <iostream>
using namespace std;
/*declaring the variable which is to be made extern an initial value can also be
initialized to x*/

int x;
void externStorageClass()
{
    cout << "Demonstrating extern class\n";
    /* telling the compiler that the variable x is an extern variable and has been
       defined elsewhere (above the main function)*/

    extern int x;
    // printing the extern variables 'x'

    cout << "Value of the variable 'x'"<< "declared, as extern: " << x << "\n";
    // value of extern variable x modified
    x = 2;

   // printing the modified values of extern variables 'x'
   cout<< "Modified value of the variable 'x'"<< " declared as extern: \n"<< x;
}
```

```
int main()
{
    // To demonstrate extern Storage Class
    externStorageClass();
    return 0;
}
```

Output

```
Demonstrating extern class
Value of the variable 'x'declared, as extern: 0
Modified value of the variable 'x' declared as extern:
2
```

static:

- This storage class is used to declare static variables which are popularly used while writing programs in C++ language.
- Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope.
- So we can say that they are initialized only once and exist until the termination of the program.
- Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program.
- By default, they are assigned the value 0 by the compiler.

AMBITION GURU

Example:

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

//Function containing static variables memory is retained during execution
int staticFun(){
    cout << "For static variables: ";
    static int count = 0;
    count++;
    return count;
}

// Function containing non-static variables memory is destroyed
int nonStaticFun(){
    cout << "For Non-Static variables: ";
     int count = 0;
    count++;
    return count;
}
```

```
int main()
{
    // Calling the static parts
    cout << staticFun() << "\n";
    cout << staticFun() << "\n";

    // Calling the non-static parts
    cout << nonStaticFun() << "\n";
    cout << nonStaticFun() << "\n";
    return 0;
}
```

Output:

```
For static variables: 1
For static variables: 2
For Non-Static variables: 1
For Non-Static variables: 1
```

register:

- This storage class declares register variables which have the same functionality as that of the auto variables. The *only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available*.
- This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program.
- If a free register is not available, these are then stored in the memory only.
- Usually, a few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program.
- An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

```
#include <iostream>
using namespace std;
void registerStorageClass(){
    cout << "Demonstrating register class\n";
    // declaring a register variable
    register char b = 'K';
    // printing the register variable 'b'
    cout << "Value of the variable b declared as register: " << b;
}
int main(){
    // To demonstrate register Storage Class
    registerStorageClass();
    return 0;
}
```

Output:

```
Demonstrating register class
Value of the variable b declared as register: K
```

AMBITION GURU

mutable:

- Sometimes there is a requirement to modify one or more data members of class/struct through const function even though we don’t want the function to update other members of class/struct.
- This task can be easily performed by using the mutable keyword.
- The keyword mutable is mainly used to allow a particular data member of const object to be modified.
- When we declare a function as const, this pointer passed to function becomes const. Adding mutable to a variable allows a const pointer to change members.

Example:

```
#include <iostream>
using namespace std;
class Test {
public:
    int x;
    mutable int y;
    Test()
    {
        x = 4;
        y = 10;
    }
};
```

```
int main()
{
    const Test t1;
    // trying to change the value
    t1.y = 20;
    cout << t1.y;
    // Uncommenting below lines
    // will throw error
    // t1.x = 8;
    // cout << t1.x;
    return 0;
}
```

Output:

20

Constructors and destructors, types of constructor (default, parameterized), Dynamic constructor, copy constructor, constructor overloading, manipulating private data members

Initialization of Class Object

Constructor

- A constructor is a special member function of a class *whose task is to initialize the data members of the class.*
- It is called special member function of a class because *its name is same as that of the class name.*

Syntax:

```
class class_name
{
    //private data_members;
    public:
    class_name(argument(s) or no argument)    //constructor
    {
        //body of the constructor;
    }
};
```

Characteristics of a constructor:

- Constructor should be declared or defined in the public section of the class.
- It is invoked automatically whenever the object of its associated class is created.
- Constructor do not have return type not even void.
- Constructor do not get inherited.
- Like other C++ functions they can have default argument(s).

Need of constructors in C++:

- Constructors initialize object variables to avoid garbage values.
- They are automatically called when an object is created.
- They prevent errors from forgetting to manually initialize variables.
- They make code cleaner and easier to maintain.
- They ensure safe object use by all programmers.
- Constructor overloading allows flexible object creation.
- They reduce the need for extra initialization functions.
- They support advanced OOP features like inheritance and object copying.

Types of constructor:

1. Default Constructor
2. Parameterized constructor
3. Copy constructor

Types of constructor:

Default Constructor

- A constructor that takes no argument is called a default constructor. The task of the default constructor is to initialize the data members of a class i.e. the different objects of the class will always be initialized with same value.
- A default constructor is automatically called when no arguments are supplied while creating objects as:

```
class_name object_name;
```

Write a C++ program to define a class triangle with three sides as data members. *Use a default constructor to initialize the sides with some default values*. Include member functions to display the side lengths and to calculate and display the perimeter of the triangle. Create two objects of the class and show the output for both.

```
#include<iostream>
using namespace std;
class triangle
{
     float side1,side2,side3;
     public:
     triangle( ) //default constructor
     {
       side1=6.5;
       side2=7.5;
       side3=8.5;
     }

     void display( )
     {
       cout<<"Side1="<<side1<<endl;
       cout<<"Side2="<<side2<<endl;
       cout<<"Side3="<<side3<<endl;
     }

     void perimeter( )
     {
      float p=side1+side2+side3;
      cout<<"\nThe perimeter of the triangle="<<p;
   }
};
```

```
int main( )
{
   triangle t1,t2;
   //default constructor is invoked for t1 and t2
   t1.display( );
   t2.display( );
   t1.perimeter( );
   t2.perimeter( );
   return 0;
}
```

Output:

Side1=6.5
Side2=7.5
Side3=8.5
Side1=6.5
Side2=7.5
Side3=8.5

The perimeter of the triangle=22.5
The perimeter of the triangle=22.5

Parameterized constructor:

- It is seen that the default constructor always initializes the objects with the same value every time they are created.
- Sometimes it is necessary to *create objects with different initial values*.
- So we can make constructor that takes arguments and initializes the data members from the values in the argument list.
- The constructors that takes parameter(s) are called parameterized constructor.
- To invoke parameterized constructor the argument(s) or parameter(s) should be passed in parenthesis during object declaration.

```
class_name object_name(parameter(s));
```

OR

```
class_name object_name;    /*default constructor is called and object_name is initialized with particular values*/

object_name=class_name(parameter(s)); /*parameterized constructor is called and first nameless object is initialized with the parameter(s) and then it is assigned to object_name*/
```

OR

```
class_name object_name=class_name(parameters(s));
```

(Note: It is better to have default constructor when we have parameterized constructor in a class because if we create just a object of a class without passing any arguments in the parenthesis in main function then in such a case compiler searches for default constructor so it should be explicitly defined by the programmer i.e. if we called parameterized constructor in the second way as mentioned above)

Example:

AMBITION GURU

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

class triangle
{
float side1,side2,side3;

public:
triangle( )
{
  side1=0.0;
  side2=0.0;
  side3=0.0;
}

triangle(float a, float b, float c)
//parameterized constructor
{
    side1=a;
    side2=b;
    side3=c;
}
```

```
void display( )
{
 cout<<"Side1="<<side1<<endl;
 cout<<"Side2="<<side2<<endl;
 cout<<"Side3="<<side3<<endl;
}

void perimeter( )
{
 float p=side1+side2+side3;
 cout<<"\nThe perimeter of thetriangle="<<p;
}
};
```

```
int main( )
{
triangle t1(5.5,6.5,7.5); //parameterized constructor is invoked and t1 is initialized
triangle t2(3.5,4.5,5.5); //parameterized constructor is invoked and t2 is initialized
t1.display( );
t2.display( );
t1.perimeter( );
t2.perimeter( );
// OR
//triangle t1; default constructor is called and t1 is initialized
// t1=triangle(5.5,6.5,7.5); parameterized constructor is called and first nameless object is
initialized //and then it is assigned to t1.
//triangle t2; default constructor is called and t2 is initialized.
//t2=triangle(3.5,4.5,5.5); parameterized constructor is called and first nameless object is
initialized //and then it is assigned to t2.
//t1.display( );
//t2.display( );
//t1.permeter( );
//t2.perimeter( );
return 0;
}
```

Output:

```
Side1=5.5
Side2=6.5
Side3=7.5
Side1=3.5
Side2=4.5
Side3=5.5

The perimeter of the triangle=19.5
The perimeter of the triangle=13.5
```

Copy constructor:

- A constructor *that takes reference object as an argument of the same class* is called copy constructor.
- The task of copy constructor *is to initialize the object by copying the value of the object of its own type from the argument.*
- We must use a reference to the argument to the copy constructor because when an argument is passed by value, a copy of it is constructed and it calls itself over and over until the compiler runs out of memory.
- So in the copy constructor, the argument must be passed by reference, which creates no copies.

Example:


```
#include<iostream>
using namespace std;
class triangle
{
float side1,side2,side3;
public:
triangle( )
{
 side1=0.0;
 side2=0.0;
 side3=0.0;
}

triangle( float a, float b, float c)
{
 side1=a;
 side2=b;
 side3=c;
}

triangle(triangle &t ) //copy constructor
{
side1=t.side1;
side2=t.side2;
side3=t.side3;
}
```

```
void display( )
{
  cout<<"Side1="<<side1<<endl;
  cout<<"Side2="<<side2<<endl;
  cout<<"Side3="<<side3<<endl;
}

void perimeter( )
{
  float p=side1+side2+side3;
  cout<<"\nThe perimeter of the triangle="<<p;
}
};
```

```
int main( )
{
triangle t1(5.5,6.5,7.5); //Parameterized constructor is called
triangle t2(t1); //copy constructor is called and content of t1 is copied to t2;
//or
//triangle t2=t1; copy constructor is called and content of t1 is copied to t2;

triangle t3; //default constructor is called
t3=t1; // no copy constructor is called just content of t1 is copied to t3;
t1.display( );
t2.display( );
t3.display( );
t1.perimeter( );
t2.perimeter( );
t3.perimeter( );
return 0;
}
```

Output:

```
Side1=5.5
Side2=6.5
Side3=7.5
Side1=5.5
Side2=6.5
Side3=7.5
Side1=5.5
Side2=6.5
Side3=7.5

The perimeter of the triangle=19.5
The perimeter of the triangle=19.5
The perimeter of the triangle=19.5
```

Destructors:

- A destructor, as name implies, is used to destroy the objects that have been created by the constructor.
- Like a constructor, *destructor is a function appeared in public section of a class preceded by the tilde(~) sign.*
- The destructor never takes any argument nor has a return type not even void.
- The constructor is always called to reserve the memory for the instantiated object however the role of the destructor is to remove the object from the memory created by the constructor.
- Cannot be declared as const, volatile, or static. However, they can be invoked for the destruction of objects declared as const, volatile, or static.
- The objects are destroyed in reverse order of creation by the destructor.

Syntax:

```
class class_name
{
    //private data members;
    public:
    //other public member functions;
    ~class_name( )//destructor
    {
       //Body of destructor.
    }
};
```

```
#include <iostream>
using namespace std;
int count=0;

class Department
{
  int DepartmentId;
  char DepartmentName[30];

  public:
  Department(char DN[30], int DI)
  {
     count ++;
     cout<<"No. of object created="<<count<<endl;
    for(int i=0;i<30;i++)
           DepartmentName[i]=DN[i];
           DepartmentId=DI;
  }

void display( )
{
   cout<<"Department Name="<<DepartmentName<<endl;
   cout<<"Department Id="<<DepartmentId<<endl;
}
```

```
~Department( )//destructor
{
 cout<<"Number of object destroyed:“ <<count<<endl;
 count --;
}
};

int main( )
{
{
   Department D1("Electronics",3);
   Department D2("Computer",5);
   Department D3("Civil",7);
   D1.display( );
   D2.display( );
   D3.display( );
}
   cout<<"End of main";
   return 0;
}
```

Output:


```
No. of object created=1
No. of object created=2
No. of object created=3
Department Name=Electronics
Department Id=3
Department Name=Computer
Department Id=5
Department Name=Civil
Department Id=7
Number of object destroyed:3
Number of object destroyed:2
Number of object destroyed:1
End of main
```

Overloading Constructors

- A constructor can also be overloaded with more than one function that have the same name but different types or number of parameters.
- Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called.
- When an object is created, the one executed is the one that matches the arguments passed on the object declaration.

Overloading Constructors

Part 1

```
#include <iostream>
using namespace std;
class CRectangle
{
 int width, height;
 public:
 CRectangle ();
 CRectangle (int,int);
 int area()
 {
   return(width*height);
 }
};
```

Overloading Constructors


```
CRectangle::CRectangle()
{
 width = 5;
 height = 5;
}
```

Part 2

```
CRectangle::CRectangle(int a, int b)
{
 width = a;
 height = b;
}
```

Overloading Constructors

Part 3

```
int main ()
{
  CRectangle rect(3,4);
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Output:

```
rect area: 12
rectb area: 25
```

Questions

AMBITION GURU

1. WAP in C++ to create a class Student with data members name, rollno, faculty and age. Use a member function to input and display the data.
2. WAP in C++ to create a class Rectangle with data members length and breadth. Use member functions to input the values and display the area.
3. Create two classesRupee and Dollar respectively. Write conversion operator to convert between Rupee and Dollar assuming that 1 dollar equals 133 rupees. Write a main program that allows the user to enter an amount in either currency and then converts it to other currency and displays the result.
4. What is constructor? Why constructor is needed in a class? Illustrate the types of constructor with an example.
5. What is destructor? Write a program to show the destructor call such that it prints the message “memory is released”.

Questions

AMBITION GURU

6. What is the benefit of passing object as arguments? Write a program to create a class named actor with data members name and rating. Initialize the data members and display those names whose rating is greater than 5 using the concept of constant object.

7. Define class and object with suitable example. How members of class can be accessed?

8. Create two classes Celsius and Fahrenheit to represent temperatures in Celsius and Fahrenheit respectively. Write conversion functions to convert from Celsius to Fahrenheit and vice versa. Assume the formulas: F = (C × 9/5) + 32C = (F − 32) × 5/9. Write a main program that allows the user to input temperature in one unit and converts it to the other.

9. Create two classes Kilometer and Mile to represent distances. Write conversion functions to convert between kilometers and miles, assuming:1 mile = 1.60934 kilometers. Allow the user to enter a distance in either kilometers or miles and convert it to the other using appropriate class conversion. Display the result.

THANK YOU

Any Queries ?