CET204A Object Oriented Programming

Department of Computer Science and Engineering

CET204A Object Oriented Programming

Teaching Scheme
Theory: 3 Hrs / Week

Credits: 02 + 01
Practical: 2Hrs/Week

- **Course Objectives:**

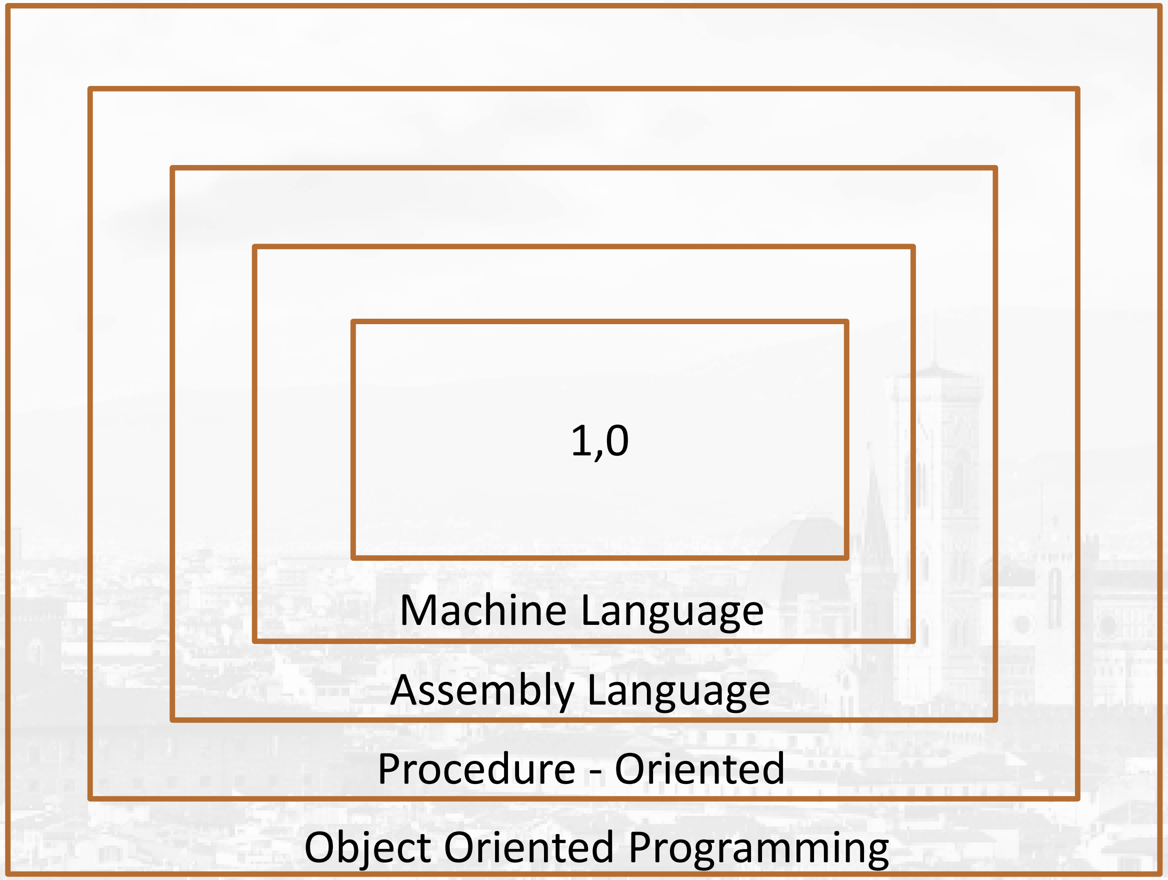
1) Understand basic concepts of Object Oriented Programming.
2) Learn Inheritance, Polymorphism and Exception Handling features of Object Oriented Programming
3) Study concepts of Standard Template Library

- **Course Outcomes:**

1) Apply the basic concepts of Object Oriented Programming in application development.
2) Design and develop real world applications using inheritance, Polymorphism and Exception Handling features.
3) Explore and use Standard Template Library to simplify programming.

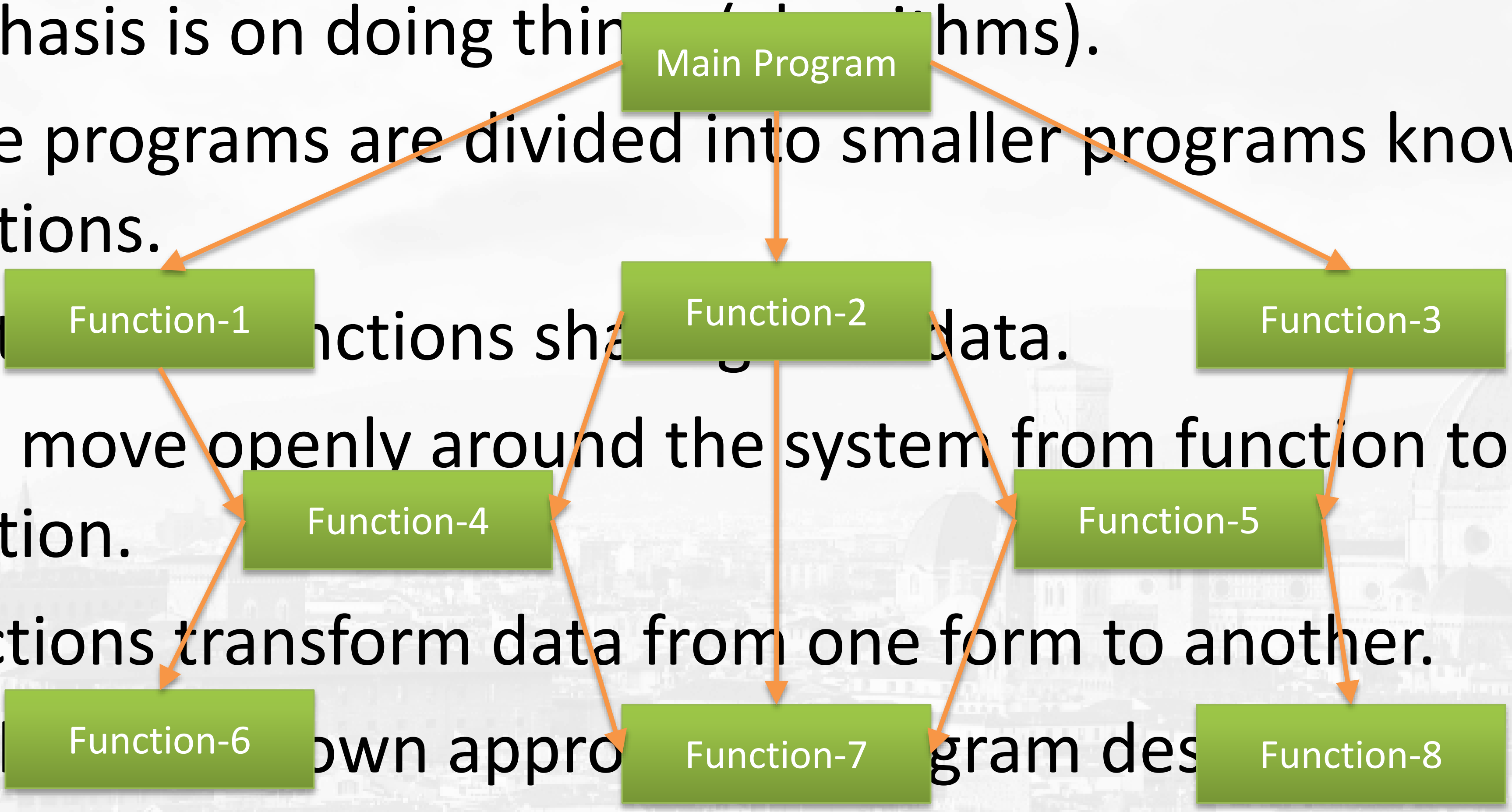
Fundamentals of OOP

Software Evolution



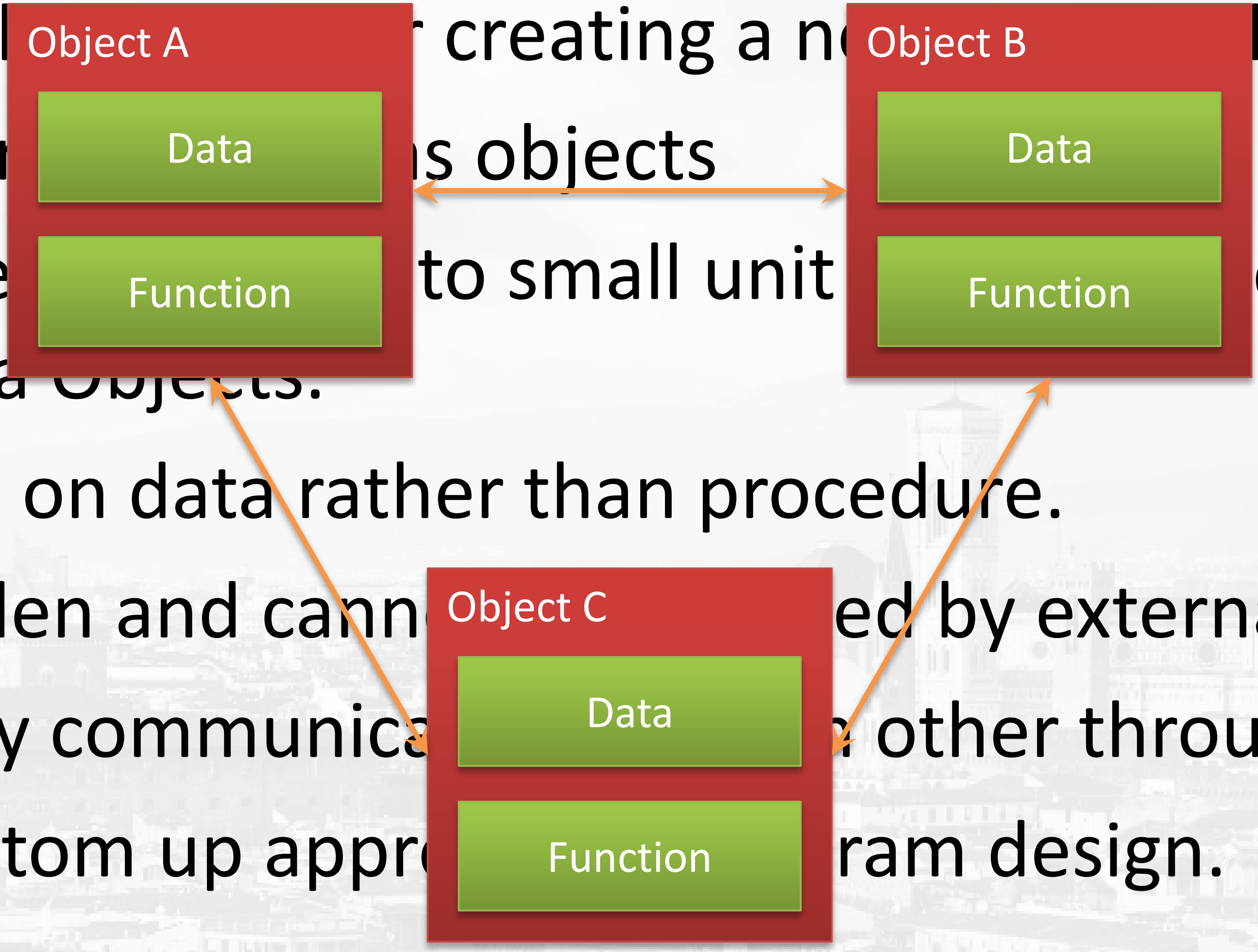
Procedural Programming

- Emphasis is on doing thin[illegible]hms).
- Large programs are divided into smaller programs known as functions.
- Most [illegible]nctions sha[illegible]data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Empl[illegible]own appro[illegible]gram des[illegible]



Object Oriented Programming

- Design meth… r creating a n… lication
- Works on en… s objects
- Decompose … to small unit … ch are accessed via … .
- Emphasis is on data rather than procedure.
- Data is hidden and cann… ed by external function.
- Objects may communica… other through function.
- Follows bottom up appr… ram design.

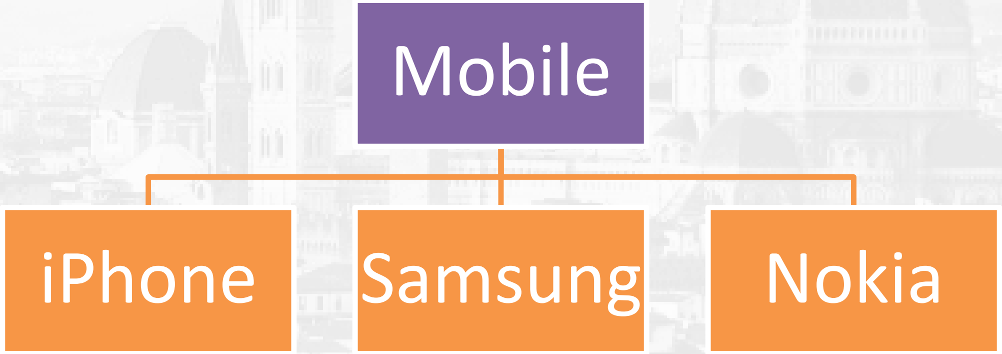


Features of OOP

- Programming language with OOP support has to fulfill these features
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

Real life Example

- Mobile as an object was designed to provide basic functionality as
  - Calling and Receiving calls
  - Messaging
- Thousands of new features and models are getting added



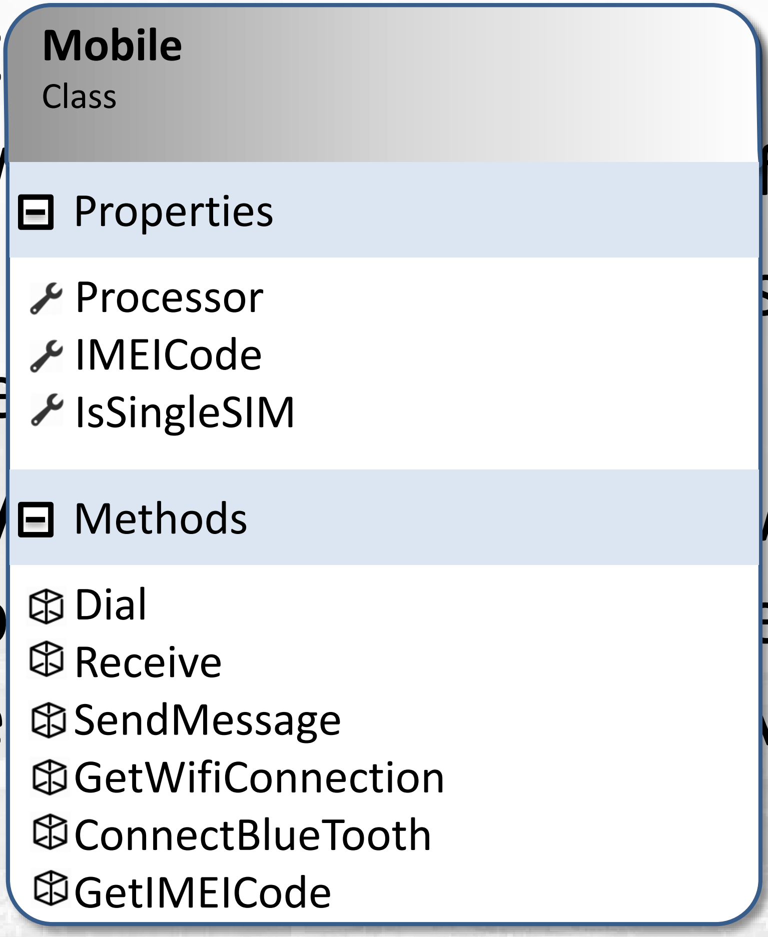
Objects

- Any real world entity which can have some characteristics or which can perform some work is called as Object.
  - This object is also called as an instance i.e. - a copy of an entity in programming language.
- A mobile manufacturing company, at a time manufactures lacs of pieces of each model which are actually an ***instance***.
- These objects are differentiated from each other via some identity (e.g. IMEI number) or its characteristics.

```
Mobile mbl1 = new Mobile ();
Mobile mbl2 = new Mobile ();
```

Cla

- A C ... scribe
  - – W ... f how t
- Ma ... sist of ope
- A M ... hich h Typ ... essor, a like ... Messag




```
class Mobile
{
    private:
        string IMEICode, SIMCard, Processor;
        int InternalMemory;
        bool IsSingleSIM;
    public:
        void GetIMEICode()  {
            cout << "IMEI Code - IEDF34343435235";
        }
        void Dial() {
            cout << "Dial a number";
        }
        void Receive() {
            cout << "Receive a call";
        }
        virtual void SendMessage(){
            cout << "Message Sent";
        }
}
```

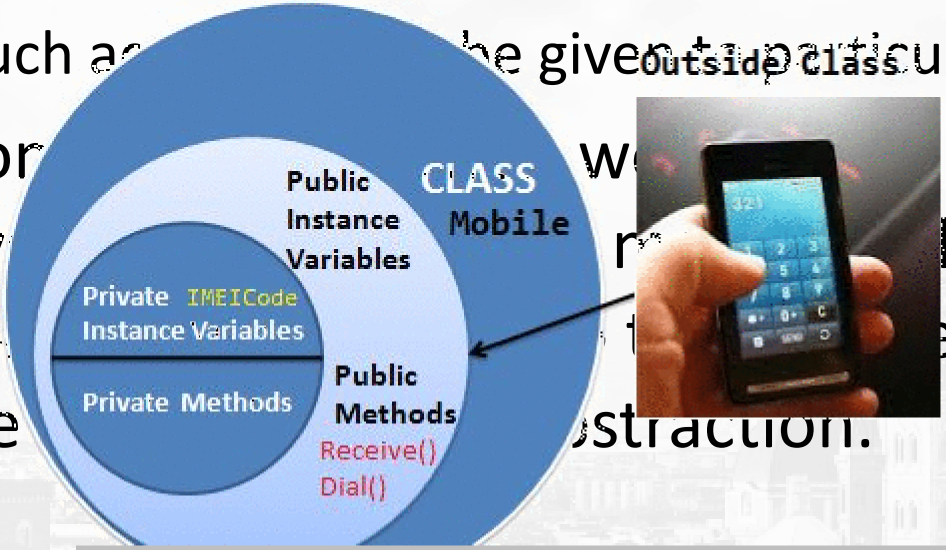
Abstraction

- Abstraction - only show relevant details and rest all hide it
  - its most important pillar in OOPS as it is providing us the technique to hide irrelevant details from User
- Dialing a number calls some method internally which concatenate the numbers and displays it on screen but what is it doing we don’t know.
- Clicking on green button actual send signals to calling person’s mobile but we are unaware of how it is doing.

```
void Dial()
{
   //Write the logic
   cout << "Dial a number";
}
```

Encapsulation

- Encapsulation is defined as the process of enclosing one or more details from outside world through access right.
  - It says how much a... be given to particular details.
- Both Abstraction ... we ... in hand because Abstraction say... ... le & Encapsulation provides the le... ... e details. i.e. – It implements the ... straction.




```
private:
    string IMEICode = "76567556757656";
```

Polymorphism

```
class Samsumg :public  Mobile
{
   public:
      void GetWIFIConnection()  {
            cout<<"WIFI connected";
      }
         //This is one method which shows camera functionality
       void CameraClick()  {
             cout<<“Camera clicked";
       }
          //overloaded method which shows camera functionality as well but with panaroma mode
       void CameraClick(string CameraMode) {
             cout<<"Camera clicked in " + CameraMode + " Mode";
       }
}
```

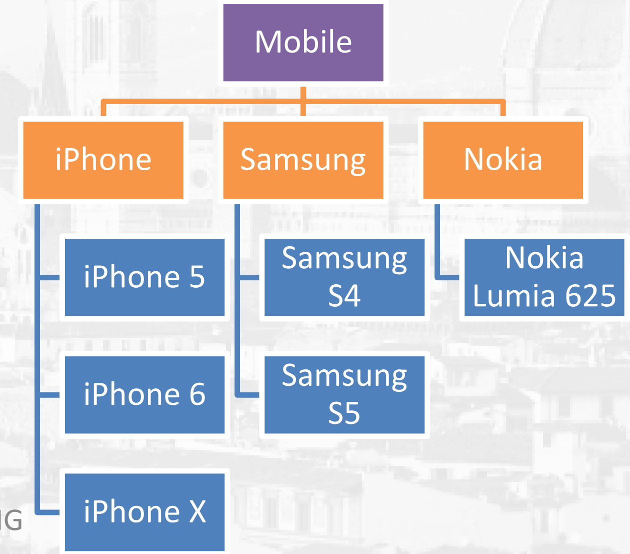
Polymorphism

- Dy
- Me
  pol
- By
  fro
  abi

```
class Nokia : public Mobile
{
   public :
      void GetBlueToothConnection()  {
            cout<<"Bluetooth connected";
       }
       //This is runtime polymorphism
       void SendMessage()  {
            cout<<"Message Sent to a group";
        }
}
```

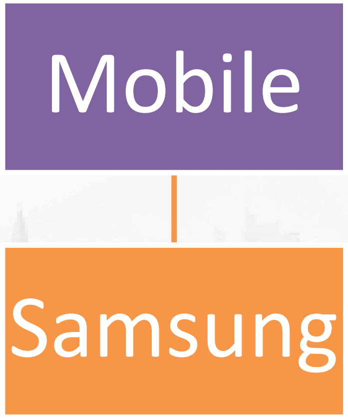
Inheritance

- Ability to extend the functionality from base entity to new entity belonging to same group.
  - This will help us to reuse the functionality which is defined before.
- There are mainly 4 types of inheritance:
  - Single level inheritance
  - Multi-level inheritance
  - Hierarchical inheritance
  - Hybrid inheritance
  - Multiple inheritance

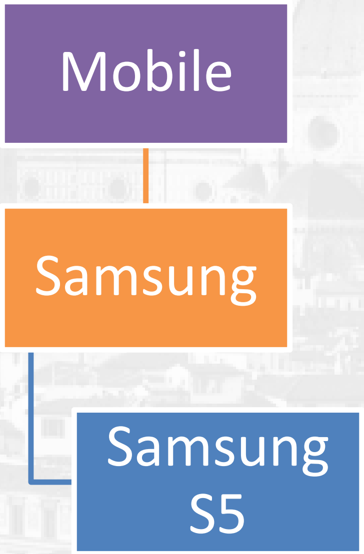


Inheritance

- Single level inheritance
  - Single base class & a single derived class i.e. - A base mobile features are extended by Samsung brand.

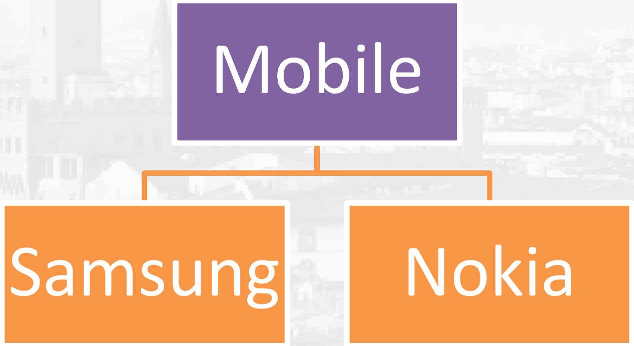


- Multi level inheritance
  - In Multilevel inheritance, there is more than one single level of derivation.
  - E.g. After base features are extended by Samsung brand, a new model is launched with latest Android OS

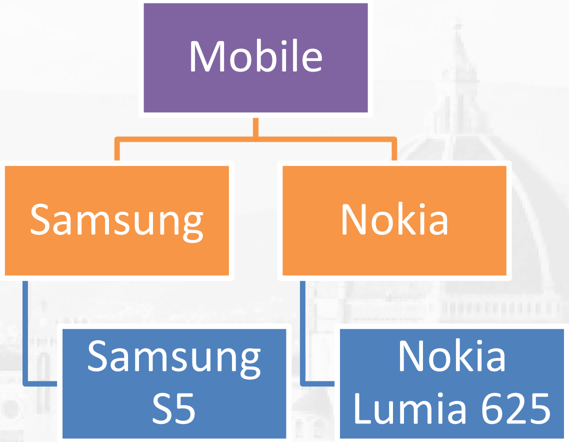


Inheritance

- Hierarchical inheritance
  - Multiple derived class would be extended from base class
  - It's similar to single level inheritance but this time along with Samsung, Nokia is also taking part in inheritance.

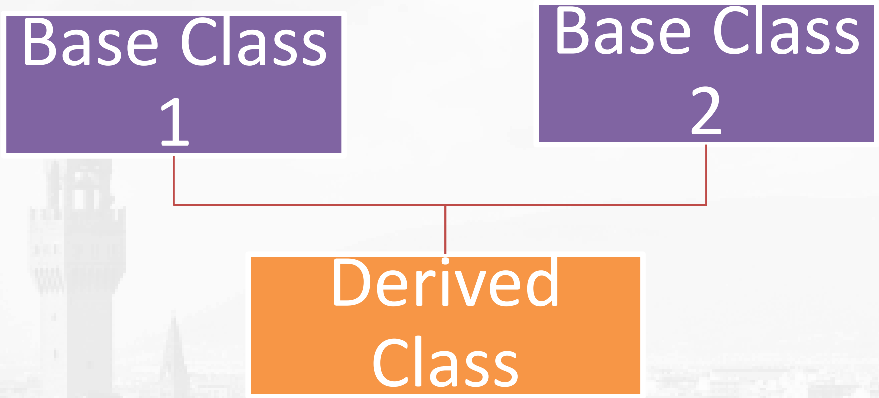


- Hybrid inheritance
  - Single, Multilevel, & hierarchal inheritance all together construct a hybrid inheritance.



Inheritance

- Derived class from multiple base classes.



Object based Vs Oriented Language

- **objects-based** programming are languages that support programming with objects
- Feature that are required for object based programming are:
  - Data encapsulation
  - Data hiding and access mechanisms
  - Automatic initialization and clear-up of objects
  - Operator overloading
- e.g. Visual Basic

- **Object-oriented** programming language incorporates two additional features, namely, inheritance and dynamic binding
- Feature that are required for object based programming are:
  - Data encapsulation
  - Data hiding and access mechanisms
  - Automatic initialization and clear-up of objects
  - Operator overloading
  - Inheritance
  - dynamic binding

Applications of OOP

- Real-business system are often complex and contain many objects with complicated attributes and methods.
- Some of the areas of application of OOPs are:
  - Real-time system
  - Simulation and modeling
  - Object-oriented data bases
  - Hypertext, Hypermedia, and expertext
  - AI and expert systems
  - Neural networks and parallel programming
  - Decision support and office automation systems
  - CIM/CAM/CAD systems

Why C++ ?

- C++ is a versatile language for handling very large programs including editors, compilers, databases, communication systems and any complex real life applications systems
  - C++ allows create hierarchy related objects to build special object-oriented libraries which can be used later by many programmers.
  - the C part of C++ gives the language the ability to get closed to the machine-level details.
  - C++ programs are easily maintainable and expandable - it is very easy to add to the existing structure of an object.
  - It is expected that C++ will replace C as a general-purpose language in the near future.

BASICS OF C++

Simple C++ Program

```
// Simple C++ program to display "Hello World"
// Header file for input output functions
#include<iostream>

using namespace std;

// main function - where the execution of program begins
int main()
{
   // prints hello world
   cout<<"Hello World";

   return 0;
}
```

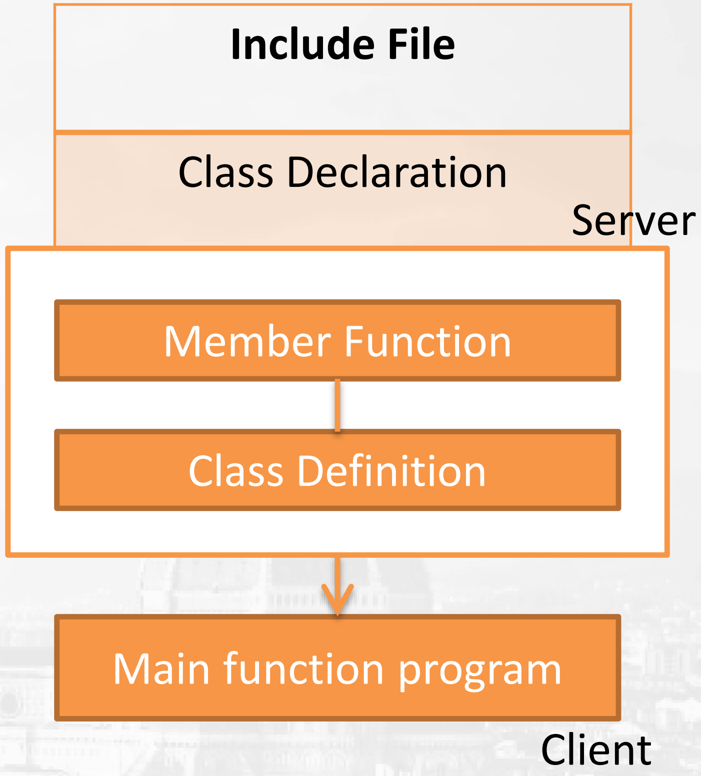
instructs the compiler to include the contents of the file enclosed within angular brackets into the source file.

defines a scope for the identifiers that are used in a program

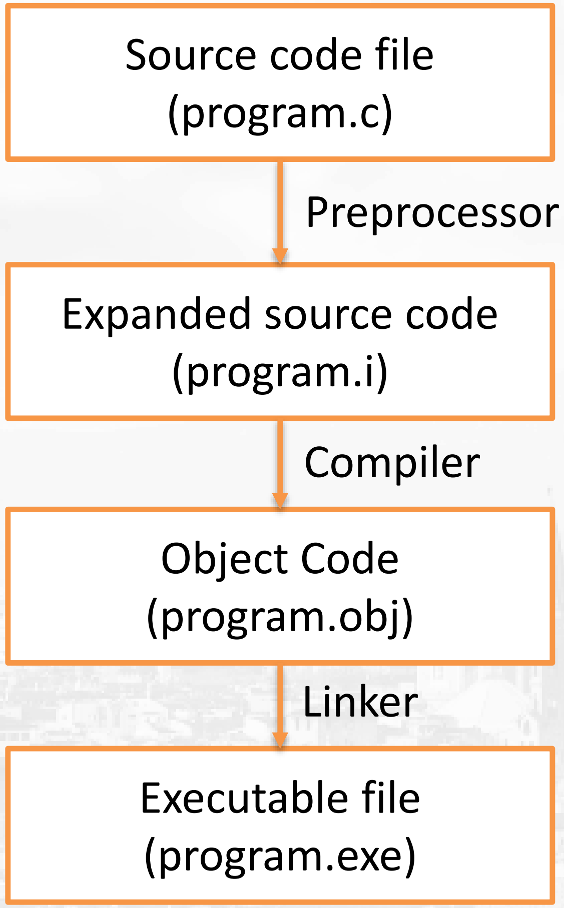
every main() returns an integer value to operating system and therefore it should end with return (0) statement

Structure of C++ Program

- Typical C++ program contains four sections
- It is a common practice to organize a program into three separate files
- The class declarations are placed in a header file and the definitions of member functions go into another file.
- The main program that uses the class is placed in a third file which "**includes**" the previous two files as well as any other file required



C/C++ Compilation and Linking



FUNCTION in C++

- Function is a collection of declarations and statements
- A function must be defined prior to it's use in the program

```
Type name_of_the_function (argument list)
{
	//body of the function
}
```

C++ Function Defination

- C++ function is defined in two steps (preferably but not mandatory)
  - Step #1 – declare the *function signature* in either a header file (.h file) or before the main function of the program
  - Step #2 – Implement the function in either an implementation file (.cpp) or after the main function

The Syntactic Structure of a C++ Function

- A C++ function consists of two parts
  - The function header, and
  - The function body
- The function header has the following syntax

```
<return value> <name> (<parameter list>)
```

- The function body is simply a C++ code enclosed between { }

Example of User-defined C++ Function

```
double computeTax(double income)
{
    if (income < 5000.0) return 0.0;
    double taxes = 0.07 * (income-5000.0);
    return taxes;
}
```

Example of User-defined C++ Function

Function header

```
double computeTax(double income)
{
    if (income < 5000.0) return 0.0;
    double taxes = 0.07 * (income-5000.0);
    return taxes;
}
```

Example of User-defined C++ Function

Function header

Function body

```
double computeTax(double income)
{
    if (income < 5000.0) return 0.0;
    double taxes = 0.07 * (income-5000.0);
    return taxes;
}
```

Function Signature

- The function signature is actually similar to the function header except in two aspects:
  - The parameters' names may not be specified in the function signature
  - The function signature must be ended by a semicolon
- Example

Unnamed Parameter

Semicolon
;

```
double computeTaxes(double) ;
```

Why Do We Need Function Signature?

- For Information Hiding
  - If you want to create your own library and share it with your customers without letting them know the implementation details, you should declare all the function signatures in a header (.h) file and distribute the binary code of the implementation file
- For Function Abstraction
  - By only sharing the function signatures, we have the liberty to change the implementation details from time to time to
    - Improve function performance
    - make the customers focus on the purpose of the function, not its implementation

Example

```
#include <iostream>
#include <string>
using namespace std;
// Function Signature
  double getIncome(string);
  double computeTaxes(double);
  void printTaxes(double);
void main()
{
    // Get the income;
    double income = getIncome("Please enter the employee
    income: ");
    // Compute Taxes
    double taxes = computeTaxes(income);
    // Print employee taxes
    printTaxes(taxes);
}
```

```
double computeTaxes(double income){
    if (income<5000) return 0.0;
    return 0.07*(income-5000.0);
}
double getIncome(string prompt){
    cout << prompt;
    double income;
    cin >> income;
    return income;
}
void printTaxes(double taxes){
    cout << "The taxes is $" << taxes << endl;
}
```

Default Arguments in Function

Case 1: No argument passed

```
void temp (int = 10, float = 8.8);
int main() {
     temp();
}
void temp(int i, float f) {
     … … …
}
```

Case 2: First argument passed

```
void temp (int = 10, float = 8.8);
int main() {
     temp(6);
}
void temp(int i, float f) {
     … … …
}
```

Case 3: All arguments passed

```
void temp (int = 10, float = 8.8);
int main() {
     temp(6, -2.3);
}
void temp(int i, float f) {
     … … …
}
```

Case 4: Second argument passed

```
void temp (int = 10, float = 8.8);
int main() {
     temp(3.4);
}
void temp(int i, float f) {
     … … …
}
```

i = 3, f=8.8

Because, only the second argument cannot be passed. The parameter will be passed as the first argument.

```
// C++ Program to demonstrate working of default argument
 #include <iostream>
 using namespace std;
void display(char = '*', int = 1);
int main()
 {
          cout << "No argument passed:\n";
           display();
          cout << "\nFirst argument passed:\n";
           display('#');
           cout << "\nBoth argument passed:\n";
          display('$', 5);
          return 0;
 }
void display(char c, int n) {
           for(int i = 1; i <= n; ++i) {
           cout << c;
           }
 cout << endl;
 }
```

Reference Variables

- A reference is an *alias*, or an *alternate name* to an existing
  - Contains the address of a variable (like a pointer)

```
int x = 5;
int &z = x;          // z is another
name for x
```

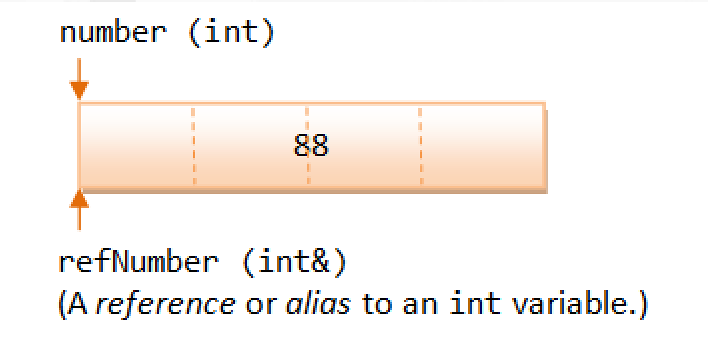
  - No need to perform any dereferencing (unlike a pointer)
  - Must be initialized when it is declared

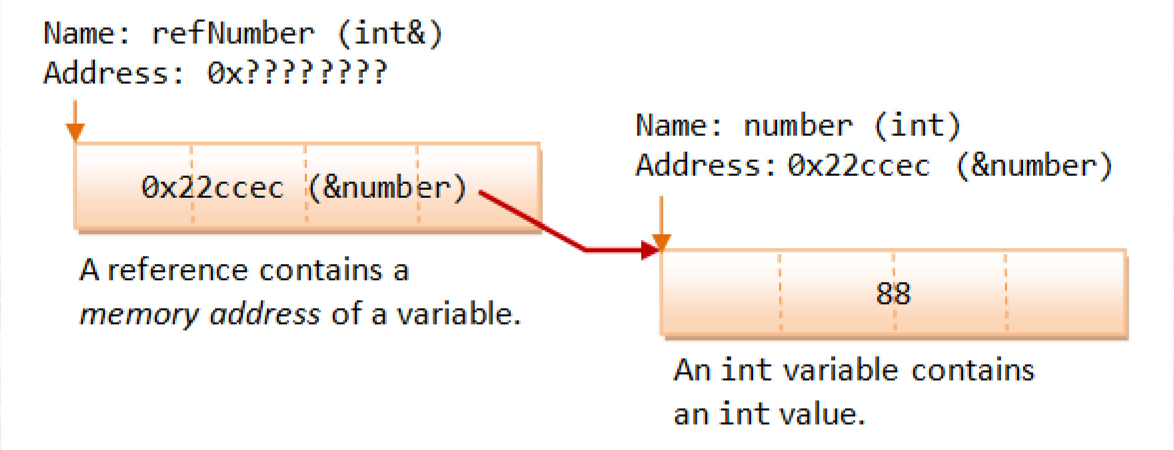
```
int &y ;          //Error: reference must be initialized
```

- References acts as function formal parameters to support pass-by-reference
- Any changes to reference variable inside the function are reflected outside the function

How References Work?

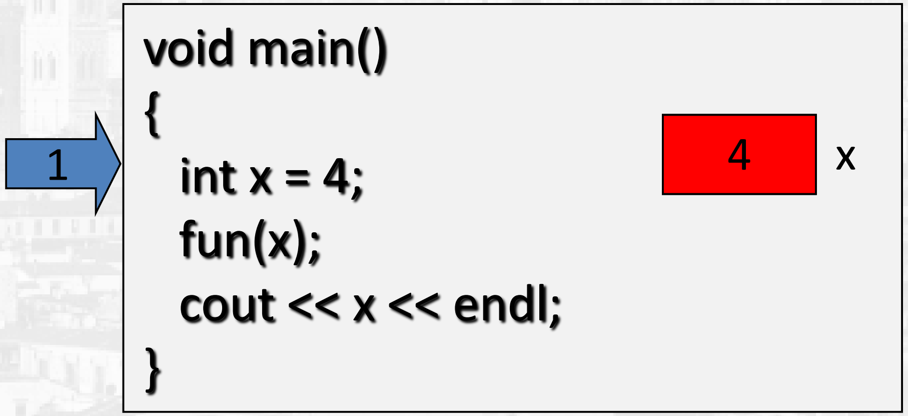
```
type &newName = existingName;

int number = 88; // Declare an int variable called number
int & refNumber = number; // Declare a reference (alias)
```





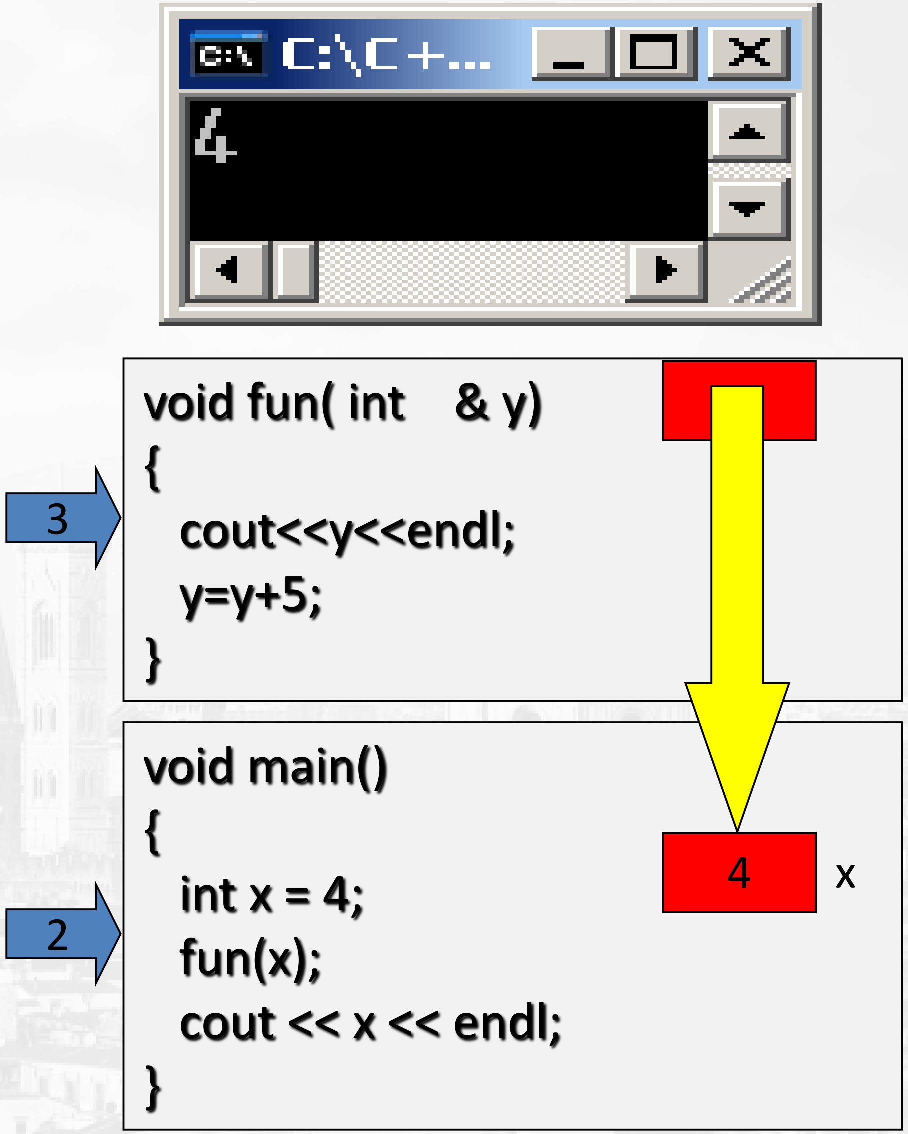
Example of Reference Parameters

```
#include <iostream.h>
void fun(int &y)
{
   cout << y << endl;
   y=y+5;
}
void main()
{
   int x = 4; // Local variable
   fun(x);
   cout << x << endl;
}
```



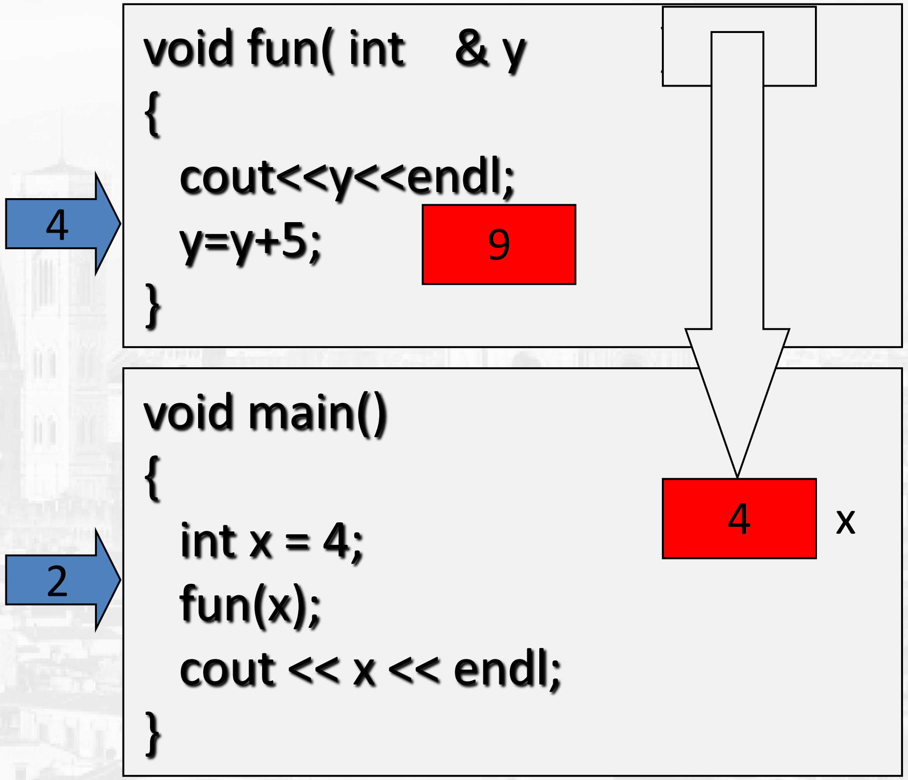
Example of Reference Parameters

```
#include <iostream.h>
void fun(int &y)
{
    cout << y << endl;
     y=y+5;
}
void main()
{
    int x = 4; // Local variable
    fun(x);
    cout << x << endl;
}
```



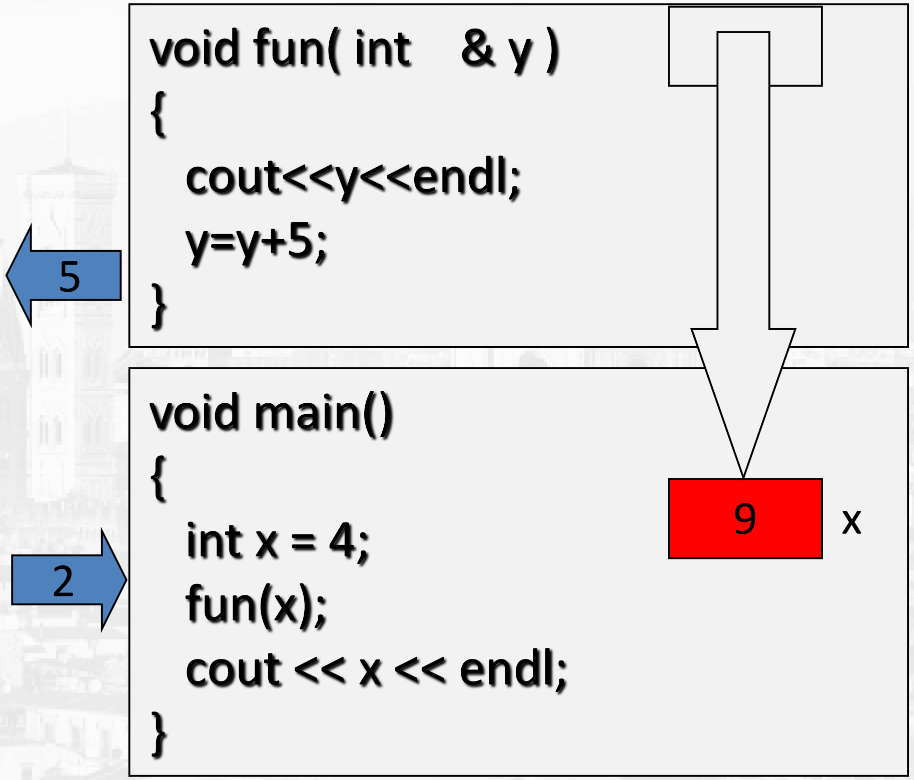
Example of Reference Parameters

```
#include <iostream.h>
void fun(int &y)
{
    cout << y << endl;
     y=y+5;
}
void main()
{
    int x = 4; // Local variable
    fun(x);
    cout << x << endl;
}
```



Example of Reference Parameters

```
#include <iostream.h>
void fun(int &y)
{
    cout << y << endl;
     y=y+5;
}
void main()
{
    int x = 4; // Local variable
    fun(x);
    cout << x << endl;
}
```



Classes and Objects

Class

- A way to bind data and associated function together
- An expanded concept of a data structure, instead of holding only data, it can hold both data and function.
- The data is to be hidden from external use.

Class

keyword

```
class class_name
{
        private:
        variable declaration;
        public:
        function declaration;
        ….
};
```

body

data member

member function

access specifiers

Access Specifiers

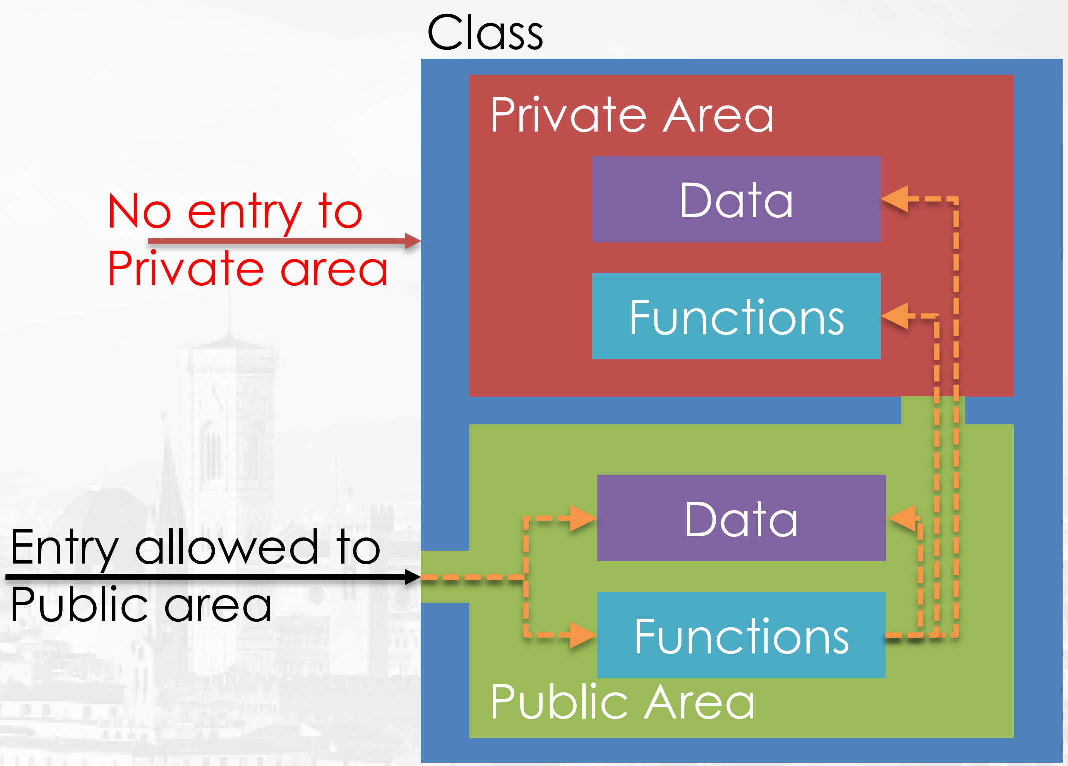
Less Restrictive

public
- **Accessible from own class**
- **Accessible from derived class**
- **Accessible from class objects**

protected
- **Accessible from own class**
- **Accessible from derived class**

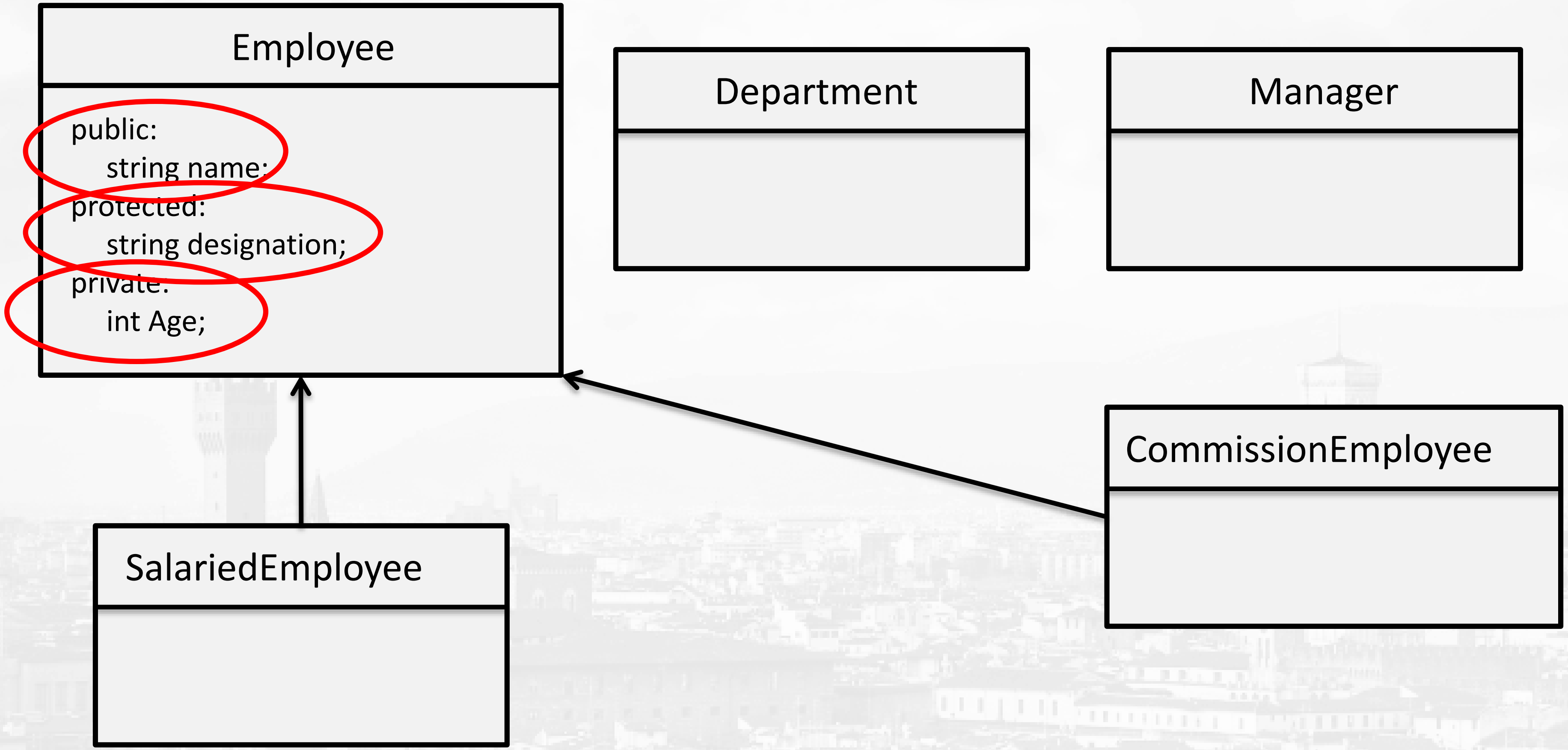
private
- **Accessible from own class**

More Restrictive



* By default all items defined in the class are private

Access Specifiers



Access Specifiers

```
class Person {
    public:                              //access control
        string firstName;      //these data members
        string lastName;       //can be accessed
        tm dateOfBirth;        //from anywhere
    private:
        string address;   // can be accessed inside the class
        long int insuranceNumber;   //and by friend classes/functions
    protected:
        string phoneNumber; // can be accessed inside this class,
        int salary;     // by friend functions/classes and derived classes

};
```

Objects

- A class provides the blueprints for objects, so basically an object is created from a class.
- Objects o ne sort of declaratic

```
#include <iostream>
using namespace std;
class Box {
        public:
        double length; // Length of a box
        double breadth; // Breadth of a box
        double height; // Height of a box
};

int main() {
       Box Box1; // Declare Box1 of type Box
       Box Box2; // Declare Box2 of type Box
       double volume = 0.0; // Store the volume of a box here
}
```

Class

Public data members

Objects of Box class

Objects

- The objects of class will have their own copy of data members.
- The public data members of objects of a class can be accessed using the direct member access operator (.)

```
int main() {
    Box Box1; // Declare Box1 of type Box
    Box Box2; // Declare Box2 of type Box
    double volume = 0.0; // Store the volume of a box here

    // box 1 specification
   Box1.height = 5.0;
   Box1.length = 6.0;
   Box1.breadth = 7.0;

   // box 2 specification
   Box2.height = 10.0;
   Box2.length = 12.0;
   Box2.breadth = 13.0;

    // volume of box 1
    volume = Box1.height * Box1.length * Box1.breadth;
    cout << "Volume of Box1 : " << volume <<endl;

    // volume of box 2
    volume = Box2.height * Box2.length * Box2.breadth;
    cout << "Volume of Box2 : " << volume <<endl;
    return 0;
}
```

Access data members of **Box1** object

Access data members of **Box2** object

Member Functions

- Can be defined inside class

```
return_type  function_name  (parameters)
{
        // function body
}
```

- Or outside the class

```
return_type  class_name::function_name (forma
{
        // function body
}
```

- Functions defined inside class are treated as inline functions by compiler

```
class Box {
  public:
    double length, breadth, height;
    double getVolume() {    //  Returns box volume
        return length * breadth * height;
    }
    double getSurfaceArea();     // returns surface area
};
// member function definition
double Box::getSurfaceArea() {
      ....
}
int main() {
      Box Box1;
      Box1.length = 10;
      Box1.height = 20;
      Box1.breadth=30;
      cout << "Volume of box: " << Box1.getVolume() << endl;
      cout << "Surface Area of box: " << Box1.getSurfaceArea() <<
endl;
}
```

Inline function

member function declaration

member function definition outside class

accessing member functions

Array of Objects

```
#include <iostream>
#include <string>
using namespace std;
class Student
{               string name;
                int marks;
                public:
                void  getdata()
                {
                                cout<<"enter name";
                                cin>> name;
                                cout <<"enter marks";
                                cin>>marks;
                }
                void putdata()
                {               cout << "Name : " << name << endl;
                                cout << "Marks : " << marks << endl;
                }
};
```

```
int main()
{               Student st[5];
                for( int i=0; i<5; i++ )
                {
                cout << "Student " << i + 1 << endl;
                st[i].getdata();
                }
                for( int i=0; i<5; i++ )
                {
                cout << "Student " << i + 1 << endl;
                st[i].putdata();
                }
                return 0;
}
```

Static and non-static variable

```
#include <iostream>
Using namespace std;
 void foo()
 {
for( int i=0; i<5; ++i )
{
 static int staticVariable = 0;
 int local = 0;
 ++local;
 ++staticVariable;
 cout << local << "\t" << staticVariable << "\n";
 }
 } int main()
 {
 foo();
return 0;
}
```

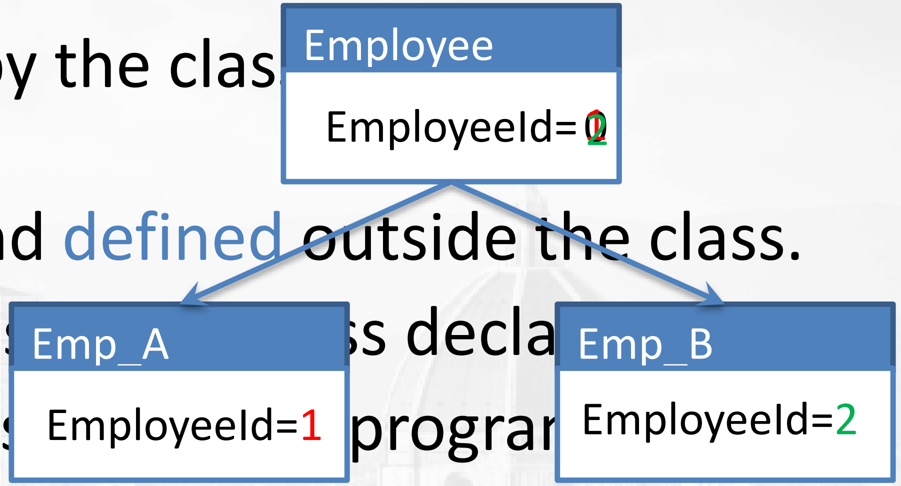
- **Results:**
- 1 1
  1 2
  1 3
  1 4
  1 5

```cpp
class Employee {
  static int EmployeeId;
  public:
    int getEmpId (void) {
        return ++EmployeeId;
    }
    void addEmployee(string);
};
void Employee::addEmployee(string name) {
    int newId = getEmpId();
    cout << "Added New Empl" <<name  <<"with Id: "<<
newId <<endl;
}
int Employee::EmployeeId;
int main() {
     Employee Emp_A, Emp_B;
     Emp_A.addEmployee("Amit");
     Emp_B.addEmployee("Bijoy");
     return 0;
}
```

```
# a.out
Added New Empl Amit with Id: 1
Added New Empl Bijoy with Id: 2
```

Members

- ared by all objects are known as
- ained by the clas
- class and defined outside the class.
- iable ins ss decla
- fetime is progra
- aintain values common to the



Dynamic memory allocation

- Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer.
- Dynamically allocated memory is allocated on **Heap** and non-static and local variables get memory allocated on **Stack**
- Memory in C++ program is divided into two parts –
  - **The stack** – All variables declared inside the function will take up memory from the stack.
  - **The heap** – This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Applications of Dynamic memory allocation

- To allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
- The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don’t need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc

new and delete Operators

- The new operator denotes a request for memory allocation on the Heap

```
pointer-variable = new data-type;
// Pointer initialized with NULL
// Then request memory for the variable
int *p = NULL;
p = new int;
                                OR
// Combine declaration of pointer and their assignment

int *p = new int;
```

Initialize memory:

pointer-variable = **new** data-type(value);

Example:

```
int *p = new int(25);
float *q = new float(75.25);
```

Pointers and Dynamic Memory

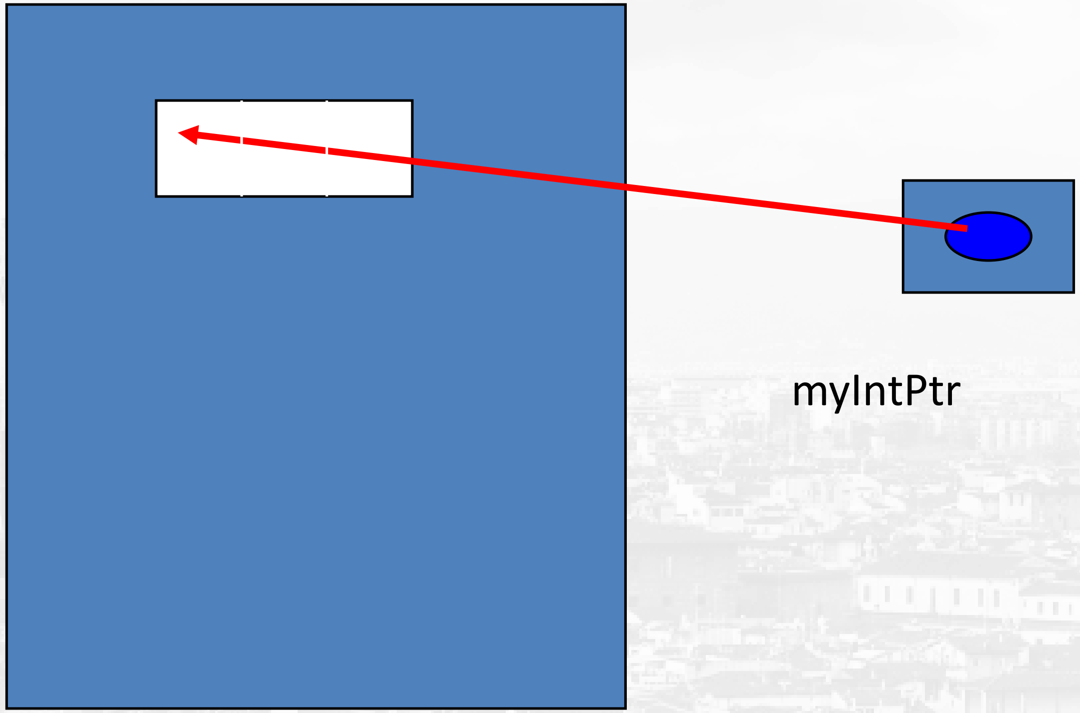
Here are the steps:

```
int * myIntPtr;        // create an integer pointer variable
myIntPtr = new int;         // create a dynamic variable  of the size integer
```

new returns a pointer (or memory address) to the location where the data is to be stored.

Pointers and Dynamic Memory

Free Store (heap)

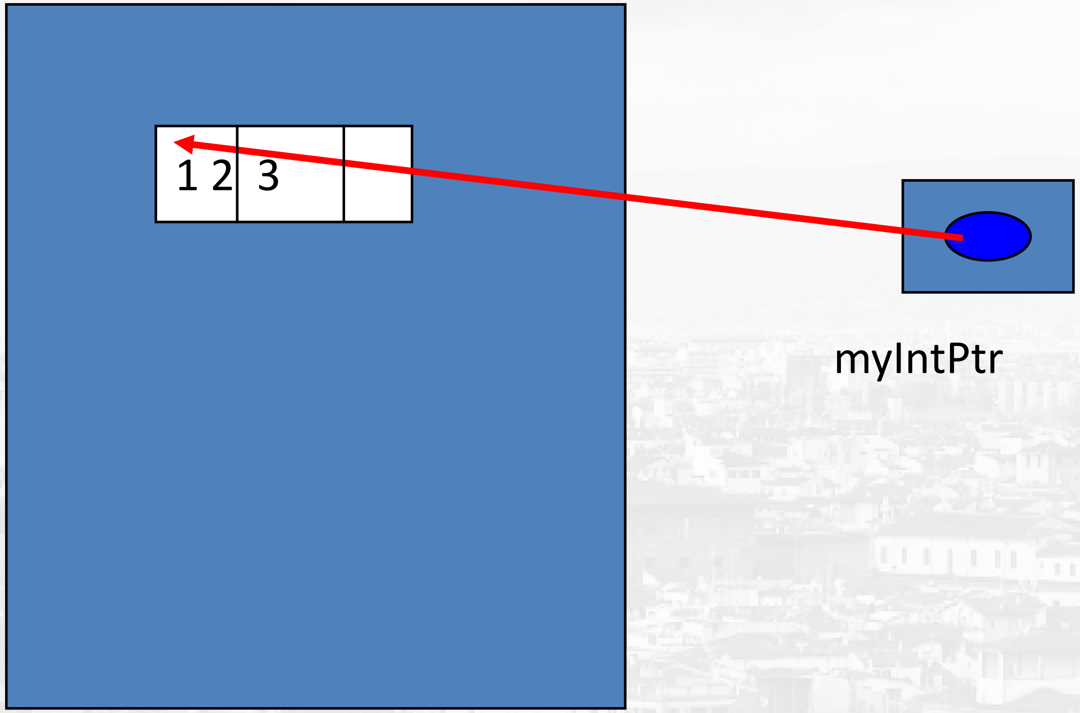



```
int * myIntPtr
myIntPtr = new int;
```

Pointers and Dynamic Memory

Use pointer variable

Free Store (heap)



int * myIntPtr
myIntPtr = new int;

*myIntPtr = 123;

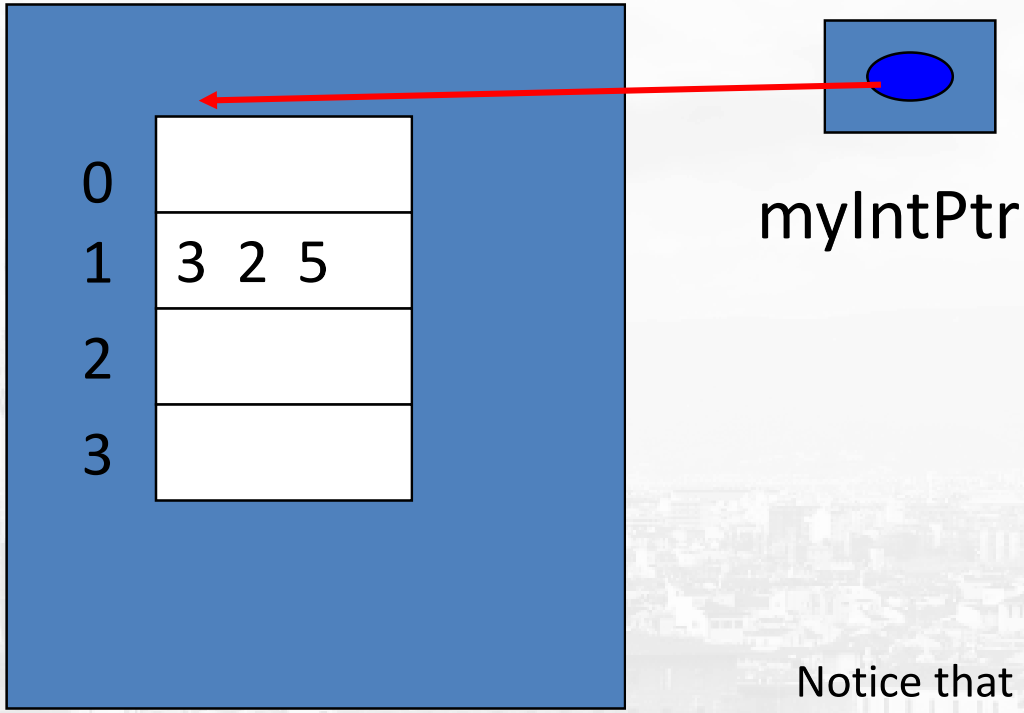
Pointers and Dynamic Memory

- We can also allocate entire arrays with the new operator. These are called dynamic arrays.

- This allows a program to ask for just the amount of memory space it needs at run time.

Pointers and Dynamic Memory

Free Store (heap)




```
int * myIntPtr;
myIntPtr = new int[4];
```

```
myIntPtr[1]= 325;
```

Notice that the pointer symbol is understood,
no * is used to reference the array element.

Pointers and Dynamic Memory

The *new* operator gets memory from the free store (heap).

When you are done using a memory location, it is your responsibility to return the space to the free store. This is done with the *delete* operator.

```
delete myIntPtr;     // Deletes the memory pointed
delete [ ] arrayPtr; // to but not the pointer variable
```

Pointers and Dynamic Memory

Dynamic memory allocation provides a more flexible solution when memory requirements vary greatly.

The memory pool for dynamic memory allocation is larger than that set aside for static memory allocation.

Dynamic memory can be returned to the free store and allocated for storing other data at later points in a program. (reused)

Inline functions

- On function call instruction, CPU stores the memory address of instruction following the function call
- CPU then transfers the control to callee function
- CPU executes callee function, stores function return value at predefined memory location/register and returns control back to caller
- It becomes an overhead if execution time of function is less than switching time for caller function

Inline functions

main.cpp

```
int main() {
	int x = 20;
	int y = 10;
	cout << "Sum of Numbers: " << AddNumbers(x, y) << endl;
	cout << "Difference between Numbers: " << SubtractNumbers(x, y)
	cout << "Multiplication of numbers: "  << MultiplyNumbers(x,
}
```

save regs
pass args

call AddNumbers

z = x – y;

restore regs

Mymaths.cpp

```
class ArithmaticOperations {
	int AddNumbers(int x, int y){
		int z;
		z = x + y;
		return z;
	}
	inline int SubtractNumbers (int x, int y) {
		int z;
		z = x – y;
		return z;
	}
	int MultiplyNumbers (int x, int y) {
		int z;
		z = x * y;
		return z;
	}
}
```

PROLOG

EPILOG

Inline Functions

- Inline functions reduce the call overhead.
- Inline functions gets expanded when called
  - i.e. when inline function is called, entire code of inline function is inserted/substituted at point of inline function call
  - The substitution is performed by compiler at compile time

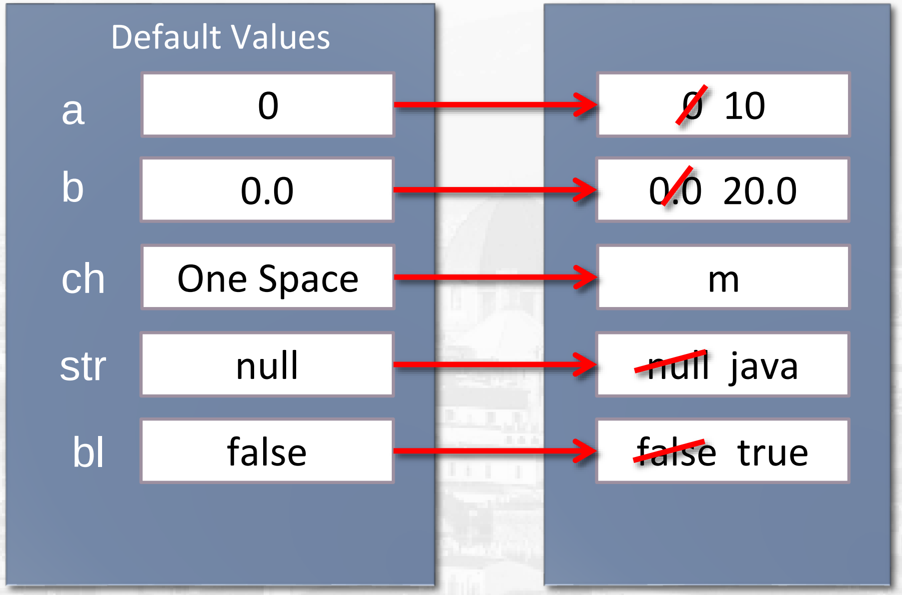
```
inline return-type function-name(parameters)
{
    // function code
}
```

- By default compiler treats class methods defined under class as inline functions

Constructors

- A constructor is a special member function that is a member of a class and has **<u>same</u>** name as that of class.
- It is used to initialize the object of the class type with a legal initial value.
- It is called constructor because it constructs the values of data members of the class.

```
Class A
{
        int a;
        float b;
        char ch;
        String str;
        boolean bl;

A()
{
        a=10;
        b=20.0;
        ch='m';
        str="java"
        bl=true
}
};
```



Constructor - Declaration

- For the T class:

  T(args); // inside class definition

  or T::T(args); // outside class definition

```
class X {
    int i ;
    public:
            int j,k ;
            X() {
                i = j = k =0;
            }
};
 X::X()
{
i = j = k =0;
}
```

Inside class

Outside class (inline definition)

Constructor - Properties

Name same as Class name

```
class X {
    int i ;
    public:
          int j,k ;
          X() {
              i = j = k =0;
          }
};
```

Under public section

Does not return anything. Not even void

- Automatically called when an object is created
- We can define our own constructors
- They can not be inherited.
- These cannot be static.
- Overloading of constructors is possible
- Constructors can have default argument as other C++ functions.
- If you do not specify a constructor, the compiler generates a default constructor for you (expects no parameters and has an empty body).
- Default and copy constructor are generated by the compiler whenever required.

Types of Constructors

- There are several forms in which a constructor can take its shape, namely

Constructors

- Default Constructor
- Parametrized Constructor
- Default Parametrized Constructor
- Copy Constructor

Default Constructor

- This constructor has no argument in it
  - Compiler creates one, if not explicitly defined
- Default Constructor is also called as *no argument constructor*

```
class rectangle{
		private:
				float height;
				float width;
		public:
rectangle(){
		cout <<“creating rectangle object”;
}
};
```

```
int main()
{
		rectangle rect;
}
```

```
# a.out
creating rectangle object
```

Parameterized Constructors

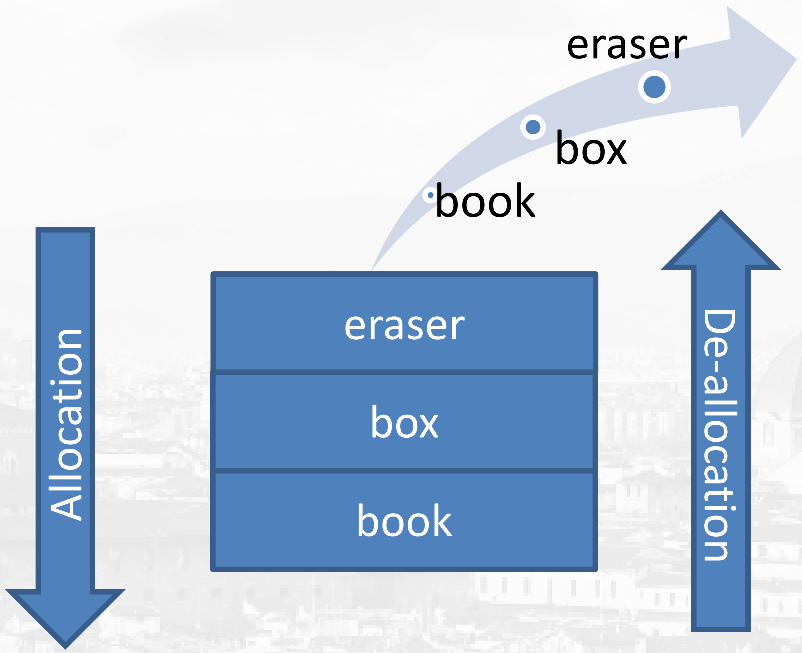
- A parameterized constructor is just one that has parameters specified in it.
- We can pass the arguments to constructor function when object is created.
- A constructor that can take arguments are called *parameterized constructors*

```
class rectangle{
    private:
        float height;
        float width;
    public:
        rectangle(float h, float w){
            height=h;
            width=w;
        }
};
```

```
int main()
{
        rectangle book(10.0, 20.0);
        rectangle box = rectangle(20.0,30.0);
        rectangle eraser= rectangle(25.0, 35.0);
}
```

Memory Allocation

- It is important to understand that compiler allocates memory to objects sequentially and destroys in reverse order. This is because C++ compiler uses the concept of stack in memory allocation and de-allocation



Default Parametrized Constructors

- Default argument is an argument to a function that a programmer is not required to specify.
- C++ allow the programmer to specify default arguments that always have a value, even if one is not specified when calling the funtion

    e.g. int power(int a, int b=2);

- The programmer may call this function in two ways

    result = power(10,3);   // result = 10^3 = 1000

    result = power(10);   // result = 10^2 = 100

- On similar lines, it is possible to define constructors with default parameters

    rectangle(float h=1.0, float w=2.0)

    and hence these are valid call statements

        rectangle book(10.0, 20.0);   // results into book object with height=10, width=20

        rectangle box(10.0);   // results into box object with height=10, width=2

Constructor Overloading

- You can have more than one constructor in a class, as long as each has a different list of arguments

```
class rectangle{
        private:
                float height;
                float width;
        public:
          rectangle(){
            height=width=10.0;
          }
           rectangle(float h, float w){
                height=h;
                width=w;
           }
        };
```

Example of default and default parameterized constructor

```
class rectangle{
        private:
                float height;
                float width;
        public:

rectangle(){
        height=width=1.0;
}
rectangle(float h, float w=5.0){
        height = h;
        width = w;
}
};
```

```
void main()
{
   rectangle book(); //implicit call of default constructor
   rectangle box(20.0); //implicit call of default
                                    parametrized constructor
   rectangle eraser(10.0, 20.0); // explicit call of default
                                    parametrized constructor
   rectangle sharpener = rectangle(10);
   rectangle geometry_box = rectangle(50.0,70.0);
   paper = rectangle (3.0, 6.0);
   calculator = rectangle (15.0, 25.0) //explicit call
                                          for existing object
}
```

Copy Constructor

clare and initialize an object

con

s **pu**

argu

ugh a copy constructor is known

T&);

```
class rectangle{
        private:
                float height;
                float width;
        public:
rectangle(float h, float w){
        height=h;
        width=w;
}
rectangle(rectangle &p){
        height = p.height;
        width = p.width;
}
};
```

```
int main()
{
        rectangle book_1(10.0, 20.0);
        rectangle book_2(book_1);
}
```

This would define the book_2 object and at the same time initialize it to the value of book_1. i.e. height and width of book_2 object would be 10 and 20 respectively

Destructors

- A destructor is used to destroy the objects that have been created by a constructor.
- Like constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.

  ~T();
- It is a good practice to declare destructors in a program since it releases memory space for further use.
- Whenever ***new*** is used to allocate memory in the constructor, we should use ***delete*** to free that memory.

Destructors

- It is a special member function of a class, which is used to destroy the memory of object
- Its name is same as class name but tilde sign preceding destructor
- It must be declared in public section
- It does not return any value; not even void
- Does not need to call because it gets call automatically whenever object is destroyed from its scope
- It can be called explicitly also using <u>delete</u> operator
- It does not take parameters
- Destructor cannot be overloaded nor inherited.

Destructors

```
int count=0;
class rectangle
{
  public:
   rectangle(){
               count++;
               cout<<"\n Created ObjectId:"<<count;
    }
 ~rectangle() {
                cout<<"\n Destroyed ObjectId:"<<count;
               count--;
               }
};
```

```
int main()
{
cout<<"\n enter main";
rectangle a1,a2,a3,a4;
cout<<"\nEnter block1";
rectangle a5;
cout<<"\nEnter block2";
rectangle A6;
cout<<"\nReenter main";
return 0;
}
```

```
#a.out
enter main
Created ObjectId:1
Created ObjectId:2
Created ObjectId:3
Created ObjectId:4
Enter block1
Created ObjectId:5
Enter block2
Created ObjectId:6
Reenter main
Destroyed ObjectId:6
Destroyed ObjectId:5
Destroyed ObjectId:4
Destroyed ObjectId:3
Destroyed ObjectId:2
Destroyed ObjectId:1
```

Destructors

```
int count=0;
class rectangle
{
  public:
   rectangle(){
              count++;
              cout<<"\n Created ObjectId:"<<count;
    }
 ~rectangle() {
               cout<<"\n Destroyed ObjectId:"<<count;
              count--;
              }
};
```

```
int main()
{
                cout<<"\n enter main";
                rectangle a1,a2,a3,a4;
                {
                                cout<<"\nEnter block1";
                                 rectangle a5;
                }
                {
                cout<<"\nEnter block2";
                rectangle A6;
                }
                cout<<"\nRe-enter main";
                return 0;
}
```

```
#a.out
enter main
Created ObjectId:1
Created ObjectId:2
Created ObjectId:3
Created ObjectId:4
Enter block1
Created ObjectId:5
Destroyed ObjectId:5
Enter block2
Created ObjectId:5
Destroyed ObjectId:5
Re-enter main
Destroyed ObjectId:4
Destroyed ObjectId:3
Destroyed ObjectId:2
Destroyed ObjectId:1
```

Destructors

```
int count=0;
class rectangle
{
  public:
   rectangle(){
              count++;
              cout<<"\n Created ObjectId:"<<count;
    }
 ~rectangle() {
               cout<<"\n Destroyed ObjectId:"<<count;
              count--;
              }
};
```

```
int main()
{
              cout<<"\n enter main";
              {
              rectangle a1,a2,a3,a4;
              }
              cout<<"\nEnter block1";
              rectangle a5;
              cout<<"\nEnter block2";
              rectangle A6;
              cout<<"\nRe-enter main";
              return 0;
}
```

```
enter main
Created ObjectId:1
Created ObjectId:2
Created ObjectId:3
Created ObjectId:4
Destroyed ObjectId:4
Destroyed ObjectId:3
Destroyed ObjectId:2
Destroyed ObjectId:1
Enter block1
Created ObjectId:1
Enter block2
Created ObjectId:2
Re-enter main
Destroyed ObjectId:2
Destroyed ObjectId:1
```

Example of Dynamic Arrays

```
#include <iostream>
using namespace std;
class Box {
   public:
      Box() {
         cout << "Constructor called!"
<<endl;
      }
      ~Box() {
         cout << "Destructor called!"
<<endl;
      }
};
int main() {
   Box* myBoxArray = new Box[4];
   delete [] myBoxArray; // Delete
array
   return 0;
}
}
```

Output

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```

Friend Functions/Classes

- Friends allow function/class access to private data of other classes.
- Friend functions
  - A 'friend' function has access to all private and protected members (variables and functions) of the class for which it is a 'friend'.
  - friend function is not the actual member of the class.
  - To declare a 'friend' function, include its prototype within the class, preceding it with the C++ keyword 'friend'.

Friend class Example

```
#include <iostream>
using namespace std;
class XYZ
{ private:
        char ch='A';
        int num = 11;
 public:  friend class ABC;
};
class ABC
{ public:
        void disp(XYZ obj)
        {
         cout<<obj.ch<<endl;
         cout<<obj.num<<endl;
        }
};
int main()
{
 ABC obj;
 XYZ obj2;
 obj.disp(obj2);
 return 0;
}
```

Output:
A
11

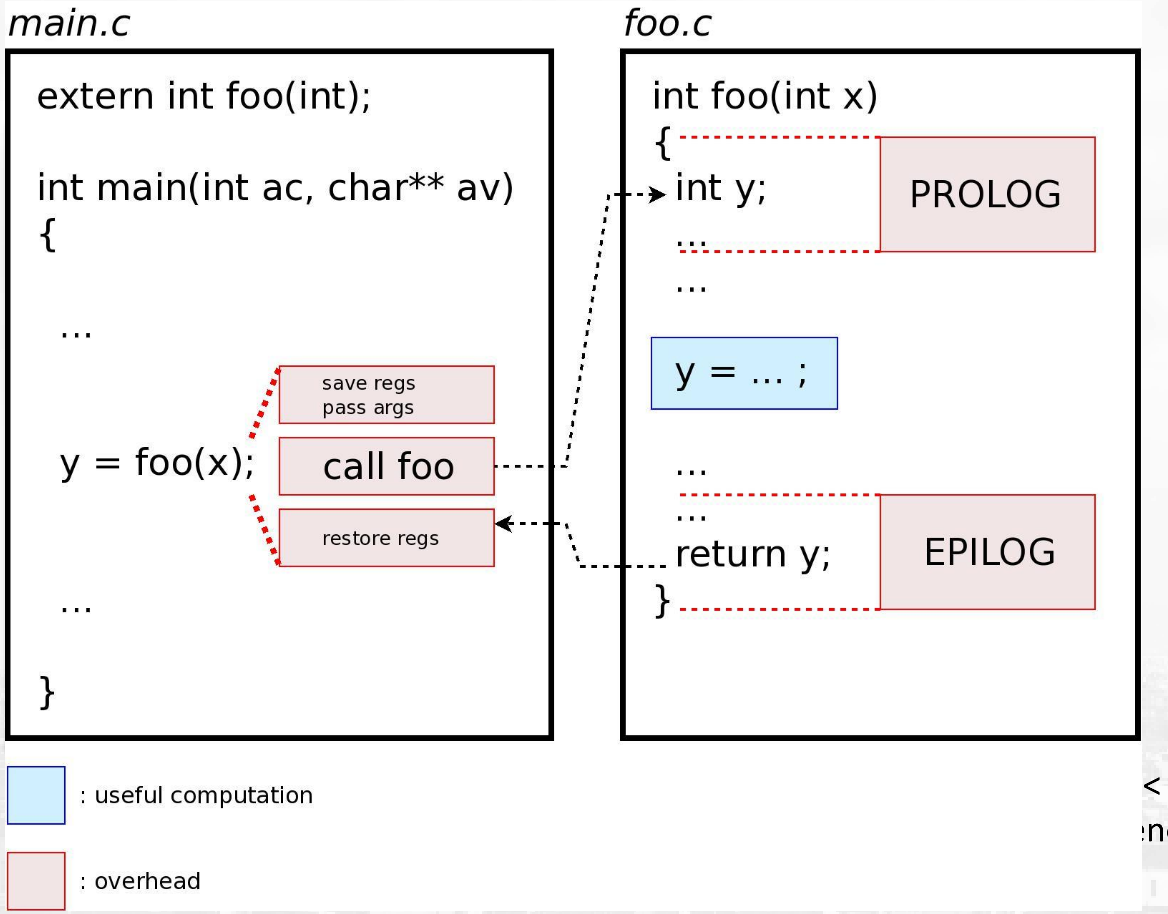
Friend Function

```
#include <iostream>
using namespace std;
class XYZ
{ private: int num=100;
        char ch='Z';
public: friend void disp(XYZ obj);
 };

void disp(XYZ obj)
{
cout<<obj.num<<endl;
cout<<obj.ch<<endl;
}
int main()
{ XYZ obj;
  disp(obj);
  return 0;
}
```

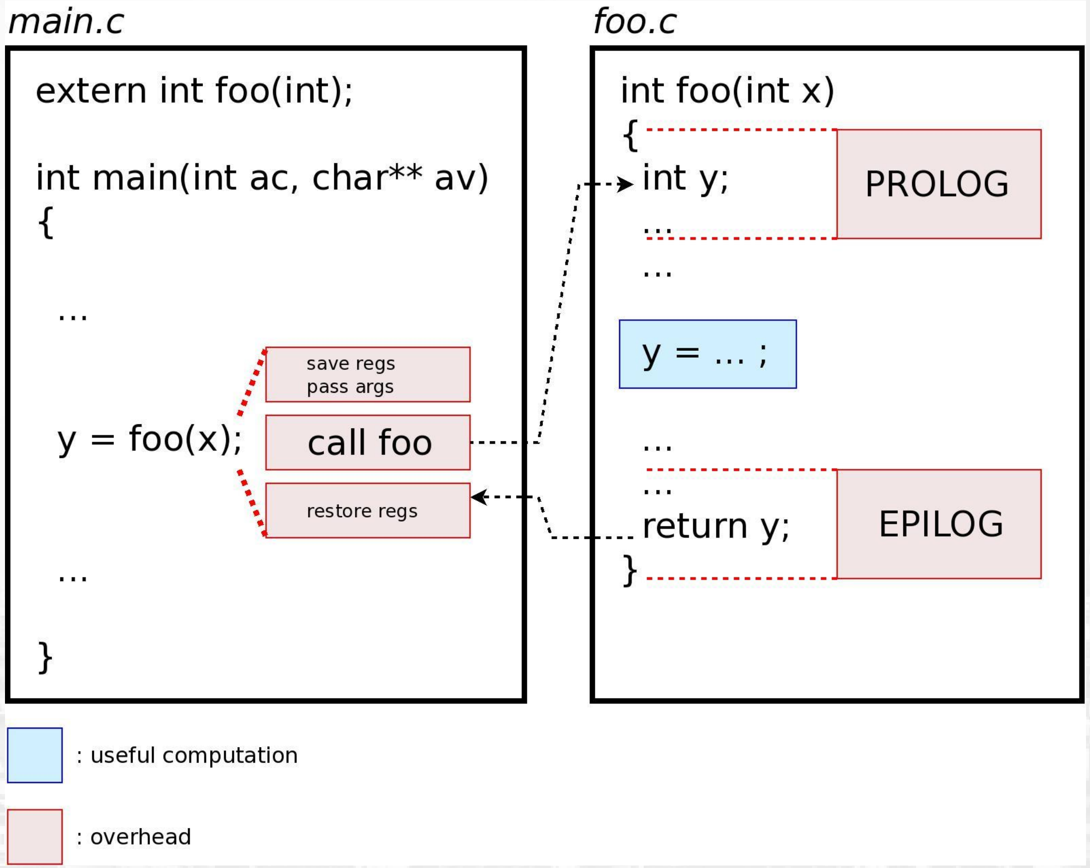
Output:
100
Z

BACKUP SLIDES

Function Calls



< endl;
ndl;



Inline functions

- Compiler may not perform inlining in such circumstances like:
  1) If a function contains a loop. (for, while, do-while)
  2) If a function contains static variables.
  3) If a function is recursive.
  4) If a function return type is other than void, and the return statement doesn’t exist in function body.
  5) If a function contains switch or goto statement.

Function Calls

```
class ArithmaticOperations {
    int AddNumbers(int x, int y){
        return x + y;
    }
    int SubtractNumbers (int x, int y) {
        return x –y;
    }
    int MultiplyNumbers (
        return x * y;
    }
}

int main() {
    int x = 20;
    int y = 10;
    cout << "Sum of Num
    cout << "Difference b
    cout << "Multiplicatio
}
```