

# Introduction to Object Oriented Programming

Course Code: CSE 205

Course Title: Object Oriented Programming

---

**NAFIZ IMTIAZ KHAN**

Lecturer, Department of CSE, MIST

Email: [nafiz@cse.mist.ac.bd](mailto:nafiz@cse.mist.ac.bd)

## Objective

1. To achieve a basic idea on **Object Oriented Programming** Language.
2. To Present object-oriented aspects of C++.
3. To learn programming with C++

## Course Outcome

1. Grasp and utilize the **fundamental features** of an object oriented language like C++ / Java.
2. Explain the benefits of **object oriented design** and understand when it is an appropriate methodology to use.
3. **Deduce** object oriented solutions for small problems, involving multiple objects.
4. Illustrate good programming style and identify the impact of style on developing and maintaining programs.

# Course Outline

Week	Topics	Remarks
1	Overview of Object Oriented Programming(OOP) and introduction to C++; Features of OOP, namespaces	
2	Introduction to class and objects, Access Specifiers	
3	Member Functions, In-line functions, Friend functions, Function Overloading	
4	Introduction to the concept of Constructors and Destructors	
5	Copy Constructors	
6	Using arrays of objects and references of objects, Using objects as arguments and returning objects from functions	
7	?	TBD

A word cloud featuring various C++ programming concepts. The words are arranged in a dense, overlapping manner against a black background. The colors of the words include shades of yellow, orange, green, and red. The most prominent words, shown in larger fonts, are 'Inheritance' and 'Overloading'. Other visible words include 'Pointers', 'Dynamic', 'Object', 'Class', 'Multiple', 'C++', 'delete', 'malloc', 'Operator', 'new', 'Constructors', 'Cout', 'Polymorphism', 'Virtual', 'Destructors', 'Binding', 'Cin', 'endl', 'functions', and 'free'.

Pointers  
Dynamic  
Object  
Class  
Multiple  
C++  
delete  
malloc  
Operator  
new  
Constructors  
Inheritance  
Cout  
Polymorphism  
Virtual  
Destructors  
Binding  
Cin  
endl  
functions  
Overloading  
free

# History

- ❑ The C++ programming language has a history going back to 1979, when **Bjarne Stroustrup** was doing work for his Ph.D. thesis.
- ❑ He began to work on "C with Classes", which as the name implies was meant to be a superset of the C language. C++ programming language was evolved from “C with Classes” in 1983 at bell laboratories of AT&T (American Telephone & Telegraph), located in U.S.A.
- ❑ It was develop for adding a feature of OOP (Object Oriented Programming) in C without significantly changing the C component.

**Reference:** <http://www.cplusplus.com/info/history/>



big  
think

# What is C++?

- ❑ C++ is a high-level Object Oriented Programming language with sufficient lower level functionality (in-line assembly code, access to memory locations, etc)
- ❑ C++ is (almost exactly) a superset of C
- ❑ At a high level:  $C++ \approx (C + OOP)$



# C Programming Language

---

- ❑ Procedural Language
- ❑ Good for general purpose programming
- ❑ Supports standard Data Types, Operators and Control Flow
- ❑ Little problematic to handle large programs.



# Procedural Programming: C

---

## Procedural Programs:

- ❑ Split a program into tasks and subtasks
- ❑ Write functions to carry out the tasks
- ❑ Instruct computer to execute the functions in a sequence

## Problems

- ❑ Large amount of data and intricate flow of tasks: complex and un-maintainable
- ❑ More importance to algorithms & functions than data being used by the functions
- ❑ Multi-function programs - shared data items are placed as global
- ❑ Changes in data structure affects functions and involves re-writing code

# Object Oriented Programming: C++

- ❑ It is C and some more.
- ❑ Concise
- ❑ Maintainable (Cause its Object Oriented)
- ❑ Portable (supports nearly any processor)
- ❑ OOP treats data as a critical element in the program development.
- ❑ Does not allow data to flow freely around the system.
- ❑ Package data and code as objects
- ❑ Programs model the problem using abstract objects and interactions between them.  
Eg : Complex numbers  $(x + yi)$
- ❑ C++ is an Object Oriented Programming language but is not purely Object Oriented.

# Object Oriented Programming: Advantages

- ❑ It is possible to map objects in the problem domain to those objects in the program.
- ❑ **Data hiding** helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- ❑ Through **inheritance** we can eliminate the redundant code and extend the use of existing code.
- ❑ It is possible to have multiple instances of an objects to co-exist without any interference.
- ❑ Easy to partition the work in a project based on objects.
- ❑ New data and functions can be easily added whenever necessary.

# Why learn C++ over C?

---

- ❑ C++ supports OOP features.
- ❑ C++ has an additional bool datatype. C does not.
- ❑ In C++, local vars can be declared anywhere in a block of code prior to using them. In C, they have to be declared at the beginning of the block of code.

# Why learn C++ over C?

- ❑ There is Stronger Type Checking in C++
- ❑ All the OOP features in C++ like **Abstraction**, **Encapsulation**, **Inheritance** etc makes it more worthy and useful for programmers.
- ❑ C++ supports and allows user defined operators (i.e **Operator Overloading**) and **function overloading** is also supported in it.
- ❑ Exception Handling

# Why learn C++ over C?

---

- ❑ The concept of **Virtual functions** and also **Constructors** and **Destructors** for Objects.
- ❑ Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.
- ❑ If a func return type is mentioned it must return a value in C++. In C it is optional.

# Features of OOP

---

- ☐ Data Abstraction
- ☐ Encapsulation
- ☐ Inheritance
- ☐ Polymorphism
- ☐ Messaging

# Reference

---

## **Books**

1. Teach Yourself C++ - Herbert Schildt
2. Object Oriented Programming with C++ - E Balagurusamy
3. Programming with C++ - Schaum's Outline Series
4. Complete Reference C++ - Herbert Schildt

## **Online**

1. <https://www.cplusplus.com/>
2. <https://www.w3schools.com/cpp/>
3. <https://www.tutorialspoint.com/cplusplus/index.htm>



**Thanks**

# Concept & Features of OOP

Course Code: CSE 205

Course Title: Object Oriented Programming

---

**NAFIZ IMTIAZ KHAN**

Lecturer, Department of CSE, MIST

Email: [nafiz@cse.mist.ac.bd](mailto:nafiz@cse.mist.ac.bd)

# Structure in C

```
struct Player {
    int jersey_no;
    char name[50];
};

void display(Player p1) {
    printf("Player Jersey: %d\n",
p1.jersey_no);
    printf("Player Name: %s\n", p1.name);
}
```

```
int main() {
    Player p1;
    scanf("%d", &p1.jersey_no);
    scanf("%s", &p1.name);

    display(p1);
}
```

- ❑ In C, structures can only store informations.
- ❑ Struct variables has to be passed to functions.

# Structure in C++

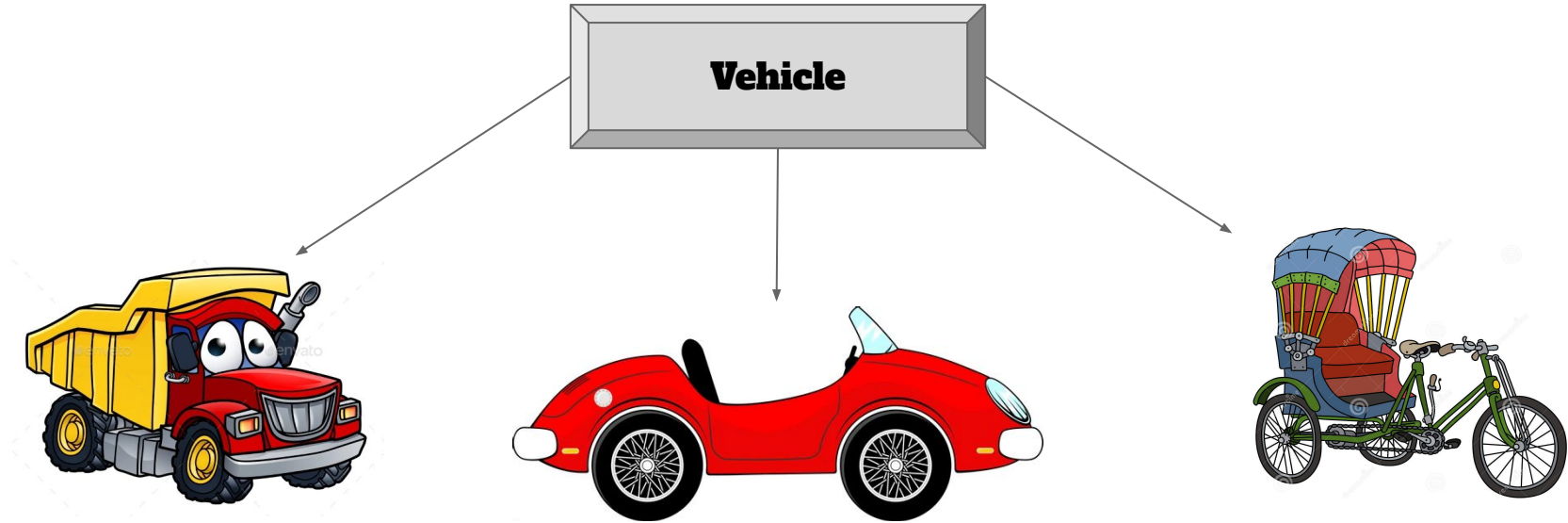
```
struct Student
{
    int id;
    int bengali, math, english;
    int getTotal() {
        return (bengali + math + english);
    }
};
```

- ❑ Struct can hold informations as well as methods
- ❑ So, struct variable can now have member functions along with variables.

```
int main(){
    Student s1;
    s1.id = 50;
    printf("%d\n", s1.getTotal());
}
```

- ❑ s1 is known as object.

# Concept: Class



- ❑ Vehicle is a thing used for transporting people, goods etc especially on land.
- ❑ Let's broadly categorize Vehicles into Trucks, Cars and Rickshaw.

# Concept: Class

- ❑ All vehicles have some common properties.
  - ❑ Trucks, Cars and Rickshaw - all have wheels.
  - ❑ All have seats to sit on.
- ❑ But some properties are unique.
  - ❑ Rickshaw don't have any engine.
  - ❑ Trucks have a space to carry loads.
  - ❑ Cars have air conditioning.
- ❑ All vehicles have some common functionalities.
  - ❑ They move from one place to other.
- ❑ But there are some specific functions which is valid to one and invalid for the others.
  - ❑ Rickshaw-puller pulls Rickshaw.

# Concept: Class

Here, we can take **Vehicle** as a **Class**.

A **Class** is a blueprint for any functional entity which defines its **properties** and **functions**. For example, Vehicles have different attributes and it can perform various actions.

## Another Example

class **HumanBeing**

### Body Parts:

Hands

Legs

Eyes etc

### Body Functions:

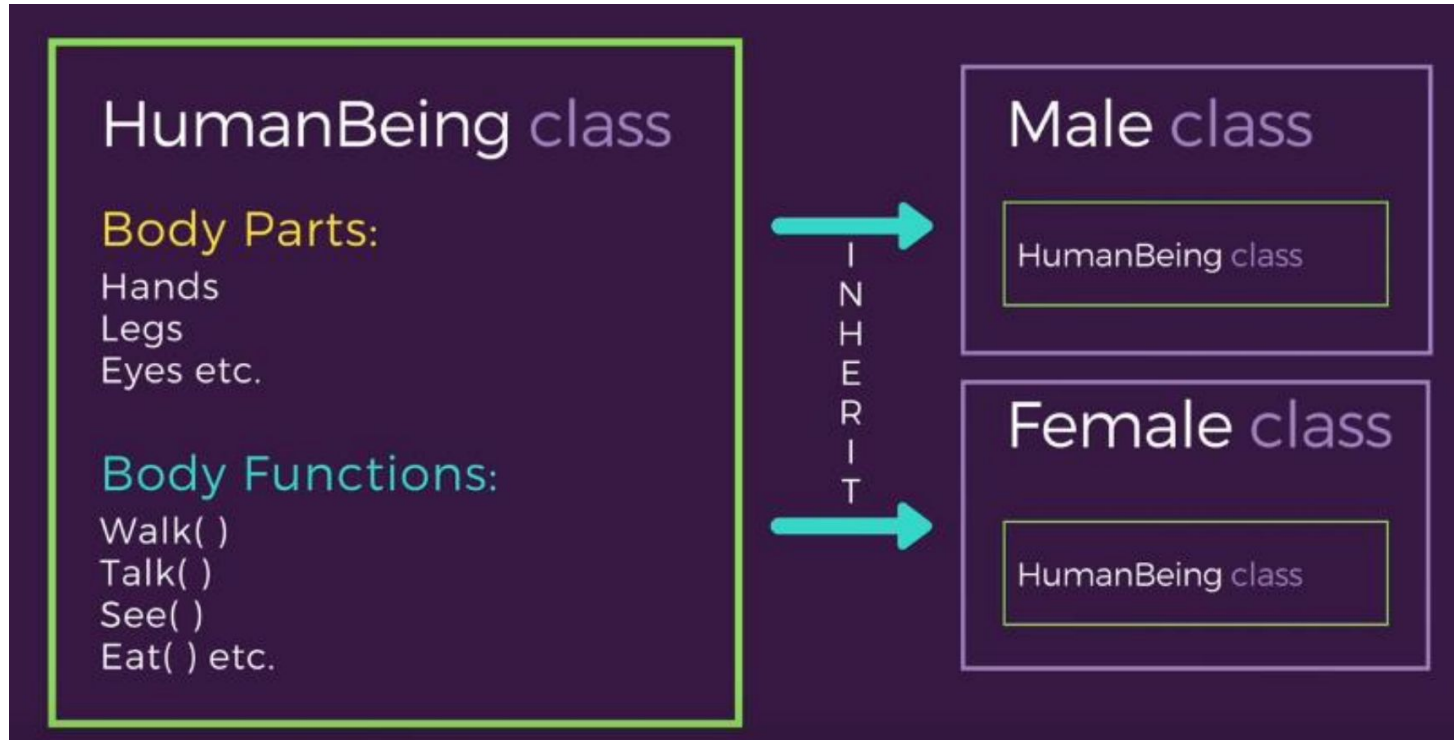
walk()

talk()

see()

eat() etc

# Concept: Object





# Concept: Object

---

- ❑ My name is **xyz**, and I am an instance/object of class **Male**.
- ❑ When we say, HumanBeing, Male or Female, we just mean a type/kind.
- ❑ You, your friend, me - we are the forms of these classes. We have a physical existence while a class is just a logical definition. We are the **objects/instances**.
- ❑ When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

# Data Abstraction

- ❑ Abstraction refers to the act of representing essential features without including the background details or explanations.
- ❑ Since the *classes* use the concept of data abstraction, they are known as *Abstract Data Types*.
- ❑ Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost. Functions operate on these attributes.
- ❑ Continuing our example, Human Being's can talk, walk, hear, eat, but the details are hidden from the outside world.
- ❑ We can take our skin as the Abstraction factor in our case, hiding the inside mechanism.

# Data Abstraction

HumanBeing class

name  
walk()  
talk()  
hear()

HIDDEN

some other class

OUTSIDE WORLD

I know HumanBeing  
has a name and it can  
walk, talk and hear. But  
how?

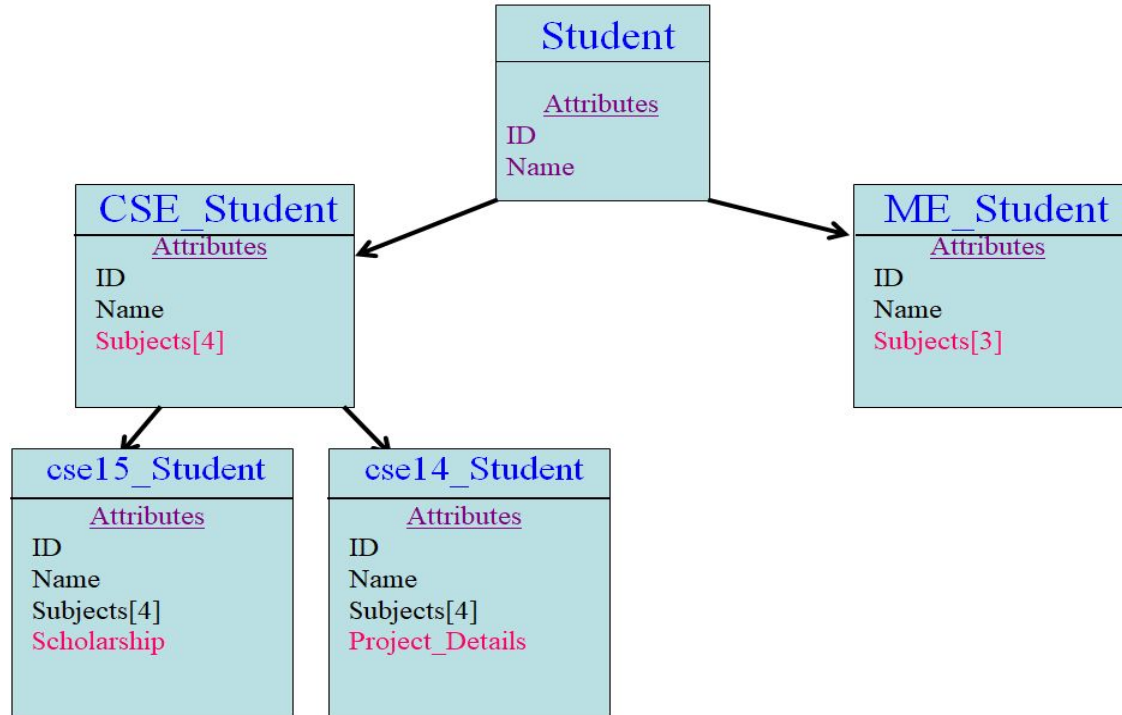
# Encapsulation

- ❑ It is the mechanism that binds together code and the data it manipulates, and keeps safe from outside interference and misuse. when code and data are bind together in this fashion, an **object** is created.
- ❑ The wrapping up of data and functions into a single unit.
- ❑ The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.
- ❑ This insulation of the data from direct access by the program is called **data hiding or information hiding**.

# Inheritance

- Consider the **HumanBeing** class, which has properties like hands, legs, eyes etc, and functions like walk, talk, eat, see etc.
- *Male* and *Female* are also classes, but most of the properties and functions are included in **HumanBeing**, hence they can inherit everything from class **HumanBeing** using the concept of Inheritance.
- Concept of inheritance provides the idea of reusability.
- Objects of one class acquire the properties of objects of another class.
- Each derived class shares common characteristics with the parent class.
- Possible to add additional features to an existing class without modifying it.

# Inheritance



# Polymorphism

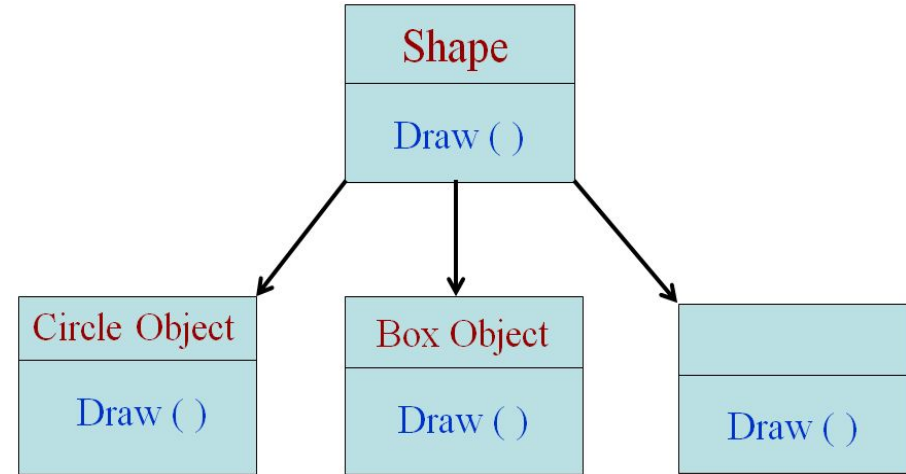
- Polymorphism means the ability to take more than one form.
- It is the quality that allows one name to be used for two or more related but technically different purposes. i.e.
  - **Function overloading:** Using a single function name to perform different types of tasks
  - **Operator overloading:** An operation may exhibit different behaviours depending upon the types of data.

Example (Addition operator):

- For two numbers the operation will generate a sum.
- For string operands, the operation would produce a third string by concatenation

# Polymorphism

- ❑ Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- ❑ Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time.
- ❑ It is associated with polymorphism and inheritance.



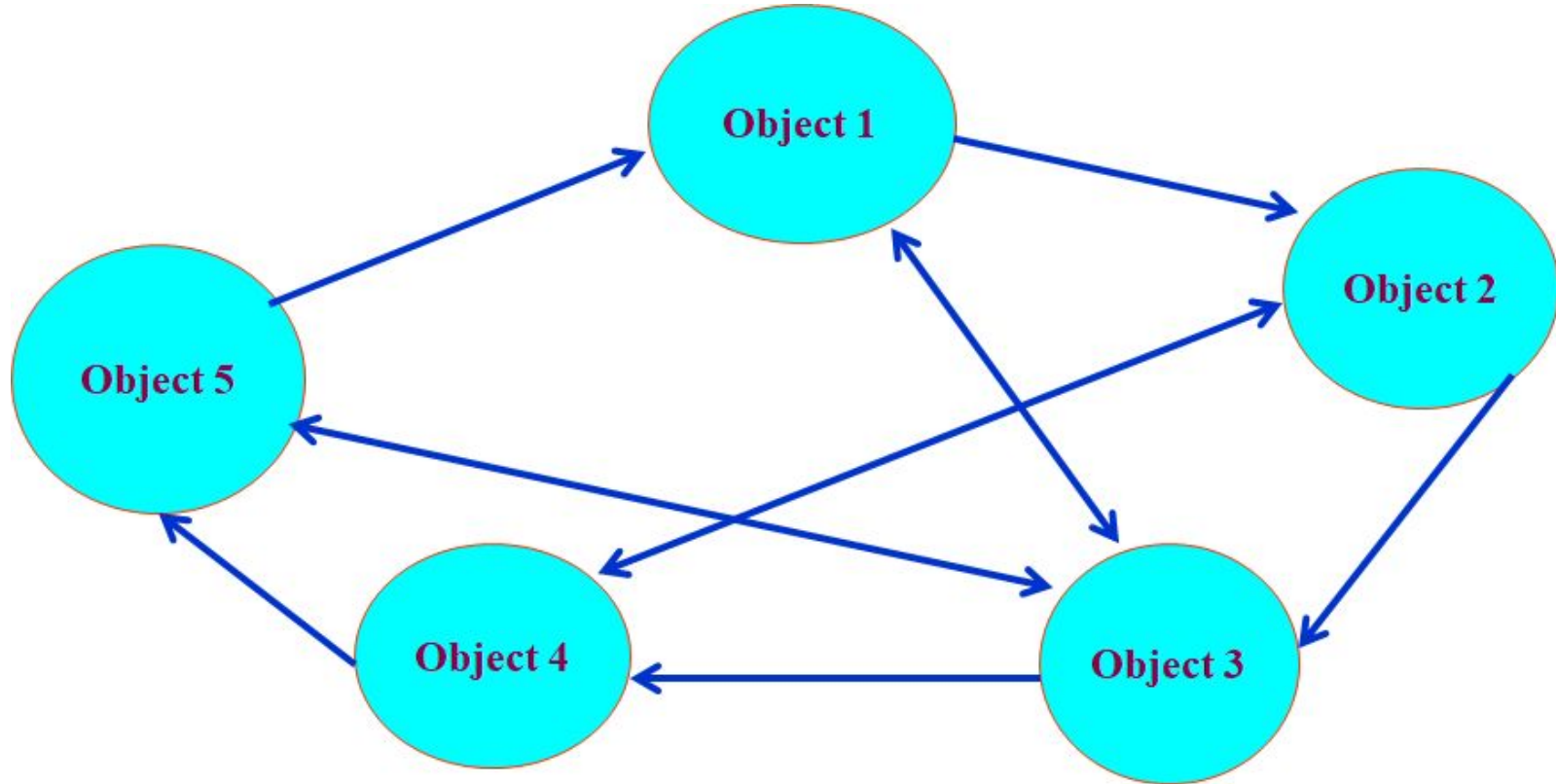


# Messaging

---

- ❑ Objects communicate with one another by sending and receiving information to each other.
- ❑ A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results.
- ❑ Message passing involves specifying the name of the object, the name of the function and the information to be sent.

# Messaging



# C++ Program Structure

- **.cpp** is the file extension.
- Line 1-3 : Preprocessor directives.
  - Lines beginning with # are preprocessor commands
  - Include the header file for functions like cin, cout, etc
  - Set the default namespace as standard
- Line 5: Entry point of the program. Begin execution here.
- Line 6-10: Body of the program

```
main.cpp
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int a;
7      cin >> a;
8      a++;
9      cout << a << endl;
10 }
```

# Input/Output

## C++

```
#include <iostream>
int main() {
    cout << "Hello World";
}
```

In C++, **stream insertion operator** (<<) is used for output

## C

```
#include <stdio.h>
int main() {
    printf("Hello World");
}
```

# Input/Output

## C++

```
#include <iostream>
int main() {
    int a;
    cin >> a;
}
```

In C++, **stream extraction operator** (>>) is used for output

## C

```
#include <stdio.h>
int main() {
    int a;
    scanf("%d", &a);
}
```

# Input/Output

## C++

```
#include <iostream>
int main() {
    int a;
    float j;
    char c;
    cin >> a >> j >> c;
    cout << a << " " << j << "
" << c << endl;
}
```

## C

```
#include <stdio.h>
int main() {
    int a;
    float j;
    char c;
    scanf("%d %f %c", &a, &j, &c);
    printf("%d %f %c", a, j, c);
}
_____
```

# Namespace

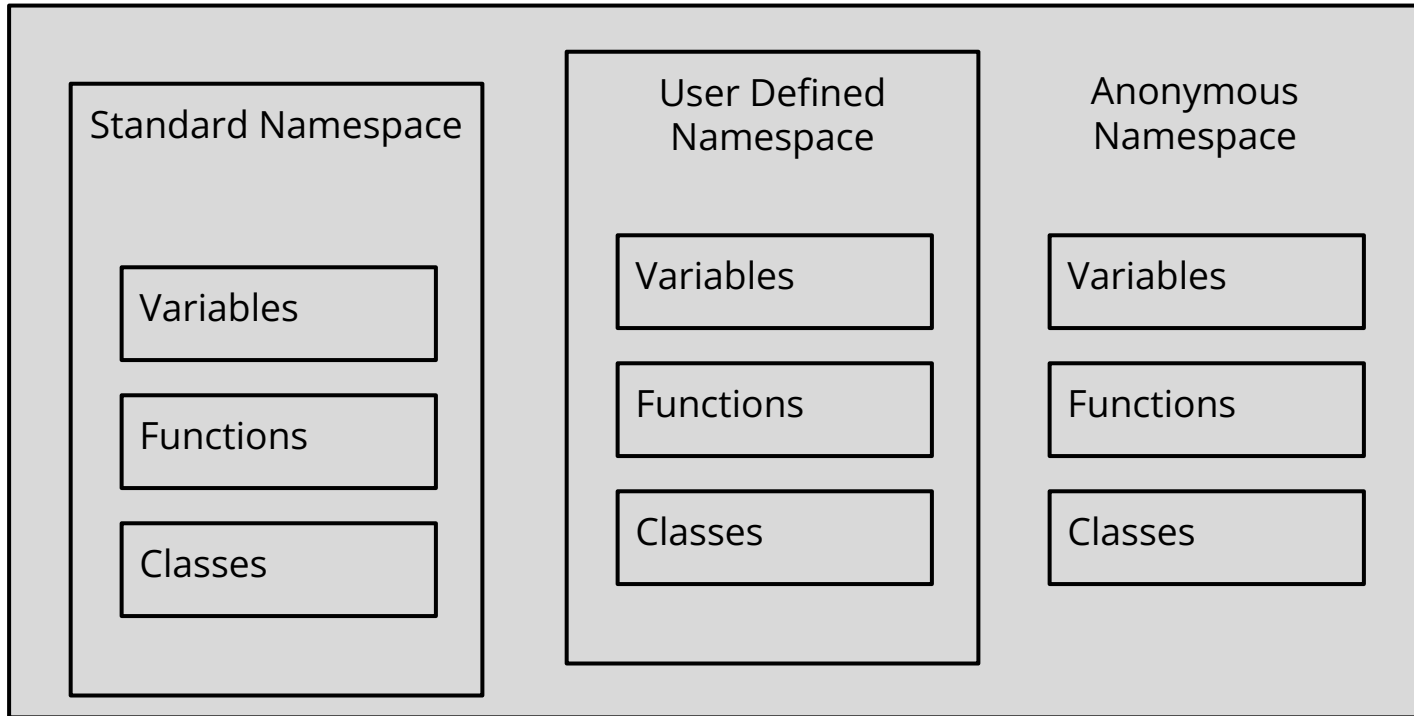
- ❑ Additional information to differentiate similar functions, classes, variables etc. with the same name, available in different libraries.
- ❑ Namespace, defines a scope. To call namespace-enabled version of either function or variable, prepend the namespace name to the function / variable:

```
std::cout << "hello"<<std::endl;  
std::cin.get();
```

- ❑ This can be avoided with preprocessor directives :

```
using namespace std; // std is abbreviation of namespace standard  
.....  
cout <<"hello"<<endl;
```

# Namespaces





# Try it

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

void display(){
    cout << "Display function" << endl;
}

void display(){
    cout << "Another Display function" << endl;
}

int main(){
    display();
}
```

**What will be the Output of this code?**

# Solution - Namespace

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

namespace MyNamespace {
    void display(){
        cout << "Display function" << endl;
    }
}

void display(){
    cout << "Another Display function" << endl;
}

int main(){
    MyNamespace::display();
}
```

**The compilation error can be resolved using the concept of namespace.**

# Try it

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

namespace first {
    int val1 = 500;
}
namespace first {
    int val2 = 501;
}
int main(){
    cout << first::val1 << "\n";
    cout << first::val2 << "\n";
    return 0;
}
```

**Extension of namespace**

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

```
namespace {
    int rel = 300;
}
```

```
int main() {
    cout << rel << "\n";
    return 0;
}
```

**Unnamed namespace**

# Try it

Is it possible to declare 2 namespaces with the same name?

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

namespace Alpha{
    int c = 100, d = 500;
    void sum(){
        cout << c+d << endl;
    }
}

namespace Alpha{
    int a = 1, b = 5;
    void sum(){
        cout << a+b << endl;
    }
}

int main() {
    Alpha::sum();
    return 0;
}
```

# Try it

```
#include <iostream>
using namespace std;
namespace first_space {
    void display() {
        cout << "Welcome to Earth!" << endl;
    }
}
namespace second_space{
    void display(){
        cout << "Welcome to Mars!" << endl;
    }
}
using namespace first_space ;
int main () {
    display();
    second_space::display();
    return 0;
}
```

**Output:**  
Welcome to Earth!  
Welcome to Mars!

# Self Study

---

- ❑ Different ways of accessing namespace
  - ❑ Normal way
  - ❑ “using” directive
- ❑ Nested namespaces
- ❑ Namespace Aliasing

Reference: <https://www.geeksforgeeks.org/namespace-in-c/?ref=rp>

# Practice Problem

1. Define two namespaces “**SquareSpace**” and “**CircleSpace**” both containing a function with same name “*Area*”. Use necessary variables.
2. Take input of a variable **a**. Call both functions from main program and show the output. Provide the input as parameter.
3. The “*Area()*” function in the **SquareSpace** should return the area of a square considering the parameter.
4. The “*Area()*” function in the **CircleSpace** should return the area of a circle considering the parameter.

**Thanks**



# Class & Objects

Course Code: CSE 205

Course Title: Object Oriented Programming

---

**NAFIZ IMTIAZ KHAN**

Lecturer, Department of CSE, MIST

Email: [nafiz@cse.mist.ac.bd](mailto:nafiz@cse.mist.ac.bd)

# Class

- ❑ The building block of C++ that leads to Object Oriented Programming is **Class**.
- ❑ A user-defined data type which groups together (**Encapsulates**) related pieces of information about an abstract thing.
- ❑ Class consists of Member Variables (data members) and Member Functions.
- ❑ It can only be accessed and used by creating an **instance** of that class.
- ❑ As class is an abstract thing, it does not occupy memory but its instance does.
- ❑ A class is like a blueprint for an **object/instance**.

# Objects

- ❑ Objects are the basic *run-time entities* in an object oriented system.
- ❑ **Abstraction :**
  - ❑ Chosen to match closely the real-world objects.
  - ❑ May represent a person, a place, a bank account, a table of data or any item that the program must handle.
- ❑ Objects take up space in the memory and have an associated address like structure in C.
- ❑ Each object has its own copy of the class member vars. But there will only be one definition for member function in memory, which is shared by all objects of the class.
- ❑ **Messaging :** When a program is executed the objects interact by sending messages to one another.

# Objects

---

- ❑ Accessing member vars of objects
  - ❑ Using object name and dot operator.
  - ❑ Using member functions.
- ❑ Objects of similar types can be assigned to one another.
- ❑ Assigning one obj to another will copy all the fields.

**No Physical Entity**

```
class Student{  
    private:  
        int id;  
        string name;  
        double cgpa;  
    public:  
        void setName(string n){  
            name = a;  
        }  
};
```

keyword

user defined name

access  
specifier

data members

member  
functions

don't forget the  
semicolon

```
int main() {  
    Student s;  
    s.setName("Verstappen");  
}
```

**Occupies Memory**

# Class

```
#include <iostream>
using namespace std;
class Account {
private:
    string name;
    int acctNo, balance;
public:
    void setBalance(int b);
    string getName();
    int getAcctNo();
};
void Account::setBalance(int b){
    balance = b;
}
string Account::getName(){
    return name;
}
....
```

**Class**

**Function Prototype**

**Member Function Definition**

## Try it

```
int main(){
    Account a;
    a.name = "Fahim";
    a.acctNo = 700254;
    a.setBalance(5000);
    cout << a.getName() << endl;
    cout << a.name << endl;
}
```

# Access Protection

- ❑ Access specifiers are used to specify access restriction to the class members.
- ❑ **Data hiding** is made possible by access specifiers.
- ❑ C++ class and struct have three levels of protection.
  - ❑ Private
    - ❑ Usage - Default / **private**:
    - ❑ Private vars accessible only by other member functions of the class & *friend functions*
    - ❑ Private functions can't be called using objects!
    - ❑ Private member vars are not accessible from child class objects!
  - ❑ Public
    - ❑ Usage - **public**:
    - ❑ Accessible by other members and by any other part of the program that contain the class.
    - ❑ Can set and get the value of public variables without any member function.

# Access Protection

## Protected

- Usage - **protected**:
- Protected vars can only be accessed using member functions of the class.
- **Protected functions can't be called using objects !**
- Inherited protected vars can be accessed using public functions in child / derived classes.

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived class	yes	yes	no
not members	yes	no	no



# Access Specifiers

- ❑ Private and Protected vars and functions can't be accessed from any other code written outside the class, not even by objects of the same class.
- ❑ Private vars are not inherited by a child class. Otherwise encapsulation could be bypassed by inheritance.
- ❑ All protected and public vars of base class, are inherited by a child class.
- ❑ Inherited protected and public vars can be accessed using public functions of the child class.
- ❑ **Access specifiers are also used in inherited class definition.**
- ❑ Compiler enforces the access protection.

# Accessor Mutator Methods

---

If all the variables in a class are private, then how do we access the values?

Using **Accessor** and **Mutator** methods -

- ***Accessor***: Also known as getter, returns the value of the private member variable. This method has a return type and usually no parameters.
- ***Mutator***: Widely known as setter methods, used to control changes to a variable. This method usually has an arguments and void as return type.

# Account Class

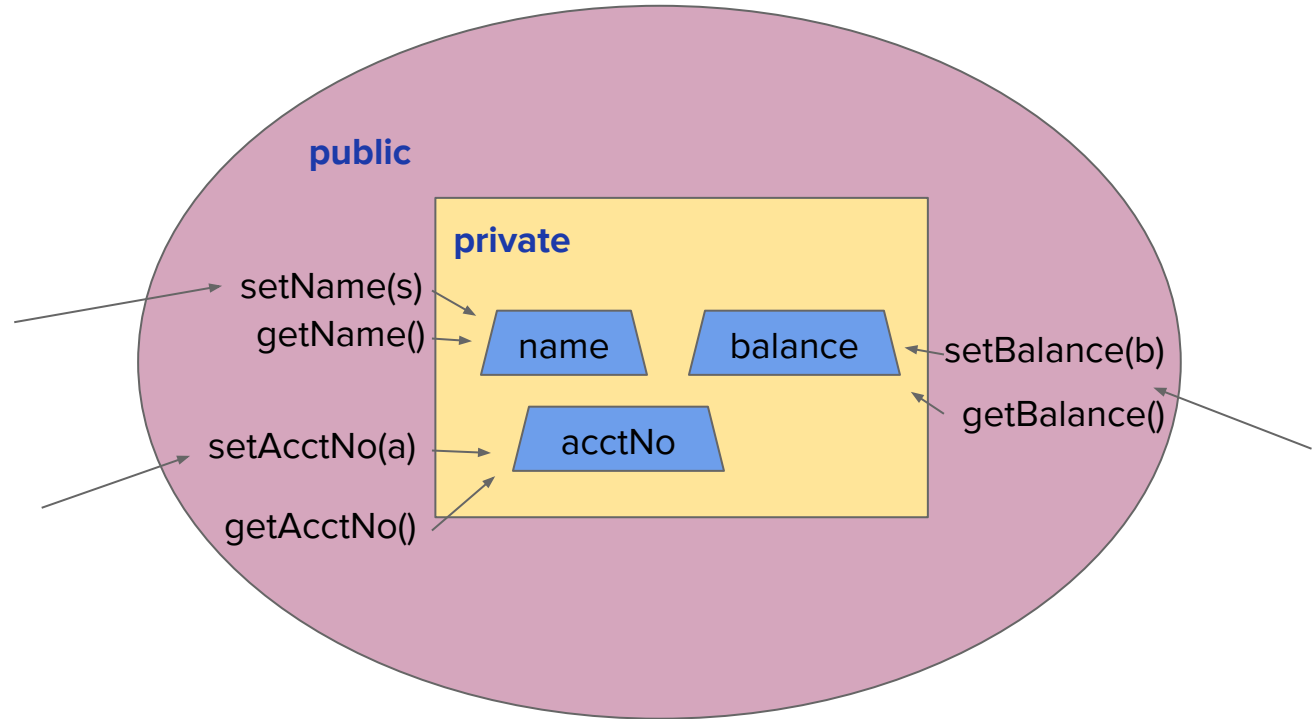
```
#include <iostream>
using namespace std;
class Account {
    string name;
    int acctNo, balance;
public:
    void setBalance(int b);
    int getBalance();
    string getName();
    void setName(string n);
    int getAcctNo();
    void setAcctNo(int a);
};
```

Public Interface

```
int Account::getBalance(){
    return balance; }
void Account::setBalance(int b){
    balance = b; }
void Account::setName(string n){
    name = n; }
string Account::getName(){
    return name; }
int Account::getAcctNo(){
    return acctNo; }
void Account::setAcctNo(int a){
    acctNo = a;
}
```

# Access Specifier: Visualization

- Inside the class, all members are visible
- Access protection pertains to clients
- Clients only access public members.



# Inheritance: Access Specifiers

## Syntax

**class** *derived\_class* : **public** *base\_class*

- ❑ The keyword **public** can be replaced with **protected** or **private**
- ❑ The access specifier limits the most accessible level for the members inherited from the base class.

## Example

**class** *cseStudent* : **protected** *mistStudent*

- ❑ It means the *cseStudent* inherits the public members of *mistStudent* but with protected access level. The protected member variables are inherited as such with same level.
- ❑ If *no access level* is specified for the inheritance, the compiler assumes **private**.

# Try it



```
class Demo_1 {  
public:  
    void f();  
};
```

```
class Demo_2 {  
public:  
    void f();  
};
```

```
void f() {  
    cout << "Global function" << endl;  
}  
  
void Demo_1::f() {  
    cout << "Demo_1 function" << endl;  
}  
  
void Demo_2::f() {  
    cout << "Demo_2 function" << endl;  
}
```

# Constructor & Destructor

Course Code: CSE 205

Course Title: Object Oriented Programming

---

**NAFIZ IMTIAZ KHAN**

Lecturer, Department of CSE, MIST

Email: [nafiz@cse.mist.ac.bd](mailto:nafiz@cse.mist.ac.bd)

# Object Initialization and Destruction

---

- ❑ Construction involves **memory allocation** and initialization for objects.
- ❑ Destruction involves **cleanup** and **deallocation of memory** for objects.



# Constructors and Destructors

```
class Student{
    int id;
public:
    void init() {
        id = 0;
    }
    int getId() {
        return id;
    }
};

void main() {
    Student st;
    st.init();
    cout << st.getId();
}
```

```
class Student{
    int id;
public:
    Student() {
        id=0;
    }
    ~Student() { }
    int getId() {
        return id;
    }
};

void main(){
    Student st;
    cout << st.getId();
}
```

**This Constructor is called and *id* is initialized.**

**Destructor**

**When the object is declared here**

# Constructors and Destructors

```
class Student{
    int id;
public:
    void setID(int id) {
        this.id = id; }
    int getId() {
        return id; }
};

void main() {
    Student st;
    st.setID(16);
    cout << st.getId() << endl;
}
```

```
class Student{
    int id;
public:
    Student(int id) {
        this->id = id;
    }
    ~Student() { }
    int getId() {
        return id;
    }
};

void main(){
    Student st(16);
    cout << st.getId();
}
```

# Constructor

- ❑ A ***constructor*** is a special member function
- ❑ Exact same name as its class
- ❑ No return type, not even void
- ❑ It had to be public
- ❑ Invoked implicitly when instance/object is created
- ❑ Can be overloaded
- ❑ Are called in the order of creation
- ❑ Address cannot be taken

# Constructor

- ❑ Types of Constructors -
  - ❑ Default-like / zero arg constructor
  - ❑ Parameterized
  - ❑ Copy constructor
- ❑ If a constructor with parameter is defined then **default constructor is not available**. Either, one has to use a zero arg constructor, or use default arguments with the parameterized constructor.

# Destructors

- ☐ Called when an instance is deallocated.
- ☐ Name is `~class_Name()`
- ☐ Does not take any arguments
- ☐ Cannot be overloaded
- ☐ Does not have a return type
- ☐ Almost always called implicitly
- ☐ Are called in the reverse order of creation
- ☐ Address cannot be taken

# Constructor & Destructor

- ❑ Like other member functions, constructors and destructors are declared within a class declaration. They can be defined inline or external to the class declaration.
- ❑ Constructors are also called when local or temporary class objects are created, and destructors are called when local or temporary objects go out of scope.

# Example Code 01

```
class Student {  
    int id;  
public:  
    Student(int i) {  
        id = i;  
        cout << "Constructing "  
            << id << endl; }  
    ~Student() {  
        cout << "Destructing "  
            << id << endl; }  
  
    int getId() {  
        return id; }  
  
};
```

```
int main(){  
    Student st1(1), st2(2);  
    cout << st1.getId() << endl;  
    cout << st2.getId() << endl;  
  
}
```

```
Output:  
Constructing 1  
Constructing 2  
1  
2  
Destructing 2  
Destructing 1
```

## Example Code 02

```
class Student {  
    int id;  
public:  
    Student(int i) {  
        id = i;  
        cout << "Constructing "  
            << id << endl; }  
    ~Student() {  
        cout << "Destructing "  
            << id << endl; }  
  
    int getId() {  
        return id; }  
  
};
```

```
int main(){  
    Student st1(1), st2;  
    cout << st1.getId() << endl;  
    cout << st2.getId() << endl;  
  
}
```

**Compilation Error!**  
**No default constructor is present for  
initializing st2.**



## Example Code 03

```
class Number {  
    int val;  
public:  
    Number(int x) {  
        val = x;  
    }  
};
```

```
int main(){  
    Number a;           //Error  
  
    Number b(9);        //b.val = 9  
  
    Number c = 10;       //c.val = 10  
  
    Number arr[5];       //Error  
    Number arr_[] = {Number(1),  
                     Number(2), Number(3)};  
    //arr_[0].val = 1, arr_[1].val = 2  
  
    return 0;  
}
```

# Constructors

- ❑ You can create objects using constructors as below:

Number ob;        *//default constructor called*

Number ob( );     *//default constructor called*

Number ob(5);    *//parameterised constructor called*

Number ob = 5;    *//parameterised constructor called*

- ❑ You may consider the following for ease of understanding:

Student s1( )        *//create s1 using the constructor student( )*

Student s1("16", "Shafin")    *//create s1 using parameterised constructor*

*Student(int id, string name)*

- ❑ In C, variable declaration without initialization is possible
- ❑ In C++ when objects are declared they are initialized always by a call to default constructor at least. So important to init variables inside constructors.

## Example Code 04

```
class Student {
    int id;
public:
    Student() {
        id = 0;
        cout << "Constructing "
              << id << endl; }
    Student(int i){
        id = i;
        cout << "Constructing "
              << id << endl; }
    ~Student() {
        cout << "Dest " << id << endl; }

    int getId() { return id; }
};
```

```
int main(){
    Student st1(1), st2;
    cout << st1.getId() << endl;
    cout << st2.getId() << endl;
}
```

```
Output:
Constructing 1
Constructing 0
1
0
Dest 0
Dest 1
```

## Example Code 05

```
class Student {  
    int id; char* name;  
public:  
    Student(char* p, int q) {  
        id = q;  
        name = new char[strlen(p)];  
        strcpy(name, p);  
        cout << "Constructing "  
                << name << endl; }  
    ~Student() {  
        cout << "Dest " << name << endl;  
        delete [] name; }  
    int getId() { return id; }  
};  
void f(Student st){cout << st.getId() <<  
                    endl;}
```

```
int main(){  
    Student st1("st1", 16);  
    func(st1);  
}
```

**Output:**  
Constructing st1  
16  
Dest st1  
Dest st1

# Question

---

## **How to prevent the destructor from being called twice?**

Solution: Restrict destructor while returning from the function  $f()$ . If there is no constructor there will be no destructor.

## **What if a reference is passed instead of an object?**

Destructor will be called once cause passing the reference doesn't invoke the call of the “invisible” constructor.

## Example Code 06 (a)

```
class Student {
    int id; char* name;
public:
    Student(char* p, int q) {
        id = q;
        name = new char[strlen(p)];
        strcpy(name, p);
        cout << "Constructing "
              << name << endl; }
    ~Student() {
        cout << "Dest " << name << endl;
        delete [] name; }
    int getId() { return id; }
};
void f(Student *st){cout << st->getId() <<
                    endl;}
```

```
int main(){
    Student st1("st1", 16);
    f(&st1);
}
```

```
Output:
Constructing st1
16
Dest st1
```

## Example Code 06 (b)

```
class Student {
    int id; char* name;
public:
    Student(char* p, int q) {
        id = q;
        name = new char[strlen(p)];
        strcpy(name, p);
        cout << "Constructing "
              << name << endl; }
    ~Student() {
        cout << "Dest " << name << endl;
        delete [] name; }
    int getId() { return id; }
};
void f(Student &st){cout << st->getId() <<
                  endl;}
```

```
int main(){
    Student st1("st1", 16);
    f(st1);
}
```

**Passing pointers or references to a function doesn't invoke the call of the invisible constructor. Thus destructor is called once only.**

**Output:**  
Constructing st1  
16  
Dest st1

## Example Code 07

```
class Student {
    int id; char* name;
public:
    Student(char* p, int q) {
        id = q;
        name = new char[strlen(p)];
        strcpy(name, p);
        cout << "Constructing "
              << name << endl; }
    ~Student() {
        cout << "Dest " << name << endl;
        delete [] name; }
    int getId() { return id; }
};
```

```
Student f(Student st1){
    Student tmpSt("tmpSt", 38);
    return tmpSt;
}

int main(){
    Student st1("st1", 16), st2("st2", 55);
    st2 = f(st1);
    cout << st2.getId() << endl;
}
```

```
Output:
Constructing st1
Constructing st2
Constructing tmpSt
Dest tmpSt (Local variable)
Dest st1 (Function arg)
Dest tmpSt (Return object)
38
Dest tmpSt
Dest st1
```



# Why tmpSt is destructed twice?

This is a special case where the compiler is allowed to optimize out the copy: this is called *Named Return Value Optimization (NRVO)*.

Basically, the compiler allocates memory for the return object on the **call site** and lets the function fill in that memory directly instead of creating the object at the **called site** and copying it back.

Modern compilers do this routinely whenever possible (**there are some situations where this isn't easy since there are several return paths in the function that return different instances**).

## Question

**Is it possible to leave out one of the arguments to the constructor for a class, when creating an array with a parameterised constructor?**

```
Student(){id = 0; name = ""}  
Student(int roll, string n){id = roll; name = n;  
  
Student arr[3] = {{1, "Zakir"}, {2, "Mahfuj"}};
```

The above statement makes a call to the parameterised constructor and creates two students arr[0] and arr[1] with the arguments provided. For the third student i.e, arr[2], the default constructor will be invoked and it will be initialised to id = 0 and name = "".

# Dynamic Memory Allocation

# Scope of a Variable

Three places

```
#include <stdio.h>

int g;                                //g -> global variable

void function(int n)                  //n -> formal param
{

}


int main()
{
    int m = 10;                       //m -> local variable
}
```

# Scope of a Variable

Scope of n and m

```
#include <stdio.h>
```

```
int g;                                //g -> global variable
```

```
void function(int n)                 //n -> formal param  
{  
      
}
```

```
int main()  
{  
        int m = 10;                        //m -> local variable  
}
```

# Scope of a Variable

Scope of g

```
#include <stdio.h>
```

```
int g;
```

//g -> global variable

```
void function(int n)  
{  
  
}
```

//n -> formal param

```
int main()  
{  
    int m = 10;  
}
```

//m -> local variable

# Global Variable Initialization

Automatically initialized

Data Type	Initializer
int	0
char	'\0'
float	0
double	0
pointer	NULL

# Global Variable Initialization

Automatically initialized

```
#include <stdio.h>

int m; //global variable

int main()
{
    printf("%d", m);    //0
}
```



# Variable Shadowing

In case of same name, local variable takes preference

```
#include <stdio.h>

int g = 10; //global

int main()
{
    int g = 20; //local

    printf("%d", g);    //20
}
```

# Destruction of local variables

Variable is **destroyed** when it goes **out-of-scope**

```
#include <stdio.h>
void fn()
{
    int m = 0;
    m = m + 1;
    printf("%d\n", m);
} //m is now destroyed

int main()
{
    fn(); //1
    fn(); //1
}
```

# The static Keyword

Prevents local variable from being destroyed

```
#include <stdio.h>
void fn()
{
    static int m = 0;    //initializes only once
                        //in the lifetime
    m = m + 1;
    printf("%d\n", m);
} //m is not destroyed

int main()
{
    fn();    //1
    fn();    //2
}
```

# Reference to Local

Pointer referring to expired local variable

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    int n = 10;
```

```
    int * p;
```

```
}
```

# Reference to Local

Pointer referring to expired local variable

```
#include <stdio.h>

int main()
{
    int n = 10;
    int * p;

    if (n == 10)
    {
        int m = 20;    //m -> local variable
        p = &m;
    }

}
```

# Reference to Local

Pointer referring to expired local variable

```
#include <stdio.h>

int main()
{
    int n = 10;
    int * p;

    if (n == 10)
    {
        int m = 20;    //m -> local variable
        p = &m;
    }
    //m is now destroyed
    printf("%d", *p);
}
```

# Reference to Local

Pointer referring to expired local variable

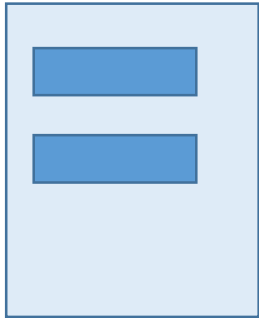
```
#include <stdio.h>

int main()
{
    int n = 10;
    int * p;

    if (n == 10)
    {
        int m = 20;    //m -> local variable
        p = &m;
    }
    //m is now destroyed
    printf("%d", *p);    //Undefined behaviour
                        //can be 20, or show garbage, or crash
}
```

# Dynamic Memory Allocation

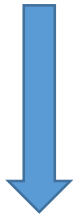
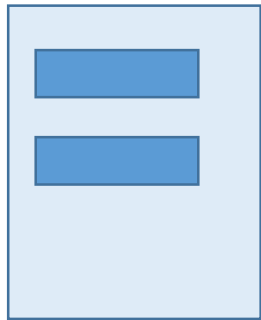
The concept



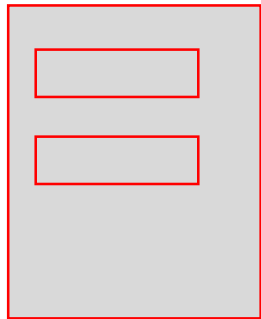


# Dynamic Memory Allocation

The concept

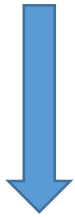
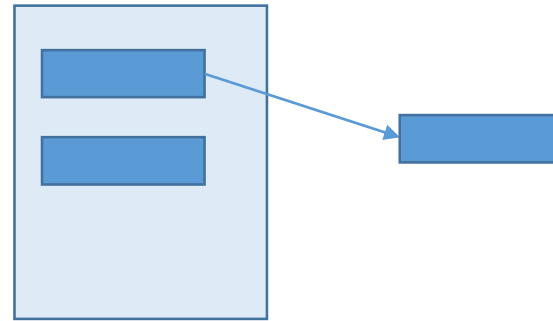
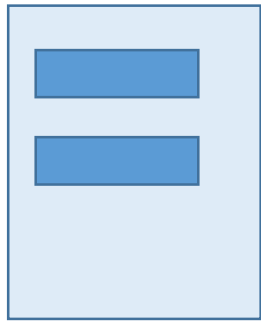


Out-of-Scope

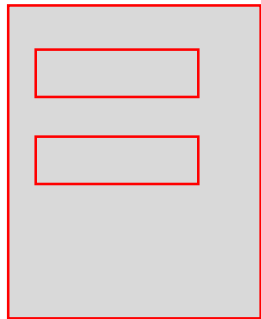


# Dynamic Memory Allocation

The concept

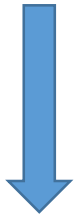
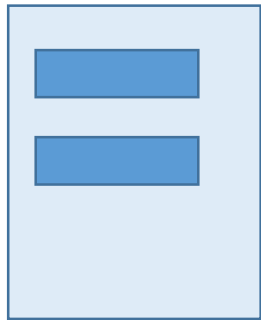


Out-of-Scope

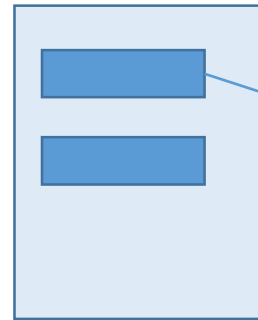
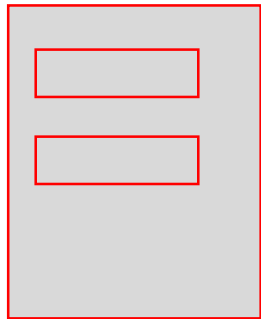


# Dynamic Memory Allocation

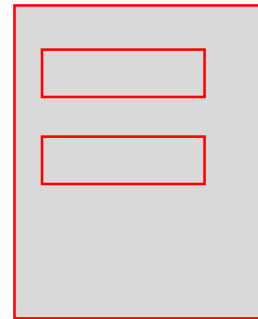
The concept



Out-of-Scope

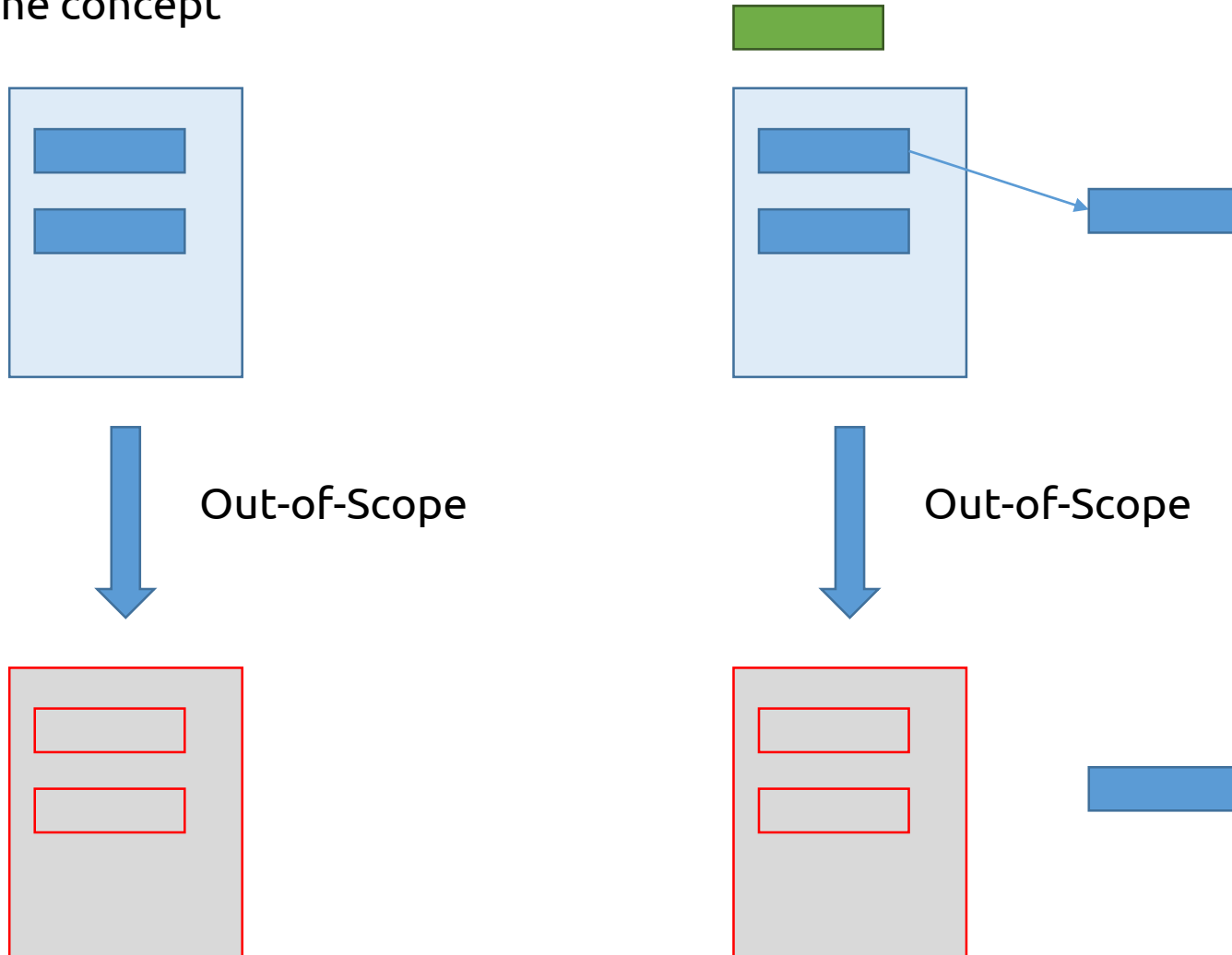


Out-of-Scope



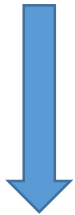
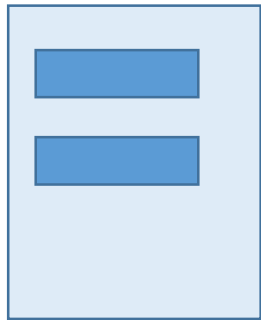
# Dynamic Memory Allocation

The concept

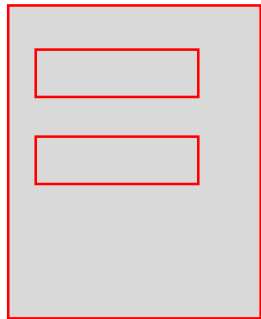


# Dynamic Memory Allocation

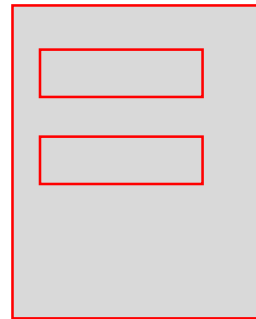
The concept



Out-of-Scope

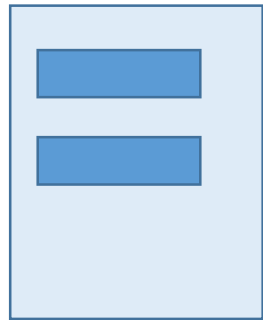


Out-of-Scope

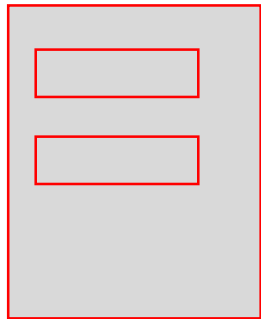


# Dynamic Memory Allocation

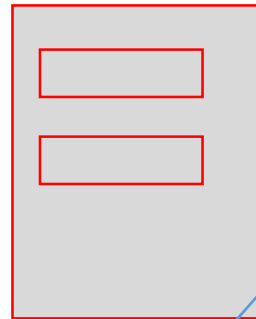
The concept



Out-of-Scope



Out-of-Scope



# The malloc function

```
void * malloc (size_t size);
```



Memory required in byte  
e.g. sizeof (int)

Returns pointer to the beginning of the block

According to the 1999 ISO C standard (C99), `size_t` is an unsigned integer type of at least 16 bit. This type is used to represent the size of an object.

- [https://en.wikipedia.org/wiki/C\\_data\\_types#stddef.h](https://en.wikipedia.org/wiki/C_data_types#stddef.h)
- <https://stackoverflow.com/questions/2550774/what-is-size-t-in-c>

# The malloc function

```
#include <stdio.h>

int main()
{
    int n = 10;
    int * p;

    if (n == 10)
    {
        int * m;
        m = (int *)malloc(sizeof (int));
        *m = 20;
        p = m;
    }
    //m is now destroyed
    //but the location is preserved
    //which it pointed by p also
    printf("%d", *p);
}
```

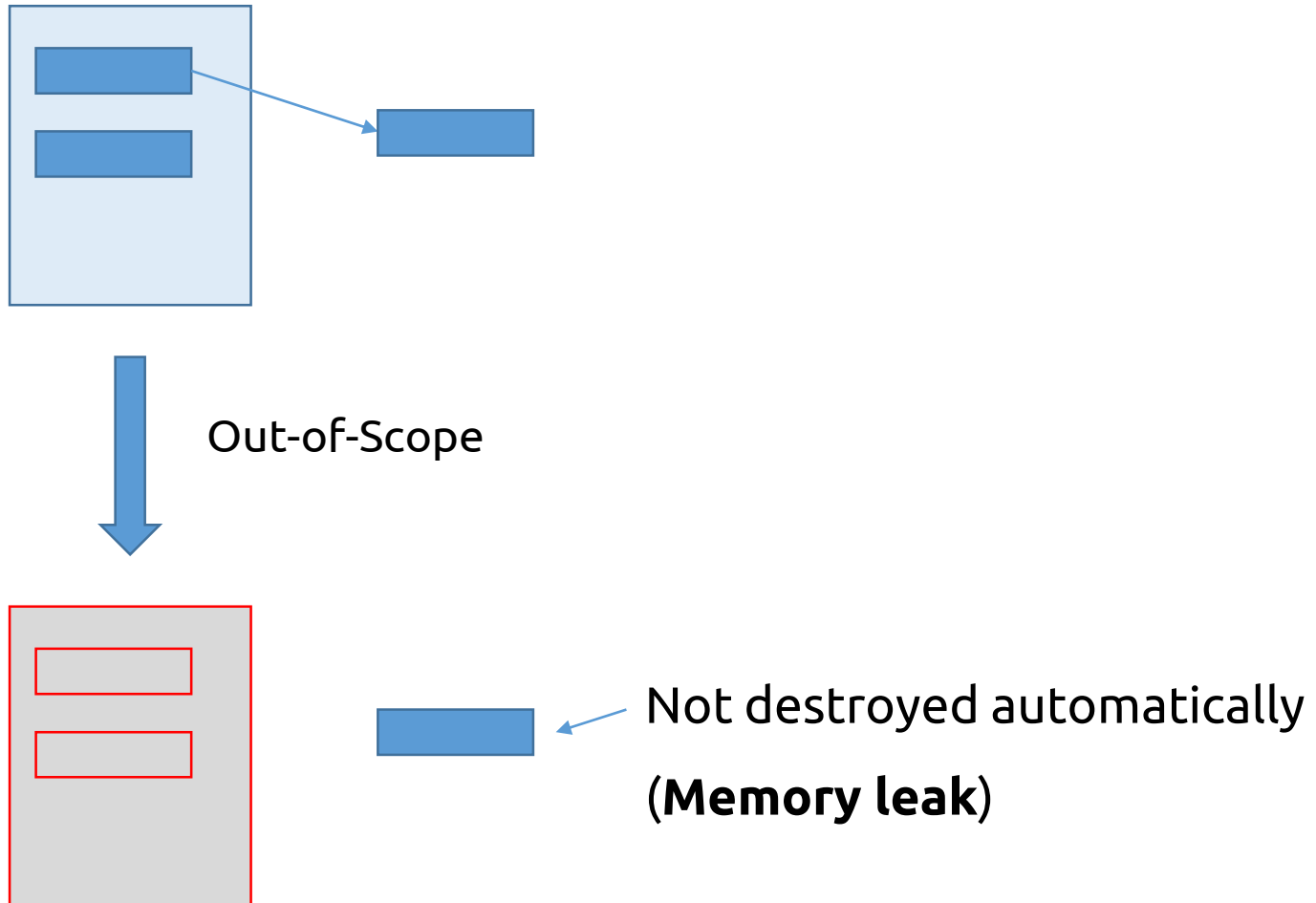


# Usage of malloc function

- Dynamic allocation of
  - Array
  - Struct
- Factory Methods

# Destruction of Dynamic Memory

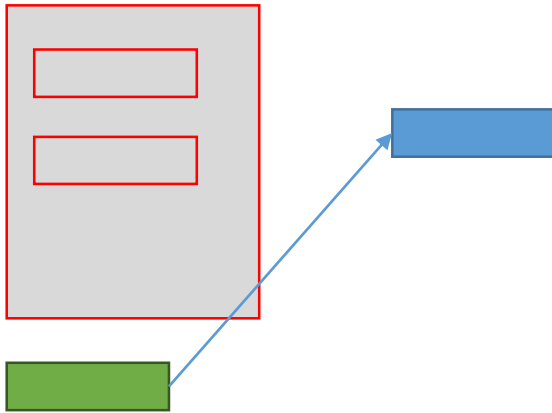
When is it destroyed?



# The free function

When the outside (dynamic) memory is no longer needed

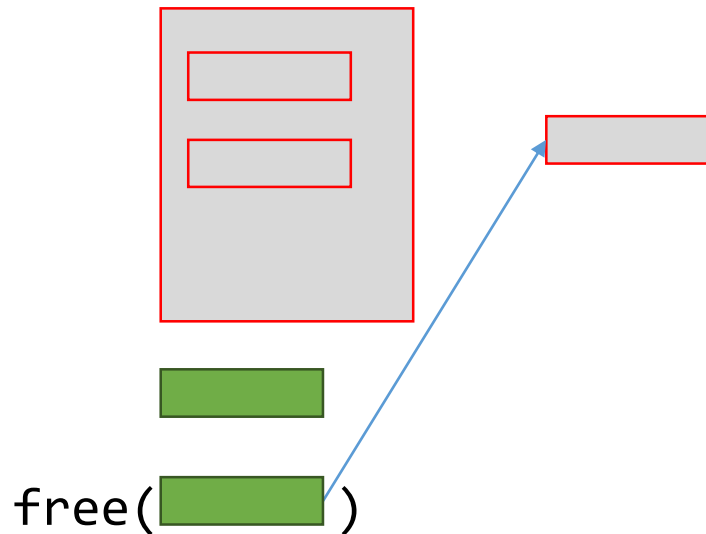
```
void free(void *ptr)
```



# The free function

When the outside (dynamic) memory is no longer needed

```
void free(void *ptr)
```



# The calloc function

Same as malloc

```
void *calloc(size_t nitems, size_t size)
```



How many items?    Size of each item

The following two lines produces similar allocation

```
malloc(20 * sizeof (int));
```

```
calloc(20, sizeof (int));
```

# The calloc function

```
void *calloc(size_t nitems, size_t size)
```

- Initializes every bit to zero
- Slower than malloc

# The realloc function

For resizing existing dynamic memory

```
void *realloc(void *ptr, size_t size)
```



Existing pointer



New size

Either memory is extended,

Or Previous items are copied to new larger location

# The realloc function

```
void *realloc(void *ptr, size_t size)
```



Existing pointer



New size

```
int * m = (int *) calloc(2, sizeof(int));
```



# The realloc function

```
void *realloc(void *ptr, size_t size)
```



Existing pointer



New size

```
int * m = (int *) calloc(2, sizeof(int));
```

```
m[0] = 10;
```

```
m[1] = 20;
```

```
printf("%d %d \n", m[0], m[1]);
```

# The realloc function

```
void *realloc(void *ptr, size_t size)
```



Existing pointer



New size

```
int * m = (int *) calloc(2, sizeof(int));
```

```
m[0] = 10;
```

```
m[1] = 20;
```

```
printf("%d %d \n", m[0], m[1]);
```

```
m = (int *) realloc(m, 3 * sizeof(int));
```

# The realloc function

```
void *realloc(void *ptr, size_t size)
```



Existing pointer



New size

```
int * m = (int *) calloc(2, sizeof(int));
```

```
m[0] = 10;
```

```
m[1] = 20;
```

```
printf("%d %d \n", m[0], m[1]);
```

```
m = (int *) realloc(m, 3 * sizeof(int));
```

```
m[2] = 30;
```

# The realloc function

```
void *realloc(void *ptr, size_t size)
```



Existing pointer



New size

```
int * m = (int *) calloc(2, sizeof(int));
```

```
m[0] = 10;
```

```
m[1] = 20;
```

```
printf("%d %d \n", m[0], m[1]);
```

```
m = (int *) realloc(m, 3 * sizeof(int));
```

```
m[2] = 30;
```

```
printf("%d %d %d\n", m[0], m[1], m[2]);
```

```
free(m);
```

# Function Overloading

Course Code: CSE 205

Course Title: Object Oriented Programming

---

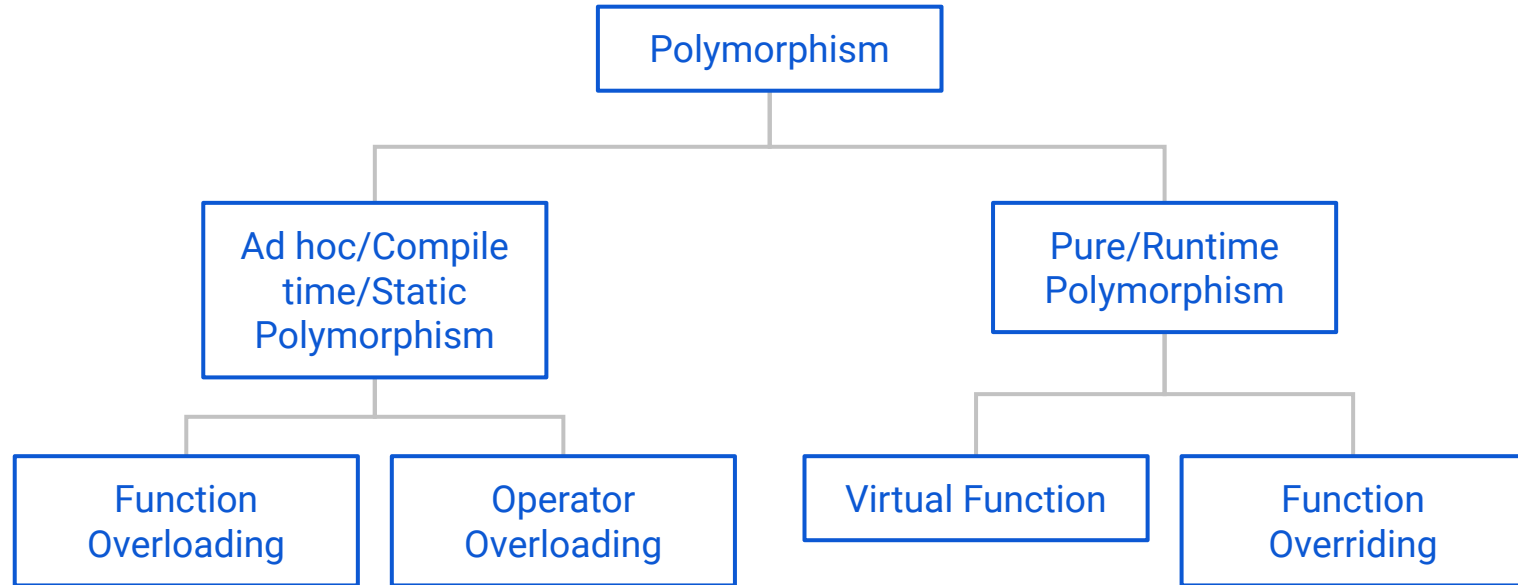
**NAFIZ IMTIAZ KHAN**

Lecturer, Department of CSE, MIST

Email: [nafiz@cse.mist.ac.bd](mailto:nafiz@cse.mist.ac.bd)

# Polymorphism

**Polymorphism** is a mean of giving different meaning to the same message. The meanings are dependent on the type of data being processed.



# Polymorphism

## ❑ Compile Time Polymorphism

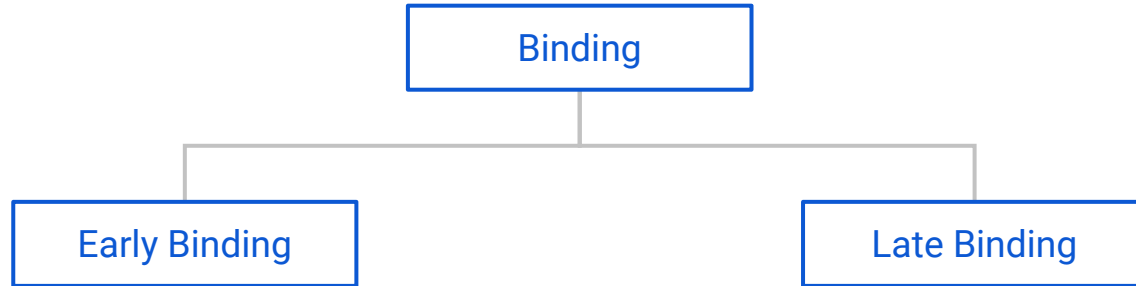
- ❑ **Function Overloading:** gives the same name but has different meaning/definition. The name has several interpretations that depends on function selection.
- ❑ **Operator Overloading:** provides different interpretation of C++ operators depending on type of its argument. *[Will be covered later]*

## ❑ Runtime Polymorphism

Runtime function selection. Related to inheritance. Examples are Function Overriding and Virtual Functions. *[Will be covered later]*

# Binding

**Binding** refers to the process that is used to convert identifiers (such as variable and function names) into addresses.





# Early Binding

---

- ❑ **Early binding** refers to events that occur at compile time.
- ❑ Early binding occurs when all information needed to call a function is known at compile time.

## Examples:

function calls, overloaded function calls, and overloaded operators

# Late Binding

---

- **Dynamic Binding** occurs at run time and is also called late binding.
- **Late binding** refers to function calls that are not resolved until run time.
- Late binding can make for somewhat **slower** execution times.

**Example:**

virtual functions

# Function Overloading

Based on in which scope a function is defined it can be divided into 2 types -

- ❑ **Member Functions:** defined inside a class. (or at-least declared inside the class and defined outside with scope resolution operator)
- ❑ **Free Functions:** defined in global scope.

In C, every function has a unique name.

C++ allows several functions with the same name with some restrictions.

# Perks - Function Overloading

- ❑ We use function overloading to save the memory space, consistency, and readability of our program.
- ❑ With the use function overloading concept, we can develop more than one function with the same name
- ❑ Function overloading shows the behavior of polymorphism that allows us to get different behavior, although there will be some link using the same name of the function.
- ❑ Function overloading speeds up the execution of the program.
- ❑ Function overloading is used for code reusability and also to save memory.
- ❑ It helps application to load the class method based on the type of parameter.
- ❑ Code maintenance is easy.

# Disadvantages - Function Overloading

---

- ❑ Function declarations that differ only by its return type cannot be overloaded with function overloading process.
- ❑ Member function declarations with the same parameters or the same name types cannot be overloaded if any one of them is declared as a static member function.

# Example

## C

```
int abs(int i);
float fabs(float f);
long labs(long l);

void main(){

    int i = -2;
    float f = -3.4;
    long l = -34001;
    cout << abs(i);
    cout << fabs(f);
    cout << labs(l);

}
```

## C++

```
int abs(int i);
float abs(float f);
long abs(long l);

void main(){

    int i = -2;
    float f = -3.4;
    long l = -34001;
    cout << abs(i);
    cout << abs(f);
    cout << abs(l);

}
```

# Overloading Mechanism

- ❑ Functions are differentiated by their names and argument list
- ❑ For overloading, argument list must vary in --
  - ❑ **Number of arguments**
  - ❑ **Types of arguments**
  - ❑ **Order of arguments**
- ❑ Only return types can't be used to differentiate functions

Example:

```
int func(int i, float f);  
float func(int i, float f);
```

Error: overloading is not possible with different return types only

# Question

```
int fun(int a, int b){  
    return a + b;  
}
```



```
float fun(int x, int y) {  
    return x - y;  
}
```

```
int fun(int a, int b){  
    return a + b;  
}
```



```
float fun(float x, float y) {  
    return x - y;  
}
```



# Overloading Example

```
int totalMark(int phy, int chem){  
    int total = phy + chem;  
    return total;  
}  
  
int totalMark(int phy, int chem, int m){  
    int total = phy + chem + m;  
    return total;  
}  
  
int totalMark(int chem, int phy){  
    int total = chem + phy;  
    return total;  
}
```

Error: This doesn't change the order of arguments. Both are integers. Not overloading

```
double avgGPA(double gpa, int term){  
    return gpa/term;  
}  
  
double avgGPA(int term, double gpa){  
    return gpa/term;  
}
```

Overloaded. Different order of arguments.

# Default Argument

- ❑ Default argument lets to overload with smaller code.
- ❑ Must be to the right of any parameter that doesn't have default value
- ❑ Default can not be specified both in prototype and definition
- ❑ Default arguments must be constants or global variables

## Example:

```
int totalMarks(int phy, int chem, int math = 0){  
    int total = phy + chem + math;  
    return total;  
}  
  
void main(){  
    cout << totalMarks(80, 90, 100) << endl;  
    cout << totalMarks(80, 70) << endl;  
}
```

# Default Argument Restrictions

```
int totalMark(int opt = 0, int phy, int chem) {  
    int total = phy + chem + opt ;  
    return total;  
}
```

Error: Must be to the right of any parameter that don't have defaults

```
int totalMark (int phy, int chem, int opt = 0);  
int totalMark ( int phy, int chem, int opt = 0) {  
    int total = phy + chem + opt;  
    return total;  
}
```

Error: Default can't be specified both in prototype and definition

```
int totalMark(int phy, int chem, int opt = phy) {  
    int total = phy + chem + opt ;  
    return total;  
}
```

Error: Default arguments must be constant or global variables

# Thanks

# Friend Functions

Course Code: CSE 205

Course Title: Object Oriented Programming

---

**NAFIZ IMTIAZ KHAN**

Lecturer, Department of CSE, MIST

Email: [nafiz@cse.mist.ac.bd](mailto:nafiz@cse.mist.ac.bd)

# Try it!

```
class Complex {  
    double real, img;  
public:  
    setValues(double r, double i){  
        real = r;  
        img = i;  
    }  
};
```

```
Complex add(Complex a, Complex b){  
    Complex res;  
    res.setValues(a.real + b.real,  
                 a.img + b.img)  
    return res;  
}
```

```
int main(){  
    Complex a, b;  
    a.setValues(3.55, 2.18);  
    b.setValues(5.99, 4.94);  
    Complex ans = add(a, b);  
}
```

error: trying to access  
private variables of the  
Complex class from a  
non-member function.

# Solution 1: Make add() a member function

```
class Complex {  
    double real, img;  
public:  
    setValues(double r, double i){  
        real = r;  
        img = i;  
    }  
    Complex add(Complex b){  
        Complex res;  
        res.setValues(real + b.real,  
            img + b.img)  
        return res;  
    }  
};
```

```
int main(){  
    Complex a, b;  
    a.setValues(3.55, 2.18);  
    b.setValues(5.99, 4.94);  
    Complex ans = a.add(b);  
}
```

But what if we insist on keeping add() as a global function and access the private variables of Complex class without using getter methods?

## Solution 2: Make add() a friend of Complex

```
class Complex {  
    double real, img;  
public:  
    setValues(double r, double i){  
        real = r;  
        img = i;  
    }  
    friend Complex add(Complex, Complex);  
};  
Complex add(Complex a, Complex b){  
    Complex res;  
    res.setValues(a.real + b.real,  
        a.img + b.img)  
    return res;  
}
```

```
int main(){  
    Complex a, b;  
    a.setValues(3.55, 2.18);  
    b.setValues(5.99, 4.94);  
    Complex ans = add(a, b);  
}
```

Friend functions have access to all the private variables of the class it is friend to.



# Concept: Forward Declaration

- A forward declaration tells the compiler about the existence of an entity before actually defining the entity.
- It refers to the beforehand declaration of the syntax or signature of an identifier, variable, function, class, etc. prior to its usage (done later in the program).

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

class B;

class A {
public:
    int a;
    int sum(B ob){
        return a + ob.b;
    }
};

class B {
public:
    int b;
    int sum(A ob){
        return ob.a + b;
    }
};
```

error: define the function outside the class and after defining class B.

# Try it!

```
class Teacher;
class Student{
    string department;
public:
    void setDepartment(string d){
        department = d;
    }
    bool sameDepartment(Teacher teacher);
};
class Teacher{
    string department;
public:
    void setDepartment(string d){
        department = d;
    }
};
bool Student::sameDepartment(Teacher teacher){
    if(teacher.department == department)
        return true;
    else return false;
}
```

error: department is  
private variable.

```
int main(){
    Student ob1, ob2;
    ob1.setDepartment("CSE");
    ob2.setDepartment("EECE");

    cout<<ob1.sameDepartmentStd(ob2)<<endl;

    Teacher tch;
    tch.setDepartment("CSE");

    cout << ob1.sameDepartment(tch);

}
```

# Solution: Friend Function

```
class Teacher;
class Student{
    string department;
public:
    ... ..
    friend bool sameDepartment(Teacher
                               tch, Student std);
};
class Teacher{
    string department;
public:
    ... ..
    friend bool sameDepartment(Teacher
                               tch, Student std);
};
bool sameDepartment(Teacher tch, Student std){
    if(tch.department == std.department)
        return true;
    else return false;
}
```

Use the friend keyword

Friend functions have access to private variables

```
int main(){
    Student ob1, ob2;
    ob1.setDepartment("CSE");
    ob2.setDepartment("EECE");

    cout<<ob1.sameDepartmentStd(ob2)<<endl;

    Teacher tch;
    tch.setDepartment("CSE");

    cout << sameDepartment(tch, ob1) << " "
    << sameDepartment(tch, ob2) << endl;
}
```

Call as global function

# Solution: Friend Function

```
class Teacher;
class Student{
    string department;
public:
    ... ..
    bool sameDept(Teacher tch);
};
class Teacher{
    string department;
public:
    ... ..
    friend bool Student::sameDept(Teacher
                                tch);
};
bool Student::sameDept(Teacher tch){
    if(tch.department == department)
        return true;
    else return false;
}
```

Use the friend  
keyword

Friend functions  
have access to  
private variables

```
int main(){
    Student ob1, ob2;
    ob1.setDepartment("CSE");
    ob2.setDepartment("EECE");

    cout<<ob1.sameDepartmentStd(ob2)<<endl;

    Teacher tch;
    tch.setDepartment("CSE");

    cout << ob1.sameDept(tch) <<endl;
}
```

Call as  
member  
function

# Friend Function

---

- ❑ Friend functions are defined outside the class scope and have right to access all private and protected members of the class. Thus, provides “member” access to non-member function.
- ❑ Prototypes for friend functions appear in the class definition and is prefaced by keyword **friend**.
- ❑ Do not require an object to call (when member of none of the classes)
- ❑ Friend Functions are not member functions.
- ❑ Do not have ‘this’ pointer
- ❑ Can be friend with more than one class
- ❑ Friend functions are not inherited

# Friend Function Properties

---

Friendship is not

- Inherited
- Reciprocal/Implied
- Transitive

Here's a story to remember these properties -

- I don't necessarily trust the kids of my friends. The privileges of friendship aren't inherited. Derived classes of a friend aren't necessarily friends. If class Fred declares that class George is a friend, classes derived from George don't have any automatic special access rights to Fred objects.

# Friend Function Properties

- I don't necessarily trust the friends of my friends. The privileges of friendship aren't transitive. A friend of a friend isn't necessarily a friend. If class Harry declares class Ronald as a friend, and class Ronald declares class Muggle as a friend, class Muggle doesn't necessarily have any special access rights to Harry objects.
- You don't necessarily trust me simply because I declare you my friend. The privileges of friendship aren't reciprocal. If class Draco declares that class Snape is a friend, Snape objects have special access to Draco objects but Draco objects do not automatically have special access to Snape objects.

# Member Friend Function

```
class A;
class B{
    int n;
public:
    void setN(int b){ n = b;}
    void display(A ob);
};

class A{
    int n;
public:
    void setN(int a){ n = a;}
    friend void B::display(A);
};

void B::display(A ob){
    cout << this->n << " " << ob.n <<
endl;
}
```

```
int main(){

    A a;
    B b;
    a.setN(5);
    b.setN(10);
    b.display(a);
}
```

A member function of a class can be a friend of another class. In such case the function has access to all the private and protected variables of that class.



# Friend Class

- ❑ Friends can be either a function or a entire class.
- ❑ If it is a complete class, all the member functions become Friends.
- ❑ To declare all member functions of **class B** as friends of **class A**, place a following declaration in the definition of **class A** -

```
class A{  
    int n;  
    public:  
        friend class B;  
        void display();  
};
```
- ❑ All the member functions of class B will have access to all vars, including private vars of class A.

# Friend Class Example

```
class A {  
    friend class B;  
    int Aries;  
};
```

```
class B {  
    friend class C;  
    int Taurus;  
};
```

```
class C {  
    int Leo;  
    void Capricorn() {  
        A a; B b;  
        a.Aries = 0;  
        b.Taurus = 0;  
    }  
};
```

Friendship is strictly a one-one relationship between the 2 classes

Friendship is not transitive; C is not a friend of A

Class C is friend of B; C has access to B's private variables

```
class D : public C {  
    void Gemini() {  
        B b;  
        b.Taurus = 0;  
    }  
};
```

Friendship is not inherited; D is not a friend of B

```
class E : public B {  
    void Scorpio() {  
        C c;  
        c.Leo = 0;  
    }  
};
```

Friendship is not inherited; E is not a friend of C

# Thanks

# Functions

## Inline Functions

Course Code: CSE 205

Course Title: Object Oriented Programming

---

**NAFIZ IMTIAZ KHAN**

Lecturer, Department of CSE, MIST

Email: [nafiz@cse.mist.ac.bd](mailto:nafiz@cse.mist.ac.bd)

# Definition - Function

*What is function?*

- In C++, a function is a group of statements that is given a name, and which can be called from some point of the program.
- The most common syntax to define a function is:

*type* **name** ( parameter1, parameter2, ...) { statements }

Where:

*type* is the type of the value returned by the function.

**name** is the identifier by which the function can be called.

parameters Each parameter consists of a type followed by an identifier.

statement is the function's body that specify what it actually does.

# Functions

---

*Why do we use functions?*

- **Improve Readability:** `power(5, 2)` is clearer than copy-pasting the algorithm to compute the power at multiple places in the code.
- **Maintainability:** To change the algorithm, just change the function.
- **Code Reuse:** Using libraries.

# More about Functions .. ..

- ❑ Function declarations need to occur before invocations.

```
void main() {
```

```
    int x = power(2, 3);
```

```
}
```

```
int power(int a, int b) { ... .. }
```

error: 'power' was not  
declared in this scope

- ❑ **Solution**

- ❑ Reorder function declarations.

- ❑ Use a function prototype; informs compiler it'll be implemented later.

Function prototypes should match the signature of the method, though argument names don't matter.

Example: *int power(int, int);*

## More about Functions .. ..

*The return statement only allows you to return 1 value. How can you pass multiple values from a function ?*

- ☐ Using pointers
- ☐ Structures/Classes
- ☐ Arrays - but data has to be of the same type
- ☐ Using reference
- ☐ As STL tuple

Learn more about it [here](#)



# Problem with Function Call

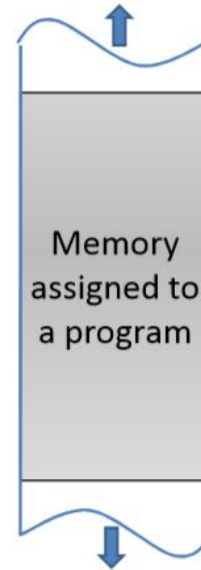
---

- ❑ The machine instructions that generate the function call and return take time each time a function is called.
- ❑ If there are parameters, even more time overhead is generated.
- ❑ Chances of overflowing the stack in case of static memory allocation.
- ❑ Nested Function Calls may create huge problems in the memory management of the program.

# Memory Management

## Memory of a Program

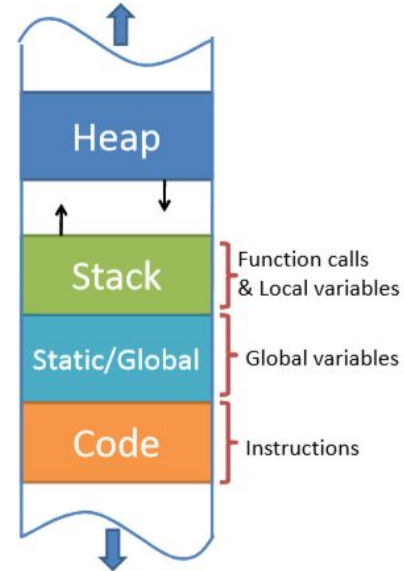
Whenever a program executes, the operating system allocates some space in memory for it.



# Memory Management

## Memory of a Program

The memory allocated to a program is divided into four parts.



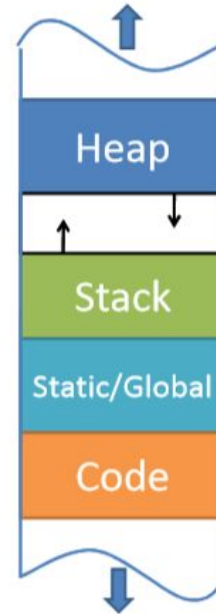
# Memory Management

## Memory of a Program

```
#include<stdio.h>
int total;
int Square(int x)
{
    return(x*x);
}

int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return(z);
}

int main()
{
    int a=4, b=8;
    total = SquareOfSum(a,b);
    printf("Output=%d",total);
}
```



# Memory Management

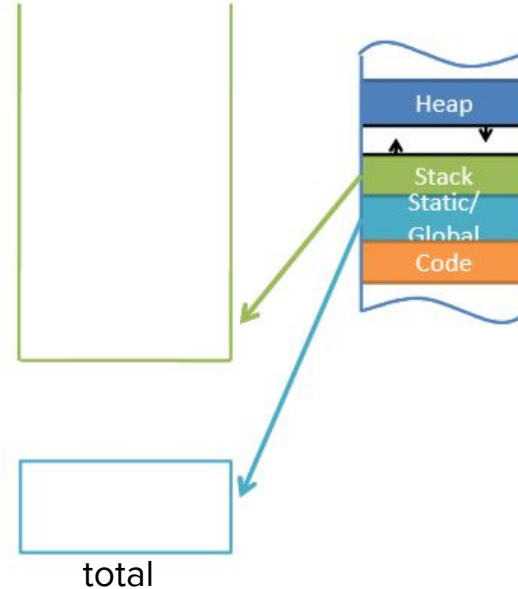
## Use of Stack

```

#include<stdio.h>
int total;
int Square(int x)
{
    return(x*x);
}

int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return(z);
}

int main()
{
    int a=4, b=8;
    total = SquareOfSum(a,b);
    printf("Output=%d",total);
}
    
```



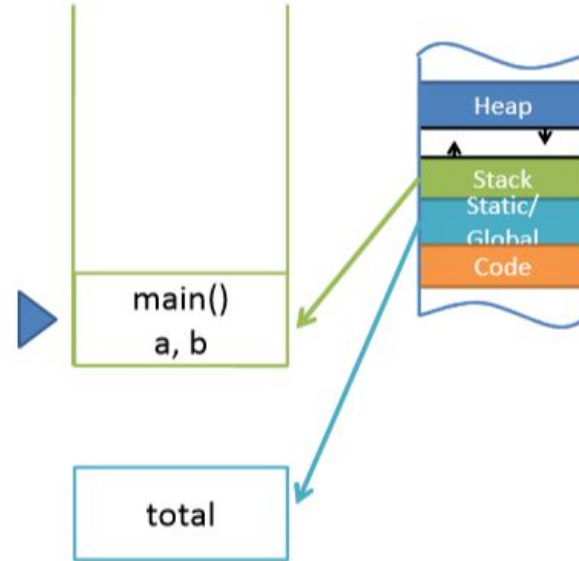
# Memory Management

## Use of Stack

```
#include<stdio.h>
int total;
int Square(int x)
{
    return(x*x);
}

int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return(z);
}

int main()
{
    int a=4, b=8;
    total = SquareOfSum(a,b);
    printf("Output=%d",total);
}
```



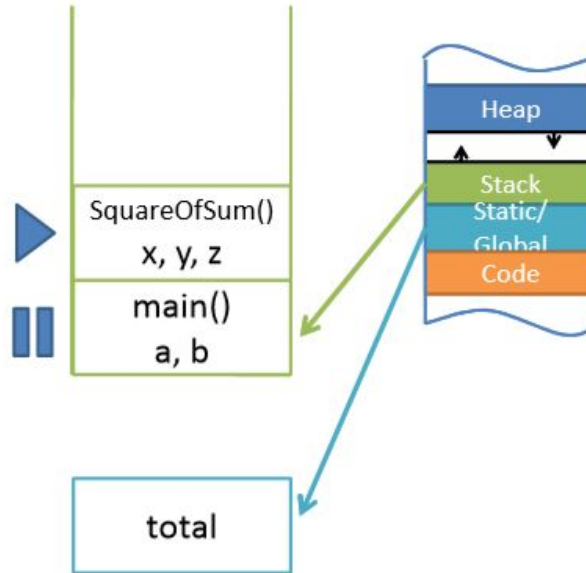
# Memory Management

## Use of Stack

```
#include<stdio.h>
int total;
int Square(int x)
{
    return(x*x);
}

int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return(z);
}

int main()
{
    int a=4, b=8;
    total = SquareOfSum(a,b);
    printf("Output=%d",total);
}
```



# Memory Management

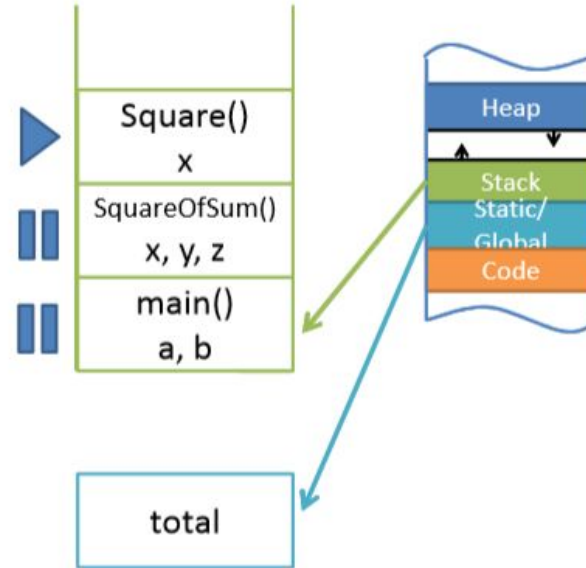
## Use of Stack

```

#include<stdio.h>
int total;
int Square(int x)
{
    return(x*x);
}

int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return(z);
}

int main()
{
    int a=4, b=8;
    total = SquareOfSum(a,b);
    printf("Output=%d",total);
}
    
```





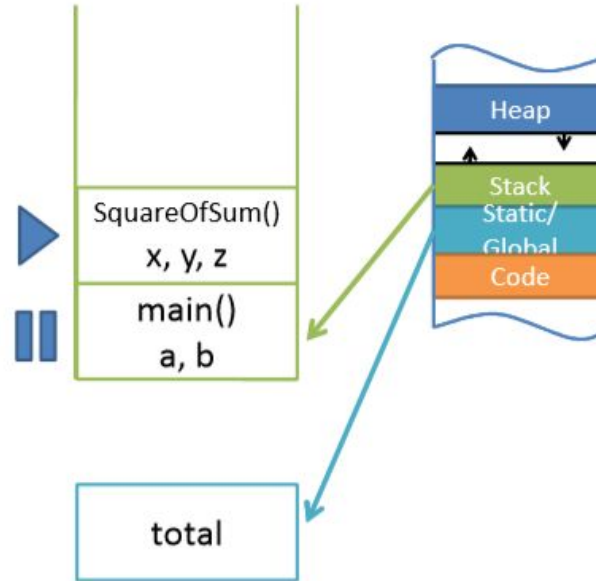
# Memory Management

## Use of Stack

```
#include<stdio.h>
int total;
int Square(int x)
{
    return(x*x);
}

int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return(z);
}

int main()
{
    int a=4, b=8;
    total = SquareOfSum(a,b);
    printf("Output=%d",total);
}
```



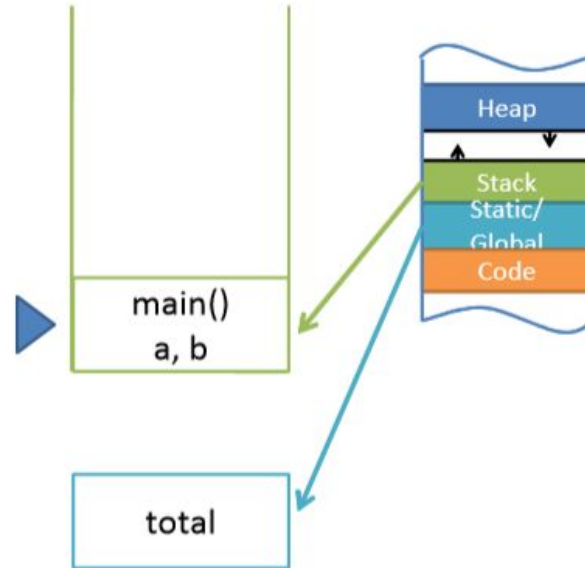
# Memory Management

## Use of Stack

```
#include<stdio.h>
int total;
int Square(int x)
{
    return(x*x);
}

int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return(z);
}

int main()
{
    int a=4, b=8;
    total = SquareOfSum(a,b);
    printf("Output=%d",total);
}
```



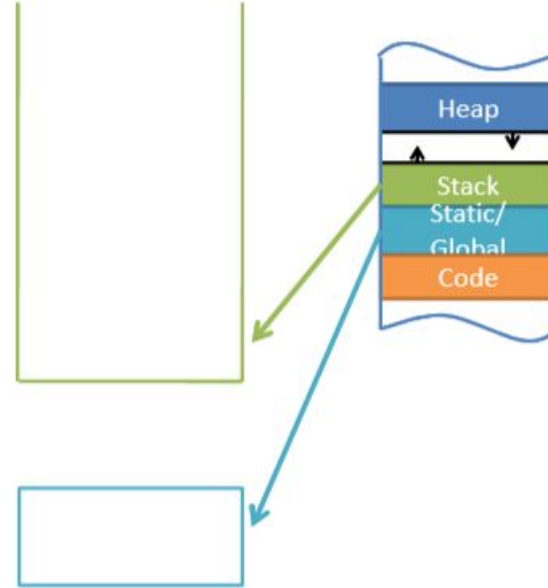
# Memory Management

## Use of Stack

```
#include<stdio.h>
int total;
int Square(int x)
{
    return(x*x);
}

int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return(z);
}

int main()
{
    int a=4, b=8;
    total = SquareOfSum(a,b);
    printf("Output=%d",total);
}
```



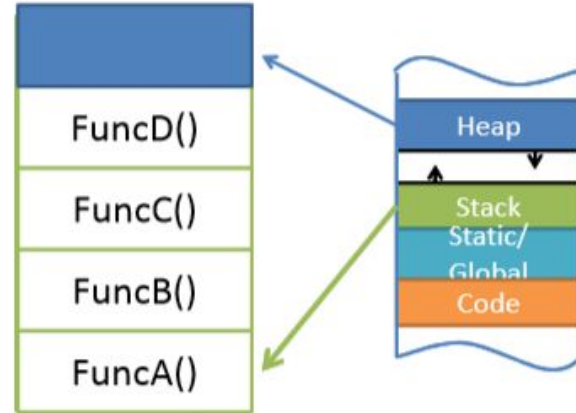
# Memory Management

## Limitation of Stack

```
#include<stdio.h>
int total;
int Square(int x)
{
    return(x*x);
}

int SquareOfSum(int x,int y)
{
    int z = Square(x+y);
    return(z);
}

int main()
{
    int a=4, b=8;
    total = SquareOfSum(a,b);
    printf("Output=%d",total);
}
```



## Stack Overflow

# Limitation of Stack

- ❑ Allocation and deallocation of stack memory are handled by Operating System. Programmer cannot control the lifetime of variables in stack memory.
  - ❑ **Allocation:** Function starts (Push onto stack)
  - ❑ **Deallocation:** Function finishes (Popped out of stack)
- ❑ Size of the stack frame for a function is known at the compile time.
- ❑ An array with unknown size (only known at run time) cannot be allocated in the stack memory.

## Solution

- ❑ Dynamic Memory Allocation (Use the Heap memory)
- ❑ **Inline Function**

# Inline Function

- ❑ Compiler places a copy of the code of that function at each point where the function is called at compile time. Stack operations are not required.
- ❑ To inline a function, place the keyword `inline` before the function name.

```
inline void func1() {  
    cout << "hello world!" << endl;  
}
```

- ❑ All functions defined within a class are inline by default. Defining a member function within the class definition declares the function inline, even if inline specifier is not used.

# Inline Function

```
class A {  
    void display() {  
        cout << "hello" << endl;  
    }  
};
```

**INLINED**  
(As the function is fully  
defined inside the class)

```
class B{  
    void displayB();  
};  
void B::displayB() {  
    cout << "hello" << endl;  
}
```

**NOT INLINED**  
(As only the prototype is  
declared inside the class)

# Inline Function

- ❑ An inline function must be defined before it is first called.
- ❑ In-lining is a request made to the compiler, not a command.
- ❑ There is no guarantee that the function will be inlined. If, for various reasons, the compiler is unable to fulfill the request, the function is compiled as a normal function and the inline request is ignored.
- ❑ When a function is inlined, the code fragment associated with it is placed at the place of function call, so no stack operations.
- ❑ The following types of functions are not inlined by the compiler, even if a request was made:

- Recursive functions.
- Functions with looping constructs.
- Functions using static variables.
- Switch or goto statement.



# Inline Function

---

## Advantages

- ❑ No overhead associated with the function call and return mechanism.
- ❑ Therefore In-line functions can be executed much faster than normal functions.
- ❑ It provides a more structured way to expand short functions in line than macros.
- ❑ An inline function might be able to be optimized more thoroughly by the compiler than a macro expansion.

## Shortcomings

- ❑ If Inline functions are too large and called too often, program grows larger.
- ❑ Therefore only short functions are declared as inline functions.

## Try it

---

Try to figure out if the compiler has made your function inline upon request or not.

You may seek help from [here](#).

# Thanks

# Copy Constructor

Course Code: CSE 205

Course Title: Object Oriented Programming

---

**NAFIZ IMTIAZ KHAN**

Lecturer, Department of CSE, MIST

Email: [nafiz@cse.mist.ac.bd](mailto:nafiz@cse.mist.ac.bd)

# Copy Constructor

- ❑ Used to fully specify exactly what occurs when a copy of an object is made
- ❑ Two different situation when object copying is done
  - ❑ Assignment
  - ❑ Initialization
- ❑ Copy constructor applies to **Initialization**
- ❑ Do not affect assignment operations
- ❑ **By default a bitwise copy is done. This is the default copy constructor**

# Copy Constructor Invocation

---

It is invoked in the following 3 ways --

- ❑ When an object is used to initialize another in a declaration statement
- ❑ When an object is passed as a parameter to a function
- ❑ When a temporary object is created for use as a return value by a function

# Copy Constructor Declaration

Most common form of declaration

```
classname (const classname &ob) {  
    //body  
}
```

```
class Student {  
    char *name;  
public:  
    Student(const Student &ob) {  
        name = new char[strlen(ob.name)];  
        strcpy(name, ob.name);  
    }  
};
```

Here *obj* is a reference to an object that is being used to initialize another object

**Let's see some  
Examples**



# Code covered in class

---

<https://www.onlinegdb.com/RY9JgZUg5J>

# Thanks

# **CSE-205: Object Oriented Programming**

## **Topic: Template and STL**

Nafiz Imtiaz Khan  
Lecturer, Department of CSE, MIST  
Email: [nafiz@cse.mist.ac.bd](mailto:nafiz@cse.mist.ac.bd)

# Template



- ❑ Foundation of generic programming - creating a generic class or a function.
- ❑ C++ library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

## Example:

```
vector <int> myIntVect;  
vector <string> myStringVect;
```

# Function Templates

---

- ❑ Functions that can operate with generic types.
- ❑ Achieved using template parameters.
- ❑ A template parameter is a special kind of parameter that can be used to pass a data type as argument just as we pass values to functions.

## Syntax:

```
template <class type> retType  
funcName(paramList) {  
    // body of function .....  
}
```

# Function Templates

---

## Example:-

```
template <typename T> T Max (T a, T b) {  
    return a < b ? b:a;  
}
```

```
Max <int> (3,4);
```

```
Max <double> (3.3, 5.4);
```

- ❑ Here T is the **template** parameter.
- ❑ It represents a type that has not yet been specified, but that can be used in the function as if it were a regular data type.

# Function Templates

---

- ❑ We are using T as – return type and data type of parameters.
- ❑ When compiler encounters call to a template function, it uses the template to automatically generate a function replacing each appearance of T by the type passed as the actual template parameter (int / double in this case) and then calls it.

# Class Templates

---

- ❑ Templates are a way of making classes more abstract.
- ❑ Possible to define the behavior of the class without actually knowing what data-type will be handled by the operations of the class.
- ❑ In essence, this is what is known as generic programming.
- ❑ Allows classes to have members that use template parameters as types.
- ❑ When defining a function as a member of a templated class, it is necessary to define it as a templated function



# Class Templates

---

**Syntax:-**

```
template class myClass {  
    T id;  
public:  
    myClass(T param) {  
        id = param;  
    }  
};
```

Syntax for declaring objects of templated class is :-

```
myClass <int> myInt;
```

# Class Templates

---

If the func is not defined in-line then the syntax is slightly different.

```
#include <iostream>
using namespace std;
template < class T > class myClass
{
    T id;
public:
    myClass (T param)
    {
        id = param;
    }
    T getVal ();
};

template < class T > T myclass < T >:getVal () {
    return id;
}
```

# Class Templates

---

If a particular data-type for a template function requires a different implementation, then we could define a template specialization

```
#include <iostream>
using namespace std;
template <class T> class myClass {
    T var;
public:
    myClass(T param) : var{param} {}
    T incr() {
        return ++var;
    }
};

template <> class myClass <char> {
    char var;
public:
    myClass(char param) : var{param} {}
    char incr() {
        if (var >= 97&& var <=122) return var -=32;
    }
};
```

# Class Templates

---

For the preceding code fragment :-

- ❑ `template<>` : means for the following templated function , all types are known and no template params are required.
- ❑ Note the use of character specialization parameter after the class name in the second case.
- ❑ The specialization parameter , identifies the data type for which the special class definitions is to be utilized.

We must define all members of the class, including those identical to the generic template class – As in this case the constructor of the class.

# Standard Template Library

---

## C++ STL

- ❑ Set of C++ template classes.
- ❑ Provides general-purpose classes and functions that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.
- ❑ Have a rich set of pre-defined functions.

## Components

- ❑ Containers : Vectors, Lists, etc.
- ❑ Iterators : Allows to step through the elements.
- ❑ Algorithms : Allows sorting, searching, etc of container contents

# Vectors : Standard Library

---

Vector is a template class - replacement for the good old C-style arrays.

## **Advantages :-**

- ❑ Same syntax
- ❑ Automatically handles its own storage requirements as it grows.

## **Syntax :-**

```
#include <vector> // include directive.
```

```
using namespace std;
```

```
vector <int> myVect; // declaration.
```

OR

```
std::vector <int> anotherVect ; // namespace std not  
used.
```

# what is vector `<int> v` ?

---

A template class that will wrap an array of variables of type `int`.

- ❑ Vector will store the variables in a contiguous memory area.
- ❑ Each element can be accessed using `v[0]`, `v[1]`, etc.

## **Problems with usage of array:-**

- ❑ Static array : can't use a variable as it's size.
- ❑ Dynamic array : ensure dealloc of memory in end.
- ❑ Both : when the array grows too big to fit into the initially assigned memory, you have to implement logic, to allocate a new block of memory and then copy all the array elements into this bigger space

# what is vector <int> v ?

---

Vector takes care of all the above problems :-

- ❑ You can use a variable to declare size of a vector.
- ❑ You do not have to deallocate memory after using it.
- ❑ As the number of elements outgrow the original size, the vector will reallocate memory and copy the contents, transparently.
- ❑ Additionally, it will provide a number of useful methods that will help us access and manipulate the members of the vector.



# what is vector <int> v ?

---

**Example :-**

```
int size =10; // or you may take input from user.
vector <int> myVect(size); // var used to declare size.
for(int i=0; i<myVect.size(); ++i){
    myVect[i] = i; // we used the size() method
}
// NO NEED TO FREE MEMORY.
```

# Vectors : Standard Library

---

## Wrapper class :-

- ❑ Vector class overloads – [] operator .
- ❑ Vector class also implements function at( int position)

How does vector handle size issues ?

– C-style arrays ( can only grow upto sizeofArray) :-

```
cin >> sizeofArray;
```

```
int *ptrSizeOfArray = new int[sizeofArray];
```

# Vectors : Standard Library

---

**Vectors (can grow infinitely):** -

```
vector<int> myVect;  
int input = 0;  
while ( input != -1)  
{  
    cin>>input;  
    myVect.push_back(input);  
}
```

# Vectors : Standard Library

---

How does vector handle size issues ?

– Vector class has

- ❑ Control sequence – internal name for the array.
- ❑ Allocated size for storing the array.
- ❑ If the allocated size becomes small, then the vector will allocate larger memory, and copy the contents into the new location.

This means :-

- ❑ Vector can grow infinitely – only limited by system memory.
- ❑ Vector can slow down programs due to this copying action.

# Vectors : Standard Library

---

We can reserve a certain amount of memory when declaring vectors :-

```
vector<int> myVect;  
myVect.reserve(20); // vector elements not initialized.
```

The following syntax, declares and initializes a vector of 10 integers to 0:-

```
vector<int> myVect(10);  
// A vector with 10 elements is created.  
// Integer constructor called 10 times.
```

# Vectors : Standard Library

---

Methods :-

- `myVector.capacity();` // number of elements vector can hold.
- `myVector.size();` // number of elements stored in it.
- `myVector.resize(int sz, int ele);`
  - // if `vector.size() < sz`, it will preserve old elements and blank spaces will be initialised with `ele`'s upto the new `sz`.
  - // if `vector.size() > sz`, rest of elements are discarded.
  - // if vector has capacity `< sz` , it will be realloc to INCR.
  - // if vector has capacity `> sz` , it won't SHRINK!

# Vectors : Standard Library

---

Methods :-

- `myVector.push_back (value);`  
    // adds an element to end of vector, also resizes it if required.
- `myVector.at(index);`  
    // returns the element stored at index position.
- `myVector.size();`  
    // returns the number of elements in the vector, as unsigned int.
- `myVector.clear();` // removes all elements in the vector
- `myVector.empty();` // returns boolean value , if vector is empty.

# Vectors : Standard Library

---

```
myVector.begin();  
    // reads vector from first element.  
    // returns an iterator to the start of the vector.  
myVector.end();  
    // only use to check if vector end is reached.  
    // returns an iterator to the end of the vector.  
myVector.insert(myVector.begin()+integer, newValue);  
    // inserts new value at the specified position.  
myVector.erase(myVector.begin()+integer);  
    // deletes element at specified position.
```



# Vector : Iterator

---

- ❑ Provide a means for accessing data stored in container classes
- ❑ An iterator is used to move through the elements of an STL container (vector, list, set, map, ...) in a similar way to array indexes or pointers.
- ❑ Vectors iterators are usually implemented as pointers.
- ❑ The \* operator dereferences an iterator (ie, is used to access the element an iterator points to) .
- ❑ ++ (and -- for most iterators) increments to the next element.
- ❑ Vectors can also be accessed using subscripts.

## Syntax:

- `class_name<template_parameter>::iterator name ;`

# Vector : Iterator

---

## Example:

```
vector<int>::iterator myIntVectorIterator;
```

- ❑ Different containers support different types of iterator.
- ❑ There are different classes of iterators, each with slightly different properties.

**Eg :** - Iterators for input /reading ; output/writing data in the container; random access iterator (allows maths).

## Accessing elements of a vector :-

- ❑ Call begin function to get an iterator;
- ❑ Use ++ to step through the objects
- ❑ Access each object with the \* operator ("\*iterator") – Stop iterating when the iterator equals the container's end iterator.

# Vector : Iterator

---

**Code :-**

```
vector<int> vec; // enter five elements in the vector.
for(int i = 0; i < 5; i++){
    vec.push_back(i);
}
// get an iterator to step through the vector.
vector::iterator v = vec.begin();
// use iterator to display the contents of vector.
while( v != vec.end()) {
    cout << "value of v = " << *v << endl;
    v++;
}
```

# Map

---

- ❑ Maps are associative containers that store elements in a mapped fashion.
- ❑ Each element has a key value and a mapped value.
- ❑ No two mapped values can have the same key values.

**<Key, Value>**

## **Declaration:**

```
#include <map>
map<int, int> m1;
m1.insert(pair<int, int>(1, 40)); // to insert values to map
m1
```

# Queue

---

- ❑ Queues are a type of container adaptors that operate in a first in first out (FIFO) type of arrangement.
- ❑ Elements are inserted at the back (end) and are deleted from the front.
- ❑ Queues use an encapsulated object of deque or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

## Declaration:

```
queue <int> q1;
```

# Queue

---

Some Functions of Queue:

- ❑ `q1.push(10);` // inserting an element at queue
- ❑ `q1.back();` //Point to the back of the queue
- ❑ `q1.front();` // Point to the front of the queue
- ❑ `q1.pop();` //remove an element from the queue

# List

---

- ❑ Lists are sequence containers that allow non-contiguous memory allocation.
- ❑ As compared to vector, the list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about a doubly linked list. For implementing a singly linked list, we use a forward list.

## Declaration:

```
#include <list>  
list <int> l1;
```

# References

---

- ❑ [The C++ Standard Template Library \(STL\)](#)
- ❑ [Vector in C++ STL](#)
- ❑ [Queue in C++ STL](#)
- ❑ [List in C++ STL](#)
- ❑ [Set in C++ STL](#)
- ❑ [Map in C++ STL](#)
- ❑ [Pair in C++ STL](#)



# Thanks