
# OBJECT PROGRAMMING

- LECTURE 3 -

(1ST YEAR OF STUDY)



Skopje, 2023/24

# Contents



## 3. Friend Functions and Constructors

# 3.1. Friend Class and Function in C++

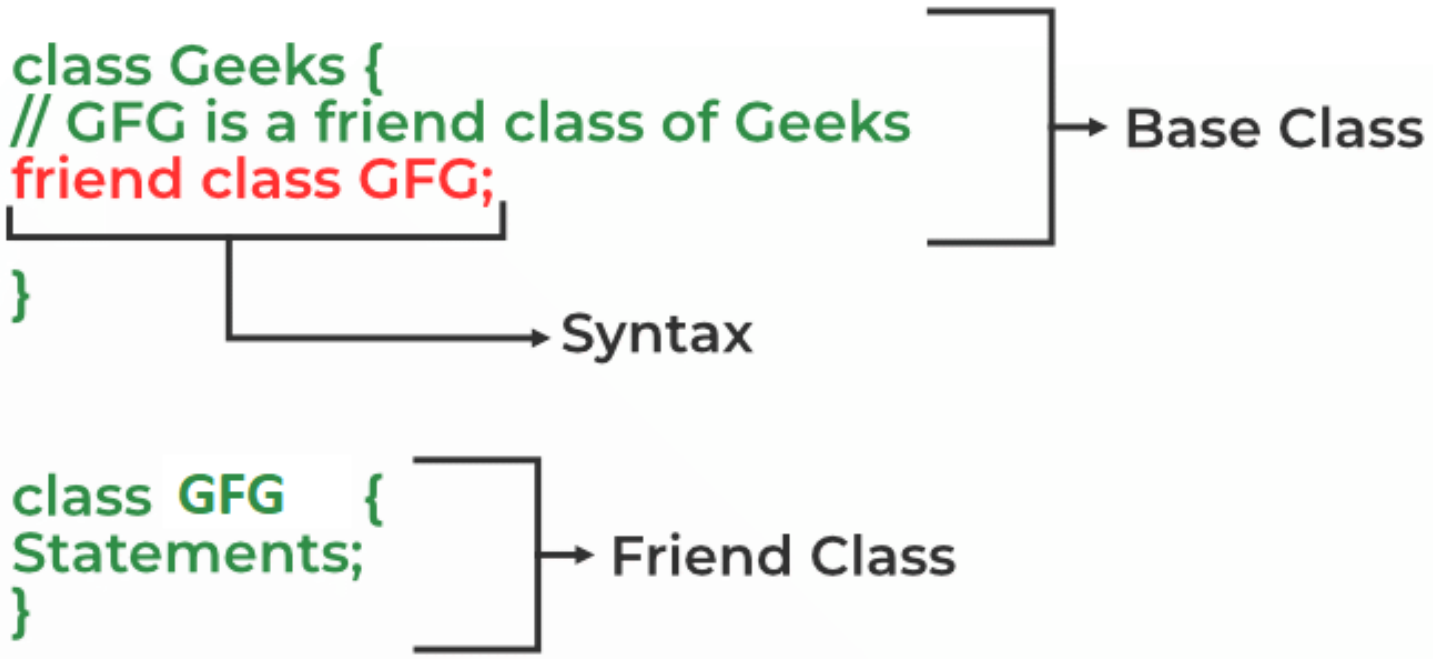


- A **friend class** can access *private* and *protected* members of other classes in which it is declared as a friend.
- It is sometimes useful to allow a particular class to access private and protected members of other classes. For example, a Linked List class may be allowed to access private members of Node.
- We can declare a friend class in C++ by using the **friend** keyword.

# 3.1. Friend Class and Function in C++



- **Syntax:**
  - friend class class_name; // declared in the base class



# 3.1. Friend Class and Function in C++



```cpp
// C++ Program to demonstrate the
// functioning of a friend class
#include <iostream>
using namespace std;

class GFG {
private:
    int private_variable;

protected:
    int protected_variable;

public:
    GFG()
    {
        private_variable = 10;
        protected_variable = 99;
    }

    // friend class declaration
    friend class F;
};

// Here, class F is declared as a
// friend inside class GFG. Therefore,
// F is a friend of class GFG. Class F
// can access the private members of
// class GFG.
class F {
public:
    void display(GFG& t)
    {
        cout << "The value of Private Variable = "
             << t.private_variable << endl;
        cout << "The value of Protected Variable = "
             << t.protected_variable;
    }
};

// Driver code
int main()
{
    GFG g;
    F fri;
    fri.display(g);
    return 0;
}
```

**Output**

```
The value of Private Variable = 10
The value of Protected Variable = 99
```

# 3.1. Friend Class and Function in C++

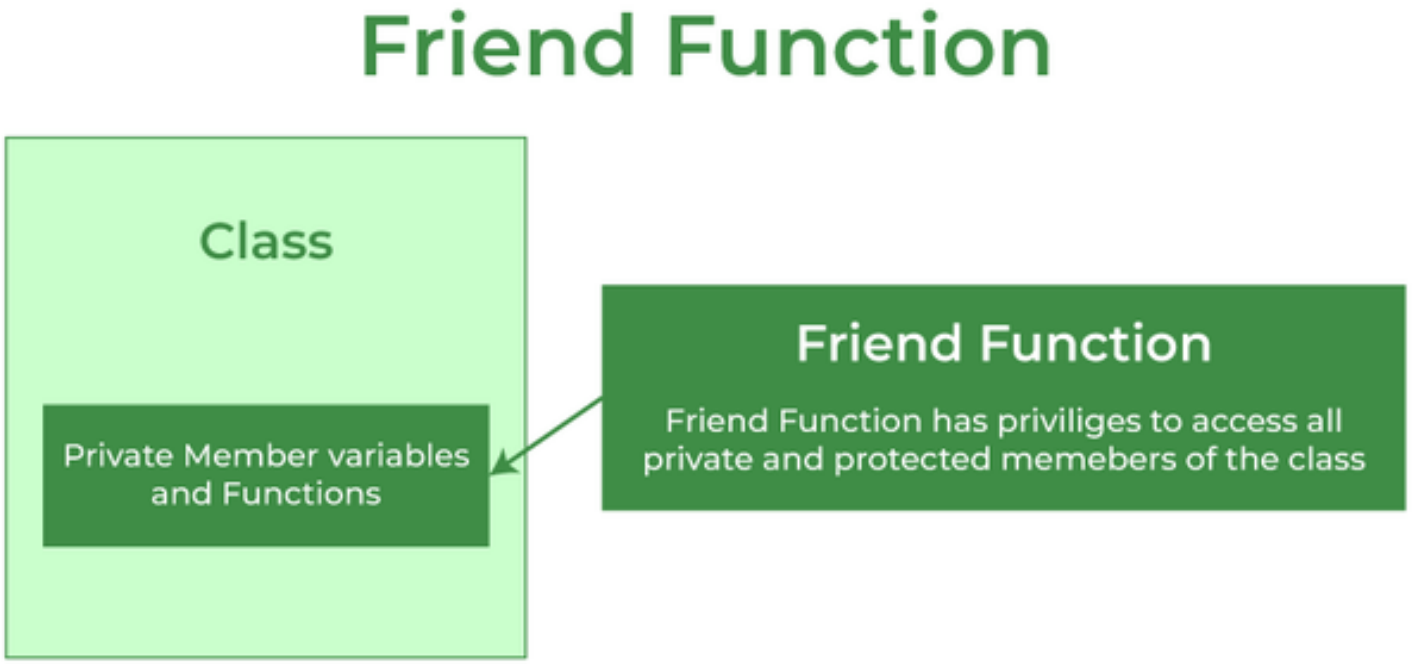


- **Note:** We can declare friend class or function anywhere in the base class body whether its private, protected or public block. It works all the same.
- Like a *friend class*, a **friend function** can be granted special access to *private* and *protected* members of a class in C++.
- They are the *non-member functions* that can access and manipulate the private and protected members of the class for they are declared as friends.

# 3.1. Friend Class and Function in C++



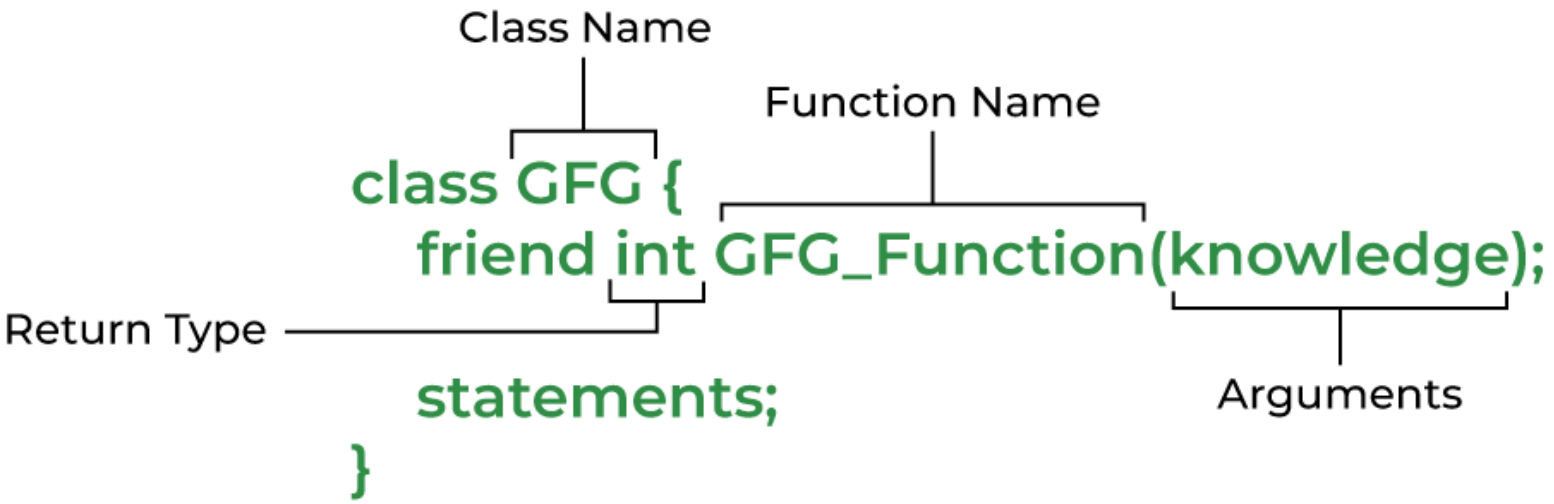
- A friend function can be:
  - **A global function**
  - **A member function of another class**



# 3.1. Friend Class and Function in C++



- **Syntax:**
  - friend return_type function_name (arguments);
    // for a global function, or
  - friend return_type class_name :: function_name (arguments); // for a member function of another class



# 3.1. Friend Class and Function in C++



## 1. Global Function as Friend Function

```
// C++ program to create a global function as a friend
// function of some class
#include <iostream>
using namespace std;

class base {
private:
    int private_variable;

protected:
    int protected_variable;

public:
    base()
    {
        private_variable = 10;
        protected_variable = 99;
    }

    // friend function declaration
    friend void friendFunction(base& obj);
};

// friend function definition
void friendFunction(base& obj)
{
    cout << "Private Variable: " << obj.private_variable
         << endl;
    cout << "Protected Variable: " << obj.protected_variable;
}

// driver code
int main()
{
    base object1;
    friendFunction(object1);

    return 0;
}
```

**Output**

```
Private Variable: 10
Protected Variable: 99
```

# 3.1. Friend Class and Function in C++



## 2. Member Function of Another Class as Friend Function

```
// C++ program to create a member function of another class
// as a friend function
#include <iostream>
using namespace std;

class base; // forward definition needed
// another class in which function is declared
class anotherClass {
public:
    void memberFunction(base& obj);
};

// base class for which friend is declared
class base {
private:
    int private_variable;

protected:
    int protected_variable;

public:
    base()
    {
        private_variable = 10;
        protected_variable = 99;
    }
    // friend function declaration
    friend void anotherClass::memberFunction(base&);
};

// friend function definition
void anotherClass::memberFunction(base& obj)
{
    cout << "Private Variable: " << obj.private_variable
         << endl;
    cout << "Protected Variable: " << obj.protected_variable;
}

// driver code
int main()
{
    base object1;
    anotherClass object2;
    object2.memberFunction(object1);

    return 0;
}
```

**Output**

```
Private Variable: 10
Protected Variable: 99
```

# 3.1. Friend Class and Function in C++



- **Note:** The order in which we define the friend function of another class is important and should be taken care of. We always have to define both the classes before the function definition. That's why we have used out of class member function definition.
- A friend function is a non-member function, or ordinary function, of a class, which is declared as a friend using the keyword "**friend**" inside the class. By declaring a function as a friend, all the access permissions are given to the function.
- Keyword "*friend*" is placed only in the function declaration of the friend function, and **not** in the **function definition** or **call.**

# 3.1. Friend Class and Function in C++



- **A Function Friendly to Multiple Classes**

```
// C++ Program to demonstrate
// how friend functions work as
// a bridge between the classes
#include <iostream>
using namespace std;

// Forward declaration
class ABC;

class XYZ {
    int x;

public:
    void set_data(int a)
    {
      x = a;
    }

    friend void max(XYZ, ABC);
};

class ABC {
    int y;

public:
    void set_data(int a)
    {
      y = a;
    }

    friend void max(XYZ, ABC);
};

void max(XYZ t1, ABC t2)
{
    if (t1.x > t2.y)
        cout << t1.x;
    else
        cout << t2.y;
}

// Driver code
int main()
{
    ABC _abc;
    XYZ _xyz;
    _xyz.set_data(20);
    _abc.set_data(35);

    // calling friend function
    max(_xyz, _abc);
    return 0;
}
```

**Output**

```
35
```

# 3.1. Friend Class and Function in C++



- **Advantages of Friend Functions**
  - A friend function is able to access members without the need of inheriting the class.
  - The friend function acts as a bridge between two classes by accessing their private data.
  - It can be used to increase the versatility of overloaded operators.
  - It can be declared either in the public or private or protected part of the class.

# 3.1. Friend Class and Function in C++



- **Disadvantages of Friend Functions**
  - Friend functions have access to private members of a class from outside the class which violates the law of data hiding.
  - Friend functions cannot do any run-time polymorphism in their members.

# 3.2. Constructors and Destructors



- **Constructor in C++** is a special method that is invoked automatically at the time of object creation.
  - Constructor is a *member function* of a class, whose name is ***same*** as the class name.
  - Constructor is a special type of member function that is used to *initialize* the data members for an object of a class automatically, when an object of the same class is created.
  - Constructor constructs the values, i.e. provides data for the object, that is why it is known as constructor.
  - Constructor ***do not return value***, hence they ***do not have a return type***.

# 3.2. Constructors and Destructors



- The prototype of Constructors is as follows:

  <class-name> (list-of-parameters);

- Constructors can be defined inside or outside the class declaration:
  - The syntax for defining the constructor within the class:

    <class-name> (list-of-parameters)

    { // constructor definition }
  - The syntax for defining the constructor outside the class:

    <class-name> : : <class-name> (list-of-parameters)

    { // constructor definition }

# 3.2. Constructors and Destructors



```
// defining the constructor within the class

#include <iostream>
using namespace std;

class student {
    int rno;
    char name[10];
    double fee;

public:
    student()
    {
        cout << "Enter the RollNo:";
        cin >> rno;
        cout << "Enter the Name:";
        cin >> name;
        cout << "Enter the Fee:";
        cin >> fee;
    }

    void display()
    {
        cout << endl << rno << "\t" << name << "\t" << fee;
    }
};

int main()
{
    student s; // constructor gets called automatically when
               // we create the object of the class
    s.display();

    return 0;
}
```

**Output:**

```
Enter the RollNo: 30
Enter the Name: ram
Enter the Fee: 20000
30 ram 20000
```

# 3.2. Constructors and Destructors



```
// defining the constructor outside the class

#include <iostream>
using namespace std;
class student {
    int rno;
    char name[50];
    double fee;

public:
    student();
    void display();
};

student::student()
{
    cout << "Enter the RollNo:";
    cin >> rno;

    cout << "Enter the Name:";
    cin >> name;

    cout << "Enter the Fee:";
    cin >> fee;
}

void student::display()
{
    cout << endl << rno << "\t" << name << "\t" << fee;
}

int main()
{
    student s;
    s.display();

    return 0;
}
```

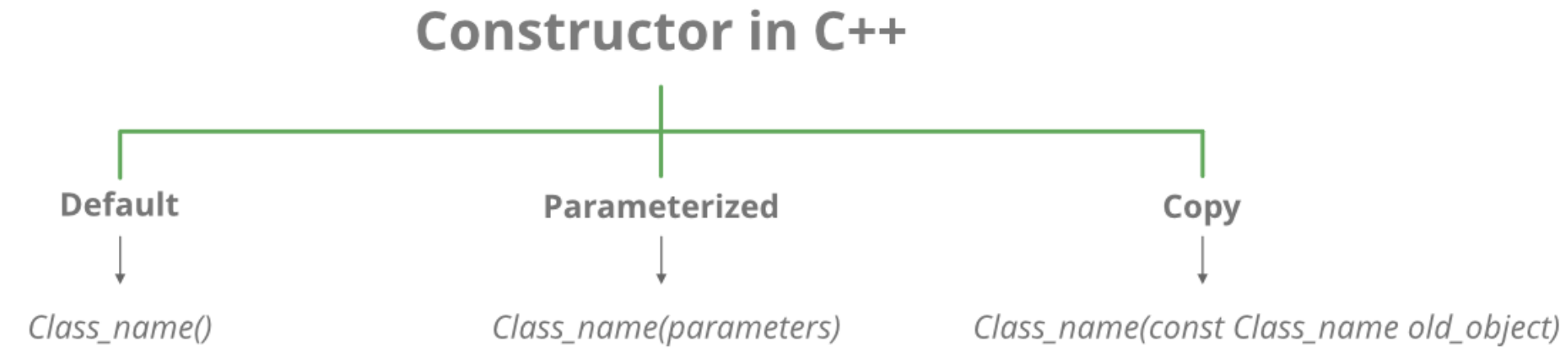
**Output:**

```
Enter the RollNo: 30
Enter the Name: ram
Enter the Fee: 20000
30 ram 20000
```

# 3.2. Constructors and Destructors



- Constructors are mostly declared in the *public* section of the class, though it can be declared in the *private* section of the class.
- Constructors can be overloaded. Constructors cannot be declared virtual. Constructors cannot be inherited.
- Types of constructors:



# 3.2. Constructors and Destructors



- A constructor is different from normal functions in following ways:
  - Constructor has same name as the class itself.
  - *Default Constructors* don't have input argument however, *Copy* and *Parameterized* Constructors have input arguments.
  - Constructors don't have return type.
  - A constructor is automatically called when an object is created.
  - If we do not specify a constructor, C++ compiler generates a ***default constructor*** for object (expects no parameters, and has an empty body).

# 3.2. Constructors and Destructors



```
// Cpp program to illustrate the
// concept of Constructors
#include <iostream>
using namespace std;

class construct {
public:
    int a, b;

    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl << "b: " << c.b;
    return 1;
}
```

- **Default constructor** is the constructor which does not take any argument. It has no parameters. It is also called a zero-argument constructor.

Output

```
a: 10
b: 20
```

# 3.2. Constructors and Destructors



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

class GFG
{
public:
      GFG()
      {
         cout << "Hi from GFG. ";
      }
} g;

int main()
{
    cout << "You are in Main";
    return 0;
}
```

- Here, g is a object of class GFG, for which constructor is fired first.
- Then statement of cout of main() is printed.

**Output:**

```
Hi from GFG. You are in Main
```

# 3.2. Constructors and Destructors



- **Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. When you define the constructor's body, use the parameters to initialize the object.
- **Note:** when the parameterized constructor is defined and no default constructor is defined ***explicitly***, the compiler will not ***implicitly*** call the default constructor and hence creating a simple object as:

```
Student s;
```

will flash an error.

# 3.2. Constructors and Destructors



```cpp
// CPP program to illustrate
// parameterized constructors
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;

public:
    // Parameterized Constructor
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    int getX() { return x; }
    int getY() { return y; }
};

int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX()
         << ", p1.y = " << p1.getY();

    return 0;
}
```

**Output**

```
p1.x = 10, p1.y = 15
```

# 3.2. Constructors and Destructors



- When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The constructors can be called explicitly or implicitly:

```
Example e = Example(0, 50); // Explicit call
Example e(0, 50); // Implicit call
```

- **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.
- **Can we have more than one constructor in a class?**
  - Yes, It is called ***Constructor Overloading.***

# 3.2. Constructors and Destructors



```
#include<iostream.h>
class constructor
{
    int x;
    public:
    constructor(short ss)
    {
        cout<< "Short" << endl;
    }
    constructor(int xx)
    {
        cout<< "Int" << endl;
    }
    constructor(float ff)
    {
        cout<< "Float" << endl;
    }
};
int main()
{
    constructor c('B');
    return 0;
}
```

- Description: As 'B' gives the integer value, i.e., 66, so the parameterized constructor with integer parameter will be executed.

Output:

```
Int
```

# 3.2. Constructors and Destructors



- A **copy constructor** is a member function that initializes an object using another object of the same class.
- 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.
- Copy constructor takes a reference to an object of the same class as an argument.

# 3.2. Constructors and Destructors



```
// Implicit copy constructor

#include<iostream>
using namespace std;

class Sample
{          int id;
    public:
    void init(int x)
    {
        id=x;
    }
    void display()
    {
        cout<<endl<<"ID="<<id;
    }
};

int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    Sample obj2(obj1); //or obj2=obj1;
    obj2.display();
    return 0;
}
```

```
// Example: Explicit copy constructor

#include <iostream>
using namespace std;

class Sample
{
    int id;
    public:
    void init(int x)
    {
        id=x;
    }
    Sample(){}  //default constructor with empty body

    Sample(Sample &t)   //copy constructor
    {
        id=t.id;
    }
    void display()
    {
        cout<<endl<<"ID="<<id;
    }
};
int main()
{
    Sample obj1;
    obj1.init(10);
    obj1.display();

    Sample obj2(obj1); //or obj2=obj1;    copy constructor called
    obj2.display();
    return 0;
}
```

**Output**

```
ID=10
ID=10
```

# 3.2. Constructors and Destructors



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

class constructor
{
    int a;
  public:
    constructor(int x)
    {
        a = x;
    }

    void display()
    {
        cout<< "a: "<<a << endl;
    }
};
int main()
{
   constructor c1(10);
   constructor c2 = c1;
   c1.display();
   c2.display();
   return 1;
}
```

- Description: This program demonstrates the copy constructor concept.

**Output:**

```
a: 10
a: 10
```

# 3.2. Constructors and Destructors



- A **destructor** is also a special member function as a constructor. Destructor destroys the class objects created by the 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. Hence destructor *cannot be overloaded*.
- Destructor neither requires any argument, nor returns any value. It is automatically called when the object goes out of scope. Destructors release memory space occupied by the objects created by the constructor. In destructor, objects are destroyed in the *reverse* of object creation.

# 3.2. Constructors and Destructors



- The syntax for defining the destructor within the class

  ~ <class-name>() { }

- The syntax for defining the destructor outside the class

  <class-name> : : ~ <class-name>() { }

# 3.2. Constructors and Destructors



- **Characteristics of a destructor:**
  - Destructor is invoked automatically by the compiler when its corresponding constructor goes out of scope and releases the memory space that is no longer required by the program.
  - Destructor neither requires any argument nor returns any value; therefore it cannot be overloaded.
  - Destructor  cannot be declared as static and const;
  - Destructor should be declared in the public section of the program.
  - Destructor is called in the reverse order of its constructor invocation.

# 3.2. Constructors and Destructors



```
#include <iostream>
using namespace std;
int count = 0;
class Test {
public:
    Test()
    {
        count++;
        cout << "\n No. of Object created:\t" << count;
    }

    ~Test()
    {
        cout << "\n No. of Object destroyed:\t" << count;
        --count;
    }
};

main()
{
    Test t, t1, t2, t3;
    return 0;
}
```

- **Output**

No. of Object created: 1

No. of Object created: 2

No. of Object created: 3

No. of Object created: 4

No. of Object destroyed: 4

No. of Object destroyed: 3

No. of Object destroyed: 2

No. of Object destroyed: 1

# 3.3. Set and Get Methods



- A **client of an object** - that is, any class or function that calls the object's member functions from *outside* the object - calls the class's public member functions to request the class's services for particular objects of the class.
- Classes often provide *public* member functions to allow clients of the class to ***set*** (i.e., assign values to) or ***get*** (i.e., obtain the values of) *private* data members.

# 3.3. Set and Get Methods



- These member function names need not begin with *set* or *get*, but this naming convention is common.
- *Set functions* are also sometimes called ***mutators*** (because they mutate, or change, values).
- *Get functions* are also sometimes called ***accessors*** (because they access values).
- Using public *get* and *set* member functions enables control of access to the data members.
- *Set* and *get* member functions can validate attempts to modify private data and control how that data is presented to the caller.