Object Oriented Programming

Unit 1: Introduction to Object Oriented Programming

Introduction to Object Oriented Programming

1.1. Structured Programming Vs. Object Oriented Programming

1.2. Object Oriented Programming Concepts

1.3. Advantages and Application of OO Methodology

1.4. Classes and Object:

- **1.4.1 Defining Class**
- **1.4.2 Access Specifier**
- **1.4.3 Creating Object**

1.5. Modular programming with functions:

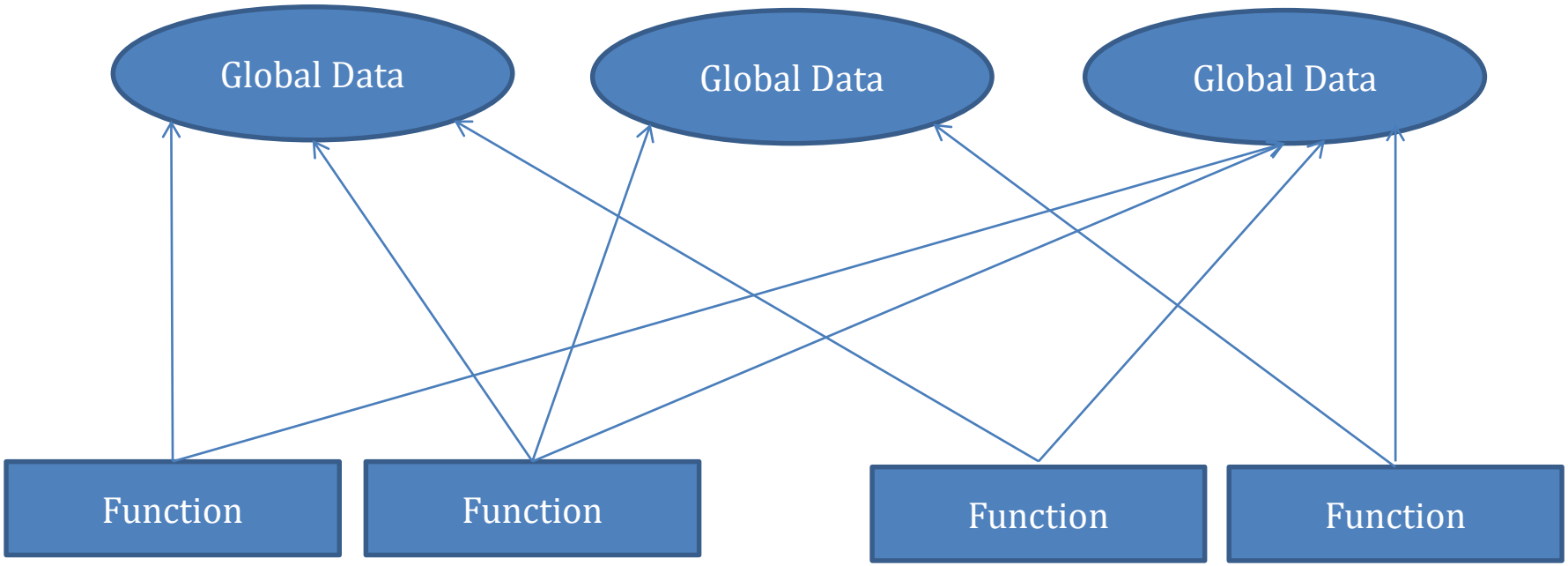
- **1.5.1 Return by reference**
- **1.5.2 Default argument**
- **1.5.3 Inline functions**
- **1.5.4 Arrays and functions**
- **1.5.5 Storage class**
- **1.5.6 Function with variable number of arguments**

CE: 0.1 Procedure – Oriented Programming Paradigm

- What happens when your local super market moves the cash from regular section 01 to section 10?

- Everyone who supports the supermarket must have to figure out where the cash is and adjust their habits accordingly.

CE: 0.1 Procedure – Oriented Programming Paradigm



CE: 0.1 Procedure – Oriented Programming Paradigm

- In a large program, there are many functions and global data items.
- When data items are modified in a large program, it may not be easy to tell which function access the data.
- Even when you figure this out modifications to the function may cause them to work incorrectly with other global data items.

CE: 0.2 Object – Oriented Programming Paradigm Basic

Think of object as Departments…..

- Such as Sales, Account, Purchase and so on in a company.
- Each department has its own assigned (tasks)duties.
- Each department has its own data as well.
- People in each department control and operate on that department's data.
- Dividing company into department makes it easier company's activity and maintain information used by company.

CE: 0.2 Object – Oriented Programming Paradigm Basic

- **Object – Oriented** approach is to remove some of the flaws encountered in the procedural approach.

- OOP ties data more closely to the functions that operate on it, and protects it from accidental modification from outside function.

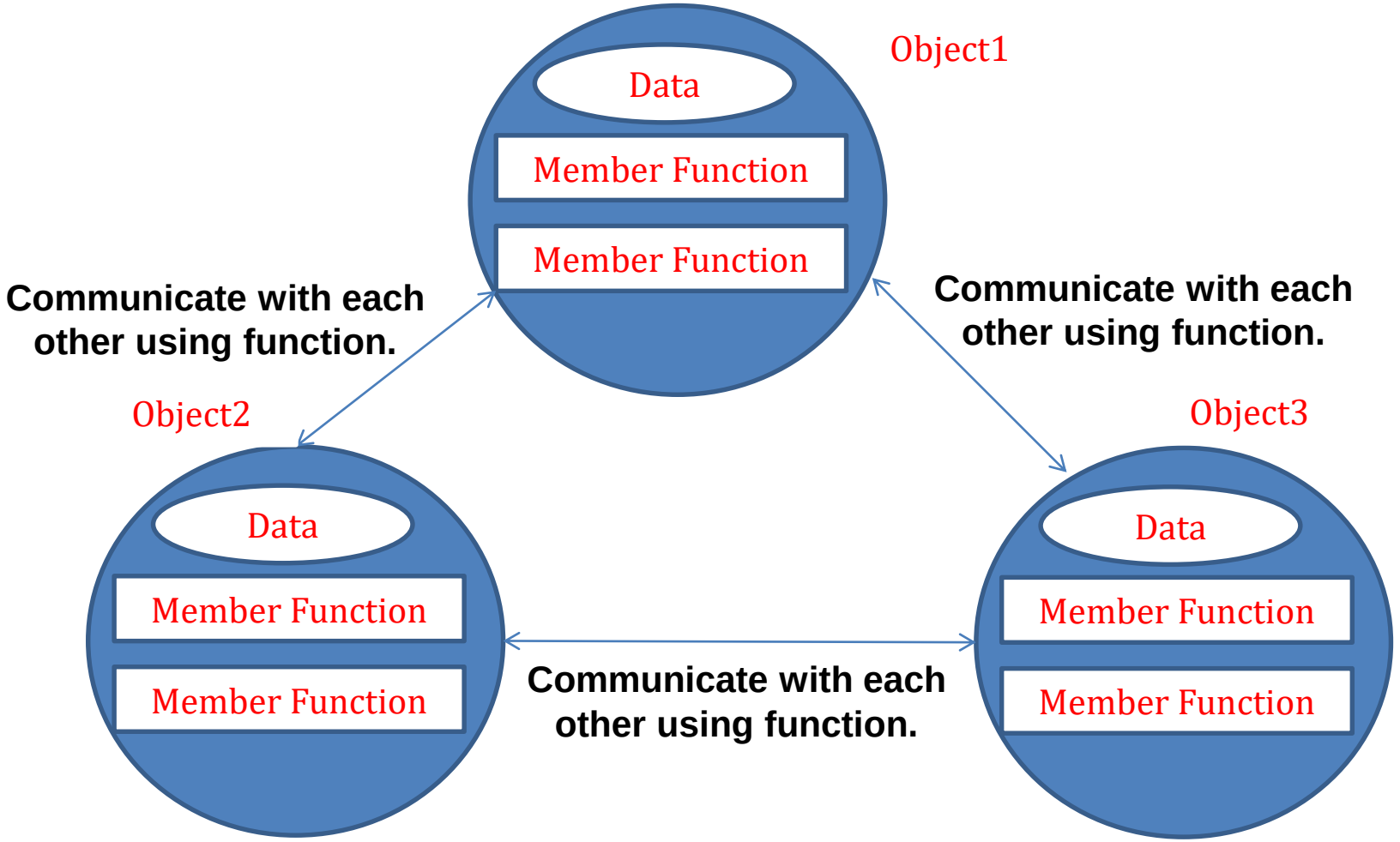
- OOP allows decomposition of a problem in to a number of entities called **objects** and **build data** and **function around these objects.**

CE: 0.2 Object – Oriented Programming Paradigm Basic

- If a sales manager need some data then he send a message to a person in the sales department, then wait for the person to access the data and send reply with information.

- It ensures that this data is accessed accurately and not by outsiders.

CE: 0.2 Object – Oriented Programming Paradigm Basic



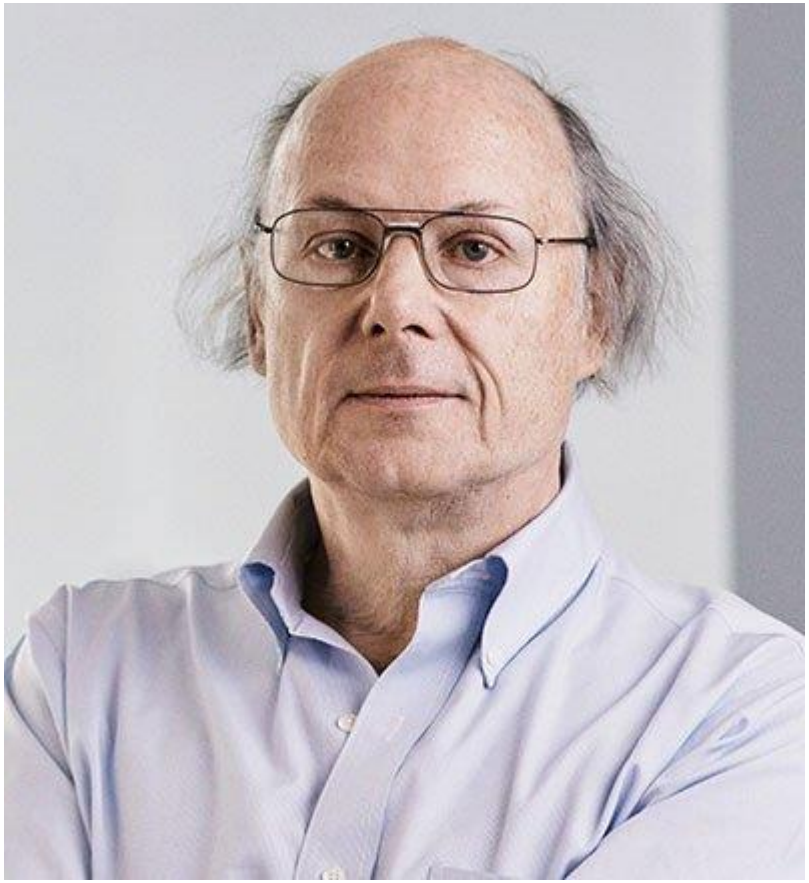
Organization of data and function in OOP

What is C++?

- C++ is observed as a **middle-level language**, as it includes a combination of both **high-level** and **low-level language** features.
- C++ is a statically typed, compiled, general purpose, case-sensitive, free-form programming language that supports procedural, object-oriented and generic programming.
- **C++ is a superset of C** that enhancement to the **C language** and originally named **"C with Classes"**, but later it was renamed **C++**.
- C++ is an **O**bject **O**riented **P**rogramming language.
- The most important facilities that C++ adds are classes, inheritance, function overloading and operator overloading.

Developed

C++



Bjarne Stroustrup

at AT&T Bell laboratories in Murray Hill, New Jersey, USA, in the year 1980.

CE: 0.3 Object – Oriented Programming Paradigm

Some strict features of OOP are:

1. OOP emphasis on data, rather than procedure or function.
2. OOP programs are divided into small procedures to perform various tasks.
3. Data structure are designed, such that they characterize the object.
4. Group that operate on the data of an object are tied together in the data structure.
5. Data are hidden and cannot be accessed by external function.
6. Objects may communicates with each other through function.

CE: 0.3 Object – Oriented Programming Paradigm

Some strict features of OOP are: (Conti…)

7. New data and function can be easily added whenever necessary.

8. Follows bottom-up approach in program design.
   [**Bottom-up approach**: Individual elements are specified in a great detail, then linked together to form a large sub system.]

CE: 0.3 Object – Oriented Programming Paradigm

- **Object Oriented programming** is a programming style that is associated with the concept of Classes, Objects and various other concepts revolving around these two, like…
  - Inheritance,
  - Polymorphism,
  - Abstraction,
  - Encapsulation, etc.

- "**Object – Oriented programming** is an approach that provides a way of modularizing programs by creating partitioned (separated) memory area for both data and function."

- **It is well-suited paradigm.**

CE: 0.3 Object – Oriented Programming Paradigm

Why it is well-suited paradigm?

1. Modeling the real-world problem as close as possible to the user's viewpoint.

2. Interacting easily with computational environment using familiar metaphors.

3. Constructing reusable software components and easily extendable libraries.

4. Easily modifying and extending implementation of components without having to recode everything from scratch.

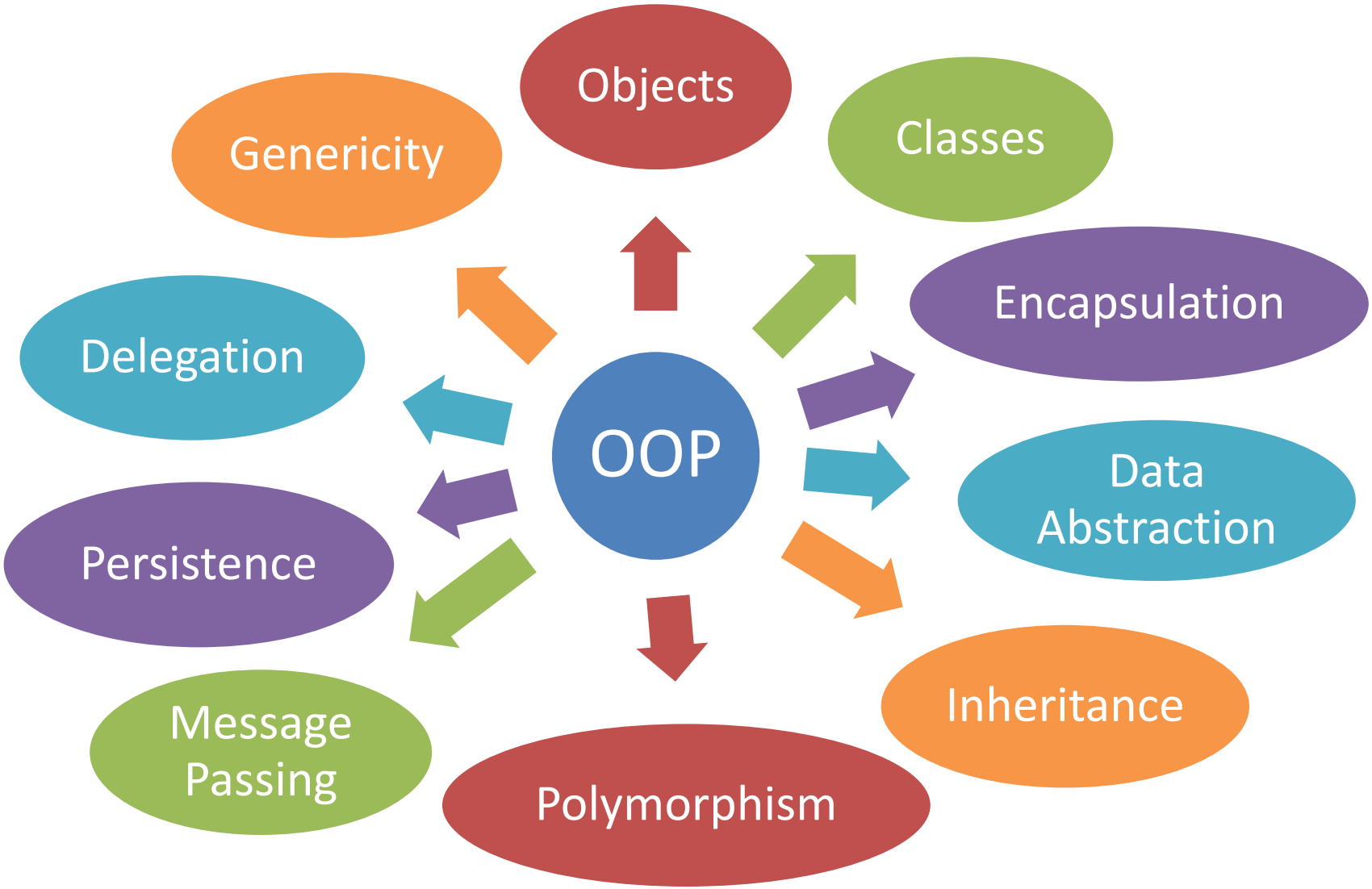
CE: 1.1
Structured Programming Vs. Object Oriented Programming

CE: 1.1 Structured Programming Vs. Object Oriented Programming

Structured Programming	Object Oriented Programming
1. It uses higher level language like C, COBOL, FORTRAN.	1. It uses 30 many languages. Some of them are like C++, JAVA, Smalltalk. Today C++ is also known as Industrial Standard Language.
2. It employs top-down programming approach.	2. It employs bottom-up programming approach.
3. Programs are divided into smaller programs known as function.	3. Programs are divided into collection of a number of entities called objects.
4. Data move openly around the system from function to function.	4. Data is hidden and cannot be accessed by external function.
5. Most of the function share global data. Function transfers data from one to another.	5. Objects may communicate with each other through function. New data and function can be easily added whenever necessary.

CE: 1.2
Object Oriented Programming Concepts

CE: 1.2 Object Oriented Programming Concepts

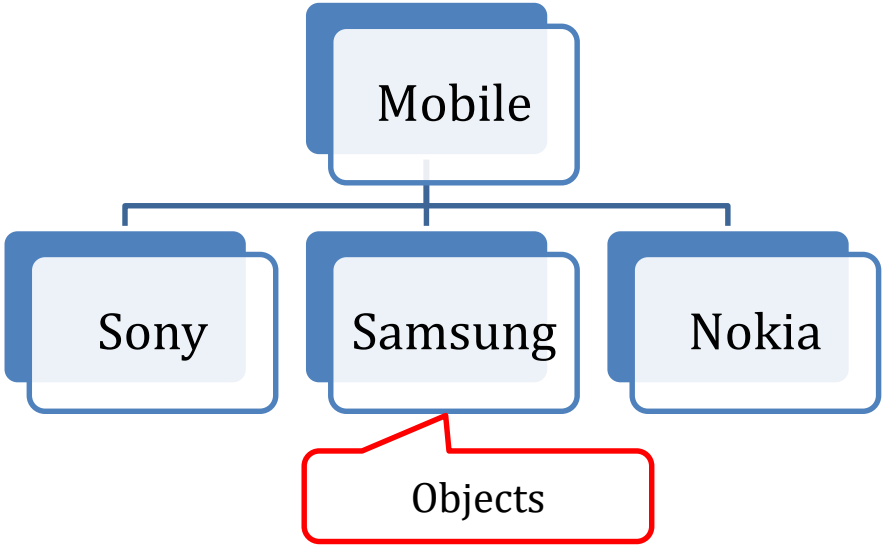


CE: 1.2 Object Oriented Programming Concepts

Features of OOPs:

1. Objects
2. Classes
3. Encapsulation
4. Data Abstraction
5. Inheritance
6. Polymorphism
7. Message Passing
8. Persistence
9. Delegation
10. Genericity

1. Objects:

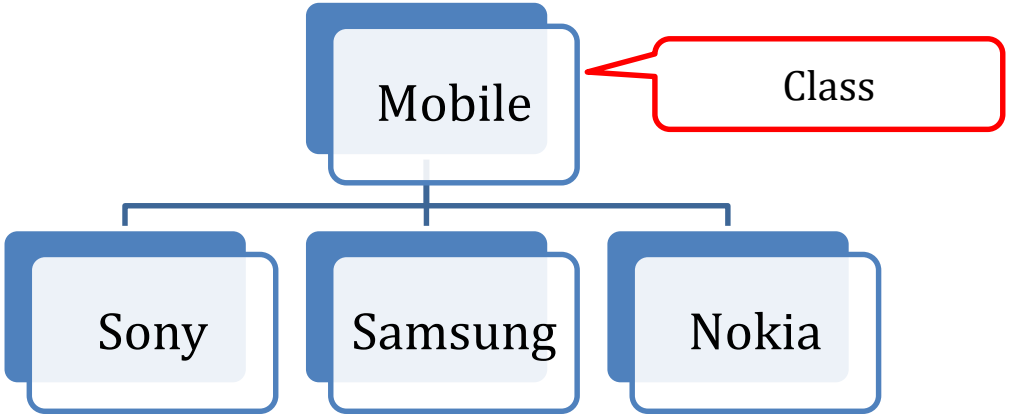


- Basically, an object is anything that is identifiable as a single material item.
- You can see around and find many objects like Camera, Monitor, Laptop, etc.
- **An object is nothing, but an instance of a class, which represents a real-world entity.**

1. Objects: (Conti…)

- Object are basic run-time entities in an Object-Oriented system.
- For Example:
  A person , A place, A bank account, An item.
- Objects in program should be chosen, such that it must match closely with the real-world objects.
- It takes space in the memory and have an association address.
- It interacts by sending message to one another.
- **Objects are variable of the type class.**

2. Classes:



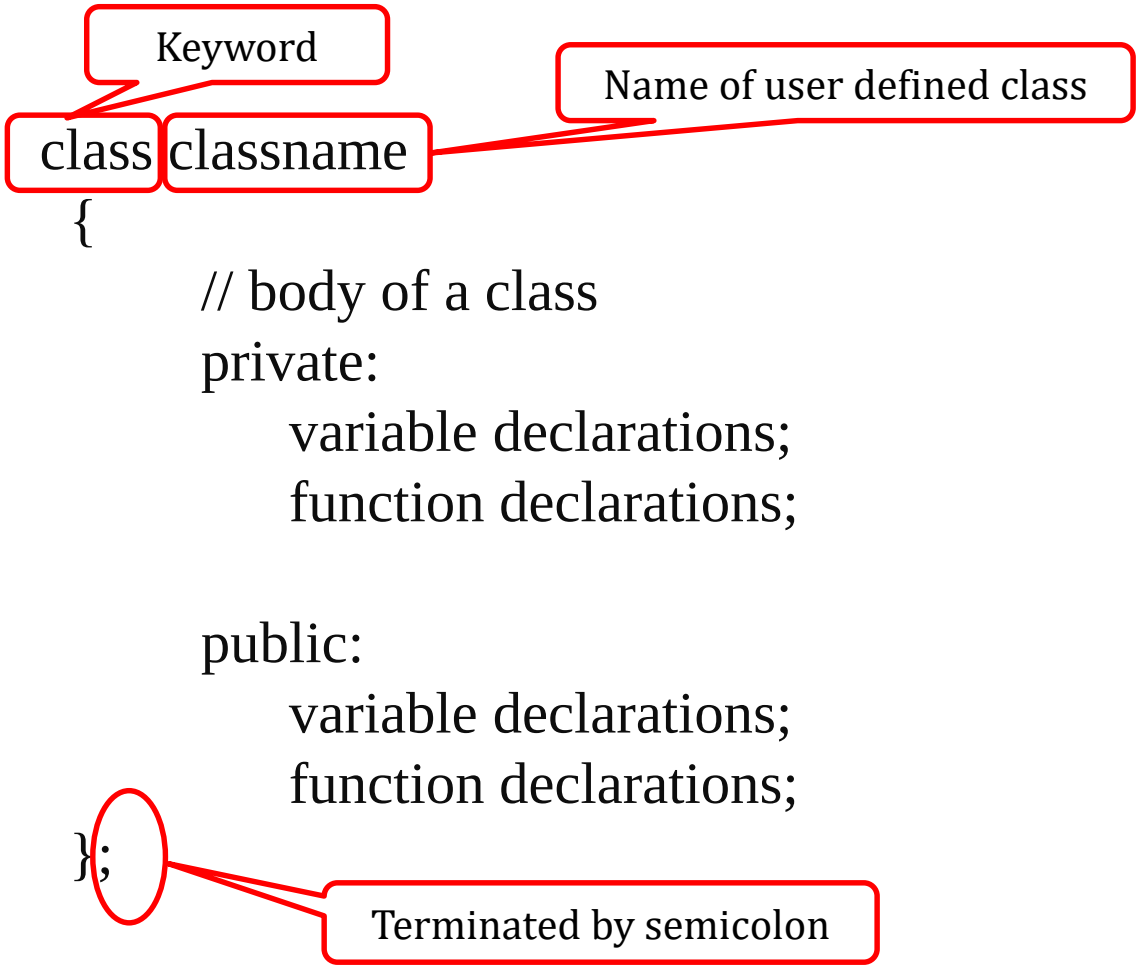
- A Mobile can be a class which has some attributes like….
  Profile Type, IMEI Number, Processor, and some more.) & operations like…..
  Dial, Receive & Send Message.

- **A class is a collection of objects of similar type.**

- The class declaration is similar to a **struct** declaration.

2. Classes: (Conti…)

- Syntax of a class:



2. Classes: (Conti…)

- Classes are data types based, on which objects are created.
- Once a class is created, we can create number of variables of that type using the class name.
- The classes contain not only data, but also functions.
- The functions are called **member functions**.

3. Encapsulation:

How do you think a car runs?

- Is there only steering, break and whatever you are able to see directly??
- **NO!**

- There are many things to make car run. Engine, wires, fuel tank, etc. but we can't see it.

- So that, we can say that car is basically a collective unit of many parts which helps car to run.

- Same way, in classes also, we have **functions** and **data members** that are wrapped together.

3. Encapsulation: (Conti..)

- The wrapping up of data and functions into a single unit is known as **encapsulation**.
- Its helps to keep data safe from external interference and misuse. Therefore it is not accessible to the outside world.
- Only those functions that are wrapped in the class can access them.
- These functions provide the interface between the object's data and the program.
- It also takes care about, how much access should be given to a particular.
- This protection of data from direct access by the program is called **data hiding** or **information hiding**.

4. Data Abstraction:

Lets continue with the above example.

- Suppose you started going to learn how to drive.
- Does the instructor teaches you about the functioning of car engine?
- **NO!**
- Why, he doesn’t??
- Because he knows that for learning driving, you don’t need to know about the functioning of engine. All those are kept away from you.

4. Data Abstraction: (Conti…)

- **Abstraction** means displaying only essential information and hiding the details.
- **Data abstraction** refers to provide *only essential information about the data to the outside world*; and *hiding the background details or implementation*.
- It creates the ability to create user-defined data types, for modeling a real world object and set of permitted operators.
- Attributes are called **data member**; and functions are called **methods** or **member function**.

Abstraction & Encapsulation:

- Both Abstraction & Encapsulation works hand in hand, because **Abstraction takes care about what details to be made visible** & **Encapsulation provides the level of access rights to that visible details.**
  i.e. – It implements the desired level of abstraction.

Abstraction & Encapsulation: (Conti..)

- For Example:
- Bluetooth, which we usually have it in our mobile. When we switch on the Bluetooth I am able to connect another mobile, but not able to access the other mobile features like dialing a number, accessing inbox etc. This is because, Bluetooth feature is given some level of abstraction.

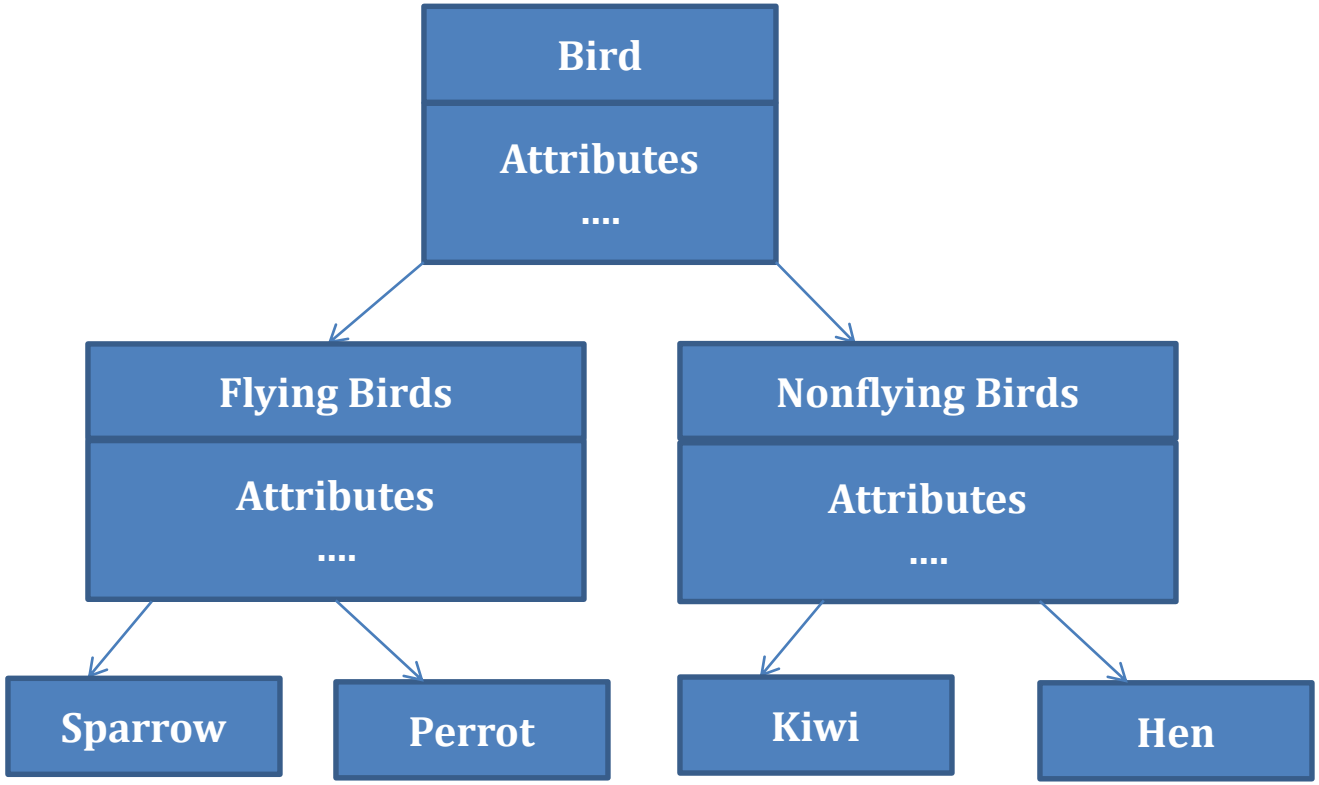
- Another point is when mobile A is connected with mobile B via Bluetooth, whereas mobile B is already connected to mobile C, then A is not allowed to connect C via B. This is because of accessibility restriction.

5. Inheritance:

- Technically, we are derived from our parents and they are derived from grandparents and so on.
- In our case the child is always smaller than the parents, but here the derived class is always bigger than the base class.
- **Inheritance** is the process by which object of one class obtain the properties of another class.
- Inheritance allows the **extension** and **reusability** of existing code without writing the code again from the scratch.
- We can derive a new class from existing class. Then, the new class will be having the combined feature of both classes.

5. Inheritance: (Conti…)

- It supports the concept of hierarchic classification.



6. Polymorphism:

Lets take the example of some words which can be used in different situations with different meaning.

- 'watch'.... watch can be used as a verb as well as a noun...
- "he is wearing a watch".. in this it is used as a noun.. "watch your actions".. in this it is used as a verb..
- Thus we see that the same word can be used in different situations differently.
- Likewise same operators or function names are used with various meanings in various situations.

6. Polymorphism: (Conti…)

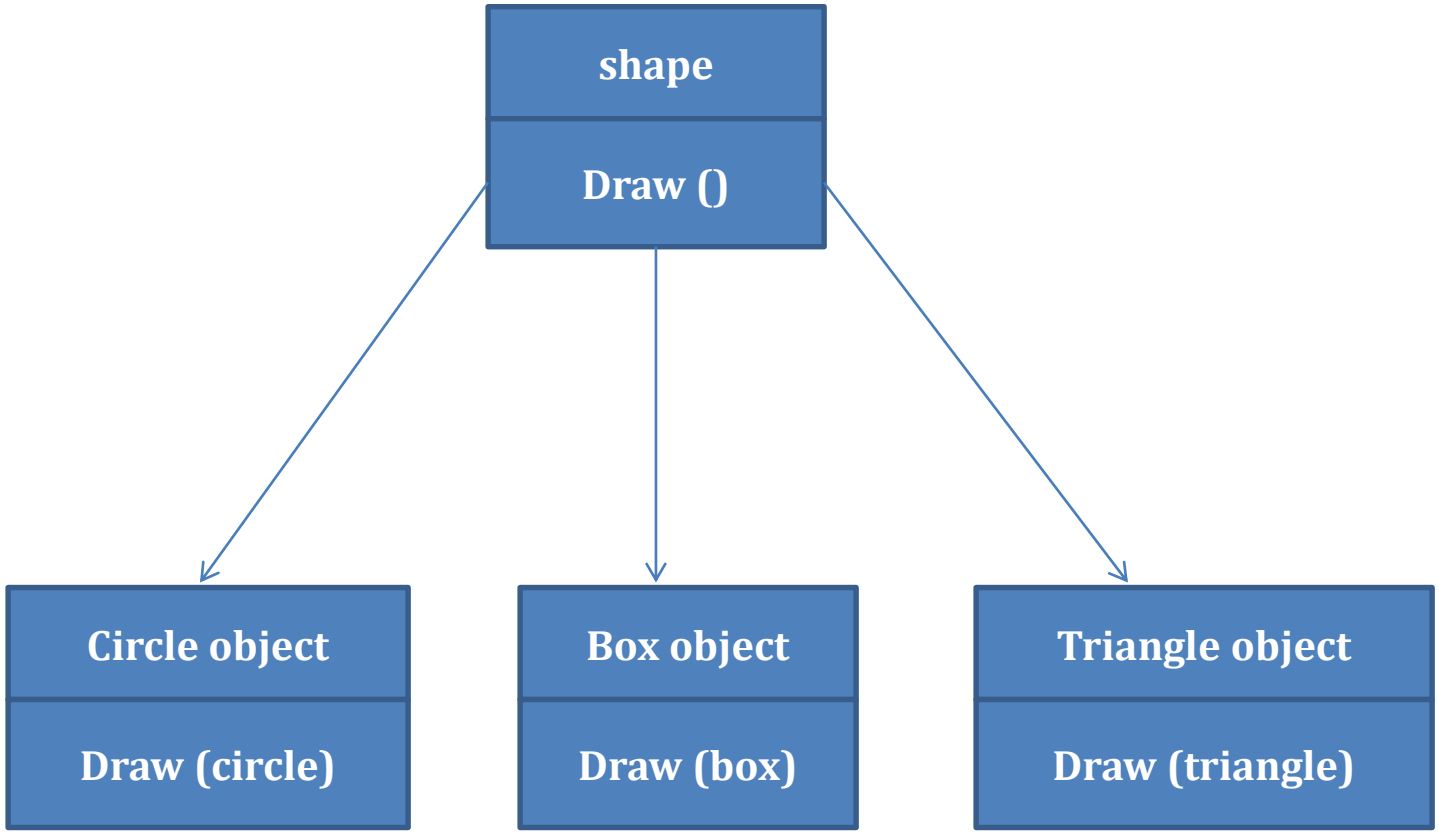
- Polymorphism is a Greek term, it means ability to take one form.
- An operation may shows different behavior in different instances.
- The behavior depends upon the type of data used in the operation.
- In C++, it is achieved by **function overloading**, **operator overloading** and **dynamic binding** (virtual functions.).
- **Operator Overloading:**
  The process of making an operator to show different behaviors in different request is known as ***Operator Overloading***.

Operation: Addition
Two Numbers = 2 + 5 =7
Two String = A + B = AB

6. Polymorphism: (Conti…)

Function Overloading:

- By using a single function name, to perform different types of tasks is known as ***function overloading***.



6. Polymorphism: (Conti…)

Dynamic Binding (virtual functions):

- ***Binding*** refers to the performance of associating an object or a class with its member.
- The matching of a function call, by the C++ compiler function definition at run time is called ***dynamic binding***.
- **Dynamic binding is associate with inheritance and polymorphism.**

7. Message Passing:

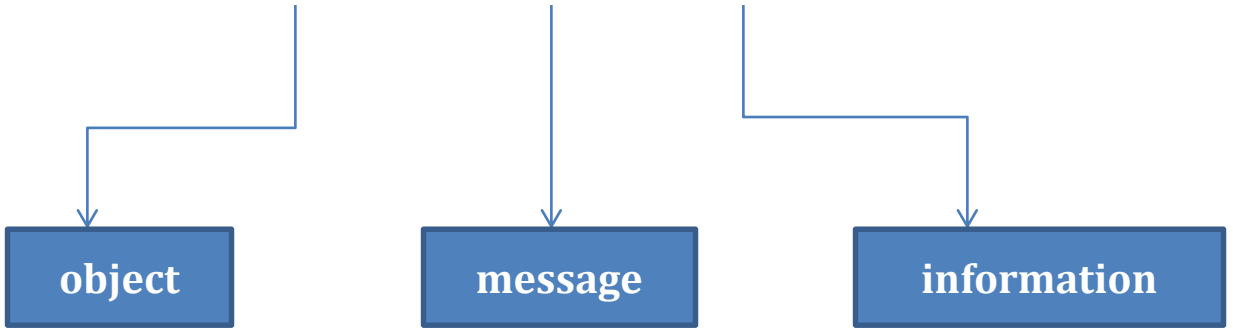
- In a C++ program, communications takes place between various objects.

- During objects communication, messages are passed in functions, this process is known as message passing.

- The process of programming in an object-oriented language, involves following basic steps:
  - ✓ Creating classes that defines objects and their behavior,
  - ✓ Creating object from class definition, and
  - ✓ Establishing communication among objects.

7. Message Passing: (Conti…)

- Objects have a life cycle. They can created and destroyed.
- Therefore, Communication with an object is possible only when it is alive.



employee. salary(name);

8. Persistence:

- The occurrence, where the data (object) survives during the program execution time and exists between executions of a programs is known as ***persistence***.

or

- ***Persistent*** objects are those which survive between sequential invocations of the program.

- All database systems support persistence.

- **In C++, this is not supported.**

- But, the user can build it explicitly (openly) using file streams in a programs.

9. Delegation:

- It is an alternative to class inheritance.
- **Delegation** is a way of making object composition as powerful as inheritance.
- It is an object oriented technique (also called a design pattern) where certain operations on one object are automatically applied to another, usually contained, object.
- In object-oriented language, delegation can be done openly, by passing the *sending object* to the *receiving object*.
- So ultimately, this approach takes a view that an object can be a collection of many objects and the relationship, which is known as ***has-a*** relationship or ***containership***.

9. Delegation: (Conti…)

- C++ has language mechanisms, to support code reuse through inheritance, while utilizing **delegation requires extra work by the programmer**.

10. Genericity:

- It is a technique for defining software components, that have more than one interpretation depending on the data type.

- Generic programs can be written once, compiled once and used for different data types.

- C++ achieves genericity by two ways:
  1. function templates and
  2. class templates

- For Example:
  - ✓ A software company may need **sort** for different data types. Rather than writing and maintaining the multiple codes, we can write one **sort()** and pass data type as a parameter.

CE: 1.3
Advantages and Application of OO Methodology

CE: 1.3.1 Benefits of OOP

1. Through inheritance, we can eliminate redundant code and extend the use of existing classes.

2. We can build programs from the standard working modules, that communicates with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.

3. The principle of data hiding helps the programmers to build secure programs that cannot be entered by code in other parts of program.

4. Object – oriented systems can be easily upgraded from small to large system.

CE: 1.3.1 Benefits of OOP (Conti…)

5. Message passing technique for the communication between objects, makes the interface descriptive with external system much simpler.

6. Software complexity can be easily managed.

CE: 1.3.2 Applications of Object- Oriented Programming

1. Real-time system
2. Simulation and Modeling
3. Object – oriented databases
4. Hypertext
5. AI and Expert System
6. Neural network and parallel programming
7. Decision support and office automation system
8. CIM/CAM/CAD system

CE: 1.3.2 Applications of C++

1. Development of editors
2. Development of compiler
3. Development of database
4. Communication system
5. Any complex real-life application system

CE:
Basic of C++ Program

I/O Operators:

- Input Operator: To accepts the user input.

cin >> number1;

- The above statement introduce two new C++ feature: cin and >>
- The identifier cin (pronounce as ‘C in’)
- The operator >> is called the **extraction** or **get from** operator.
- **Output Operator:** To display text on output screen.

cout << “Hello! Welcome to C++”;

- The above statement introduce two new C++ feature: cout and <<
- The identifier cout (pronounce as ‘C out’)
- The operator << is called the **insertion** or **put to** operator.

Cascading of I/O Operators:

- **Cascading of Output Operator:** The multiple use of << insertion operator in one statement is called **cascading**.
- For Example:

```
cout<<"Sum = " << sum <<"\n";
cout<<"Sum = " <<sum <<"," <<"Average = " <<average ;
```

- **Cascading of Input Operator:** The multiple use of >> extraction operator in one statement.
- For Example:

```
cin>>average>> sum ;
```

Simple basic example of C++:

```
#include<iostream>

using namespace std;

int main()
  {
        cout<<“ Welcome to C++”; //Output

        int a, b;
        cin>>a>>b; //Cascading of input
        cout<<“a = ”<<a <<“b = ”<<b;

        return 0; // it terminates main() function and causes it to return
                     the value 0 to the calling process.
   }
```

#include<iostream>:

- In C++, it is basically a standard library header file.
- If you want to add streams in your program you need to include it.
- It provides basic **input** and **output** services for C++ programs.
- **iostream** uses objects like…
  - ✓ **cin** for sending data to the standard streams **input** and
  - ✓ **cout** for sending data from the standard streams **output**.

Namespace:

- **Namespace** define a scope for the ***identifiers*** that are used in program.
- For using the identifier defined in the namespace scope, we must **include** using directives like,

using namespace std;

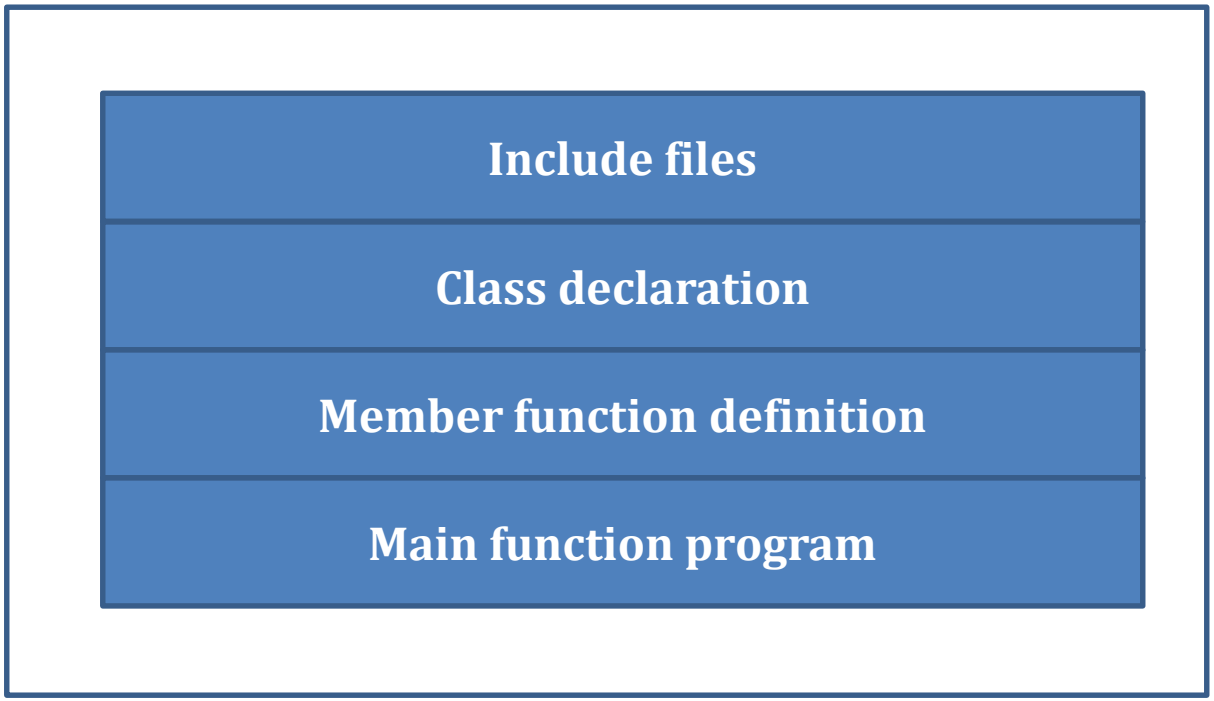
- **Std** is the namespace, where ANSI C++ standard class libraries are define.
- This will bring all the identifiers define in **std** to the current scope.
- It tells the compiler to use the **std** namespace. Namespaces are a relatively recent addition to C++.

Identifier:

- Identifier refer to the ***names of variables, function, arrays, classes,*** etc., created by programmer.
- The rules for naming identifiers
  1. Only alphabetic character, digits and underscores are permitted.
  2. The name cannot start with digit.
  3. Uppercase and lowercase letters are distinct.
  4. A declared keyword cannot be used as a variable name.

C recognize only the first 32 character in identifier name, where as **C++ places no limit on length of identifier name.**

Structure of C++ Program:



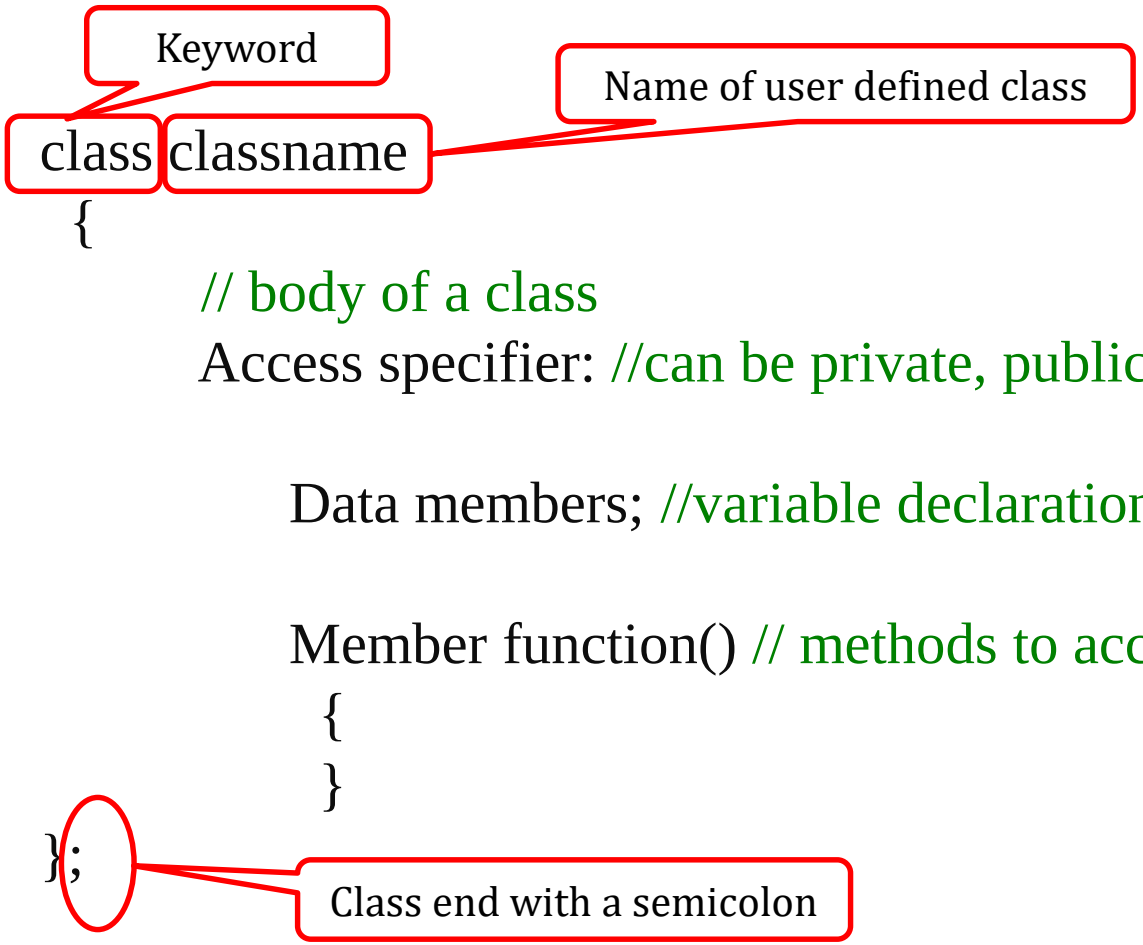
Basic built-in Datatypes in C++:

DATA TYPE	SIZE (IN BYTES)	RANGE
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	
wchar_t	2 or 4	1 wide character

CE: 1.4
Classes and Object

1.4.1. Defining Class:

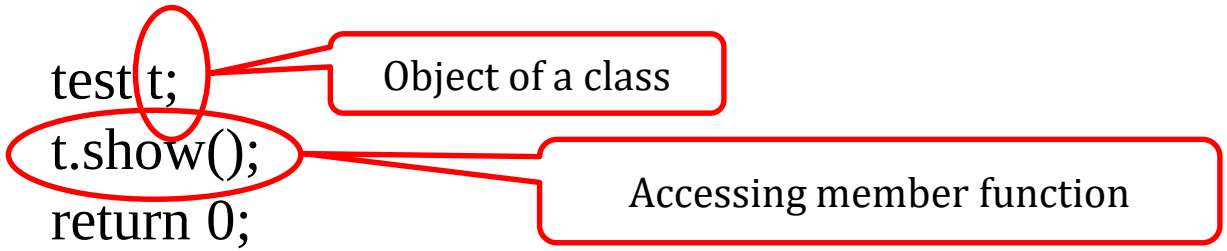
- Syntax of a class:



1.4.1. Defining Class: (Conti…)

- For Example:

```
class test
  {
        public:  //Access specifier
            void show()
               {
                   cout<<"Welcome to C++ class";
                }
};
int main()
   {
        test t;
        t.show();
        return 0;
   }
```



1.4.2. Access Specifier in C++:

- Access specifiers/modifiers are used to implement an important feature of Object Oriented Programming known as Data Hiding.

- Access specifiers in a class are used to set the accessibility of the class members.

- That is, it is used to set some restrictions on the class members not to get directly accessed by the outside functions.

- There are 3 types of access modifiers available in C++:
  1. Public
  2. Private
  3. Protected

1.4.2. Access Specifier in C++: (Conti…)

Access Specifier	Accessible to	
	Own class members	Objects of a class
private:	Yes	No
protected:	Yes	No
public:	Yes	Yes

Visibility of class members

1.4.2. Access Specifier in C++: (Conti…)

1. Private:

- A ***private*** member variable or function cannot be accessed, or even viewed from outside the class.

- Only the class and friend functions can access private members.

- By default all the members of a class would be private.

1.4.2. Access Specifier in C++: (Conti…)

1. Private: (Conti…)

- For example:

```
class student
  {
        private:
            int rno;
            char name[];
            void getdata();
   };

int main()
   {    student s1;
        s=s1.rno; // cannot access private function
        s1.getdata(); // cannot access private function
    }
```

1.4.2. Access Specifier in C++: (Conti…)

1. **Private: (Conti…)**

- Since, class is only having access control to all its members, there is no means to communicate with the external world.

- Thus, the “private” access specifier will not contribute anything to the program.

1.4.2. Access Specifier in C++: (Conti…)

2. Protected:

- A ***protected*** member variable or function is very similar to a private member, but it provides one additional benefit that they can be accessed in child classes which are called derived classes.

1.4.2. Access Specifier in C++: (Conti…)

2. Protected: (Conti…)

- For example:

```
class student
  {
        protected:
            int rno;
            char name[];
            void getdata();
   };

int main()
   {    student s1;
        s=s1.rno; // cannot access private function
        s1.getdata(); // cannot access private function
    }
```

1.4.2. Access Specifier in C++: (Conti…)

3. Public:

- The members of a class are accessible by the outsider class, should be declared in a ***public*** section.

- All data members and functions declared in the public section of the class can be accessed without any restriction from anywhere in the program.

1.4.2. Access Specifier in C++: (Conti…)

3. Public: (Conti…)

- For example:

```
class student
  {
        public:
            int rno;
            char name[];
            void getdata();
   };

int main()
   {    student s1;
        s=s1.rno;
        s1.getdata();
    }
```

1.4.3. Creating Object:

- To create class objects…

  class student S1;

  or

  student S1;

- Object can also be created by placing their names immediately after closing brace like…

```
class student
   {

    }S1, S2;
```

Class Work

- Write a program to print a message “Welcome to C++ class!” using class.
- Write a program to perform addition of two numbers using class.
- Create a class called “student” which contains “enro” and “name” as data members. Take the information of student by user input in one function and display it in another function.

CE: 1.5
Modular programming with functions

What is the need of Function?

- A complex problem is often easier to solve by dividing it into several smaller parts, each of which can be solved by itself. This is called **structured** programming.

- These parts are made into functions in C++.

- **main()** uses these functions to solve the original problem.

What actually the function do?[Advantages]

- Functions separate the concept **(what is done)** from the implementation **(how it is done)**.
- Functions make programs easier to understand.
- Functions can be called several times in the same program, allowing the code to be reused.

Main Function:

- Definition of main function:

```
main()
 {
        // main program statements
 }
```

- C does not specify any return type for the main() function, which is the starting point for the execution of program.

- This is perfectly valid because the main() in C does not return any value.

Main Function: (Conti…)

- In C++, the main() returns a value of type integer to the Operating System.

- C++, explicitly define main() as matching one of the following prototype:

```
int main();
int main(int argc, char * argv[]); // command line argument
```

- The function that have a return value for that return statement is used.

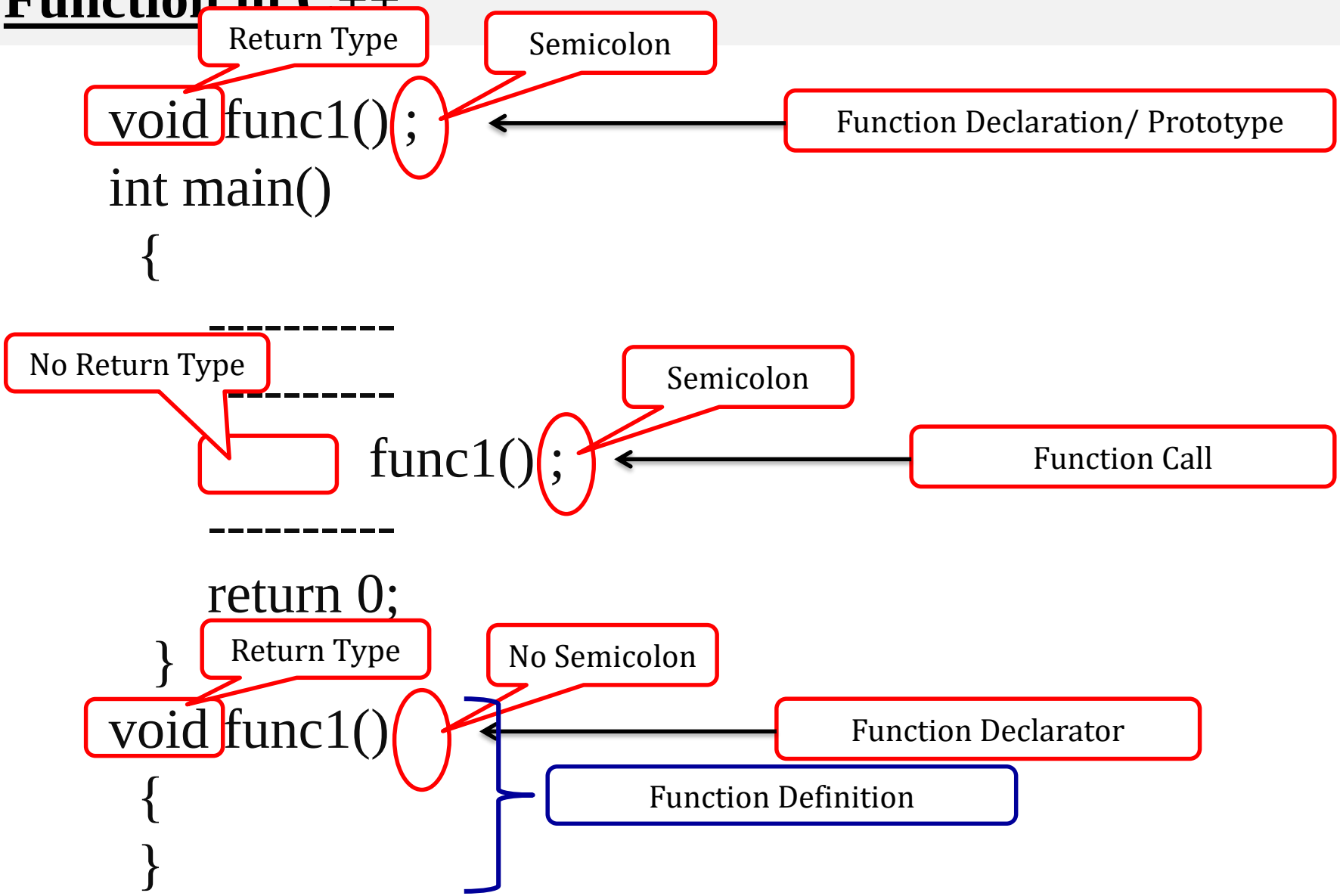
Main Function: (Conti…)

- In C++, the main() function define as:

```
int main()
  {
      ……
      ……
      return 0; // indicates that program was successfully executed.
  }
```

Non – zero value in return statement means there is a problem in program.

Function in C++:



Demonstration:

- For Example:

```
#include <iostream>
using namespace std;
void max(int a, int b);

void max(int a, int b)
 {
      if (a > b)
        {
              cout<<“a is greater!”;
        }
      else
      { cout<<“b is greater!”; }
}
```

```
int main()
 {
     cout << max(10, 20) << endl;
     return 0;
 }
```

Parameter Passing (Revision)

- C++ supports three types of parameter passing schemes:
  1. Pass by Value
  2. Pass by Address
  3. Pass by Reference (only in C++)

1. Pass by Value:

- For Example: Swap integer values by value.

```
#include <iostream>
using namespace std;
void swap(int x, int y);

void swap(int x, int y)
 {
      int k;
      cout<<“x = ”<<x;
      cout<<“y = ”<<y;
      k = x;
      x = y;
      y = k;
      cout<<“x = ”<<x;
      cout<<“y = ”<<y;
}
```

```
int main()
 {
      int a, b;
      cin>>a>>b;
      swap(a, b);
      return 0;
 }
```

2. Pass by Address:

- For Example: Swap integer values by pointers.

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

void swap(int *x, int *y)
 {
      int k;
      cout<<“x = ”<<x;
      cout<<“y = ”<<y;
      k = *x;
      *x = *y;
      *y = k;
      cout<<“x = ”<<x;
      cout<<“y = ”<<y;
}
```

```
int main()
 {
     int a, b;
     cin>>a>>b;
     swap(&a, &b);
     return 0;
 }
```

3. Pass by Reference: (It only works in C++)

- For Example: Swap integer values by reference.

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

void swap(int &x, int &y)
 {
      int k;
      cout<<“x = ”<<x;
      cout<<“y = ”<<y;
      k = x;
      x = y;
      y = k;
      cout<<“x = ”<<x;
      cout<<“y = ”<<y;
}
```

```
int main()
 {
     int a, b;
     cin>>a>>b;
     swap(a, b);
     return 0;
 }
```

1.5.1. Return by Reference:

- For Example: Return variables by reference.

```
int &max(int &num1, int &num2);
int main()
{
      cout<<“Largest No: ”<<max(10, 20);
       return 0;
}

int &max(int &num1, int &num2)
{
      if(num1 > num2)
          return num1;
      else
          return num2;
}
```

1.5.1. Return by Reference: (Conti…)

- For Example:

```
int &add(int &x, int &y);
int c;

int main()
 {
      int a,b;
      cout<<"Enter a & b=";
      cin>>a>>b;
      add(a,b);
      cout<<"\c="<<c;

      return 0;
 }
```

```
int &add(int &x, int &y)
{
      c= x+y;
      return c;
}
```

1.5.1. Return by Reference: (Conti…)

- A C++ program can be made easier to read and maintain by using references rather than pointers.
- A C++ function can return a reference in a similar way as it returns a pointer.
- When a function returns a reference, it returns an implicit pointer to its return value.
- This way, a function can be used on the left side of an assignment statement.

Class Work

- Write a program to find whether the year is a leap year or not, using function.

Hint:

If year is divided by 4, but not by 100, then it is a leap year.
If year is divided by both 100 and 400, then it is a leap year.
If year is divided by 400, then it is a leap year.
And in all other cases, it is not a leap year.

1.5.2. Default Arguments:

- **Default argument** is specified when the function is declared.
- Default value is specified in a manner syntactically similar to variable initialization.
- For Example:
- float simpleInterest(float principal, int period, float rate=0.15);
- float simpleInterst(float principal, int period=2, float rate);

Declare a default value 0.15 to rate

Illegal

Always Assign value right to left.

1.5.2. Default Arguments: (Conti…)

- For Example:

```
//shows the missing and default arguments.
#include<iostream>
using namespace std;
void repchar(char='*', int=5); //Function Prototype
int main()
   {     repchar();
         repchar('-');
         repchar('+',2);
         return 0;
   }  //End of  Main
void repchar(char ch, int n) //Function Definition
  {     for(int i=0; i<n; i++)
              cout<<ch<<endl;
  }  //End of Function
```

- 1st time: it is called with no arguments
- 2nd time: with no arguments
- 3rd time: with one

1.5.2. Default Arguments: (Conti…)

- For Example:

```
//A program to calculate simple interest.
float simpleInterest(float principal, float rate=0.15, int time=2);
float simpleInterest(float p, float r, int t)
  {
      float SI=p*r*t/100;
      return SI;
  }
int main()
  {
      float si;
      si = simpleInterest(500);
      cout<<"Simple Interest: "<<si;
            return 0;
  }
```

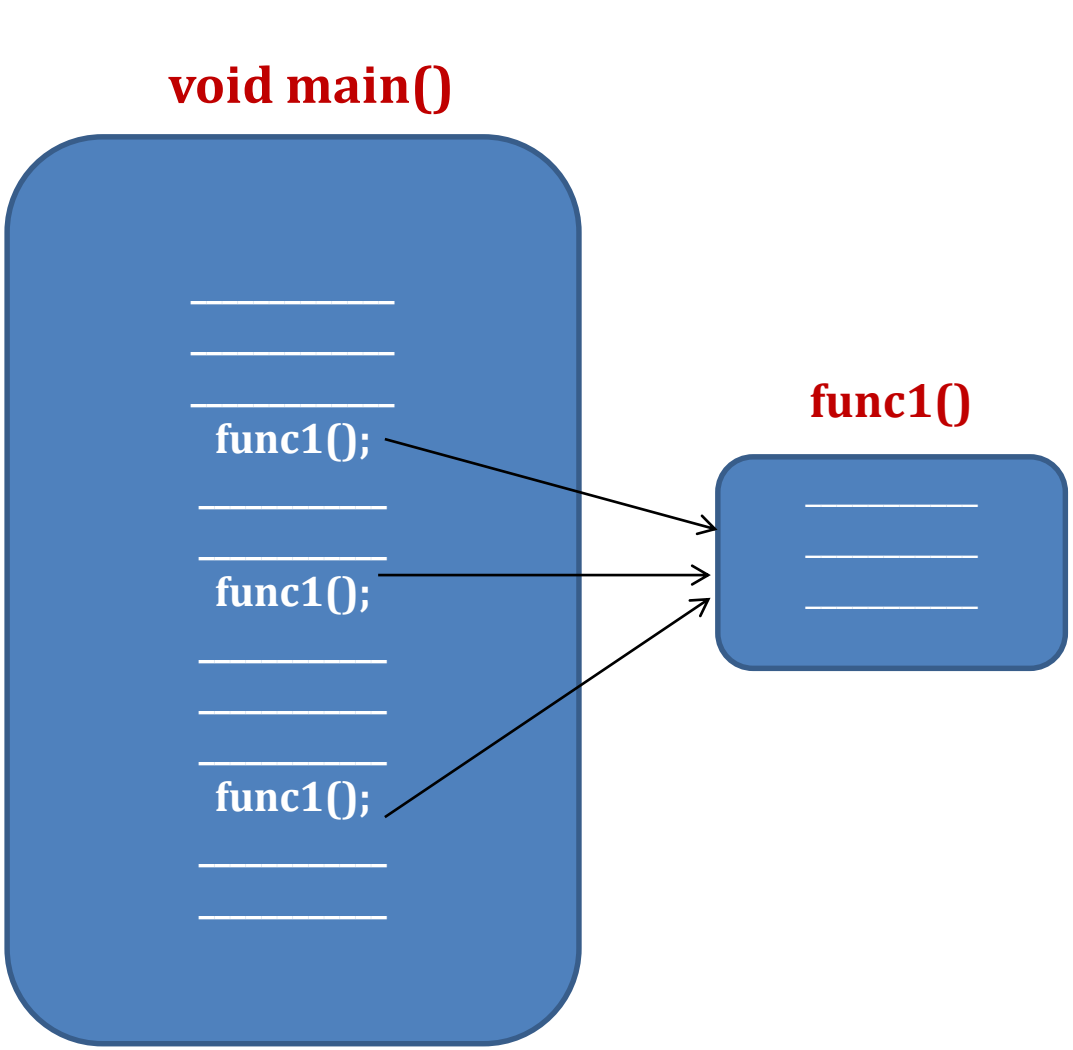
Class Work

- Write a program to calculate area of circle.
  [Use the value for PI as default argument]

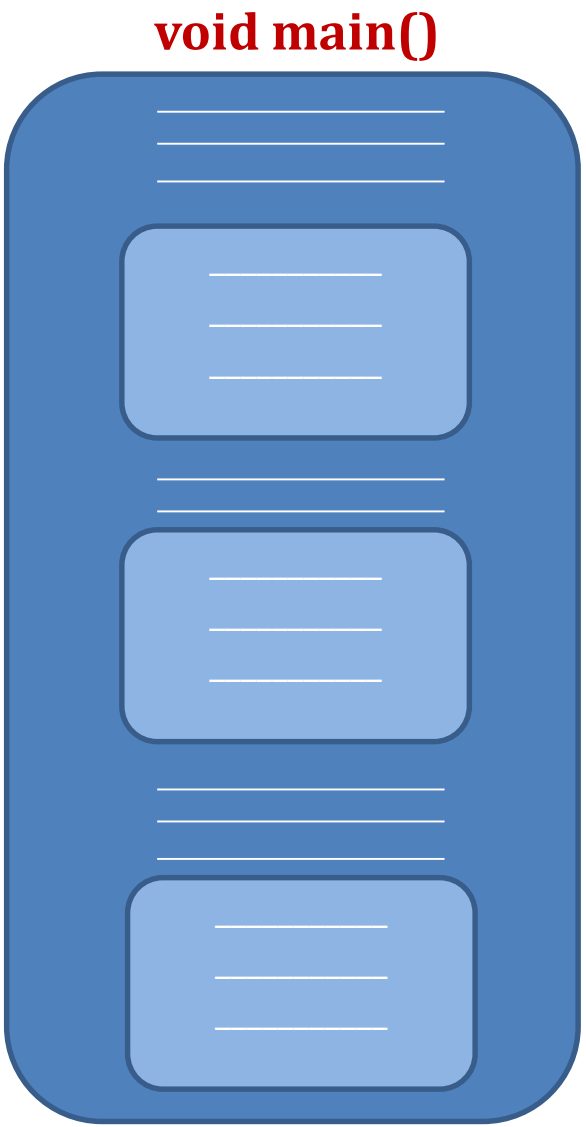
1.5.3. Inline Function:

- To eliminate the cost of the calls to small functions, C++ proposes a new features called **inline function**.
- *Inline function* is a function that is expanded inline when it is invoked.
- That is, the complier replaces the function call with the corresponding function code.

Difference between Function and Inline Function:



Repeated code placed in function



Repeated code placed in line

1.5.3. Inline Function: (Conti…)

```
inline int square(int num)
    {
        return num*num;
     }
```

```
void main()
 {
    A = square(5);
    B = square(n);
 }
```

```
void main()
 {
    A = 5*5;
    B = n*n;
 }
```

1.5.3. Inline Function:

- When a function program executes, the series of task which performs:

1. The CPU stores the memory address of the instruction.
2. Copies the argument of the function call on to the stack.
3. Finally, transfers control to the specified function.
4. CPU executes the function code.
5. Store the function return value in the predefine memory.
6. location / register.
7. Return control to the calling function.

1.5.3. Inline Function: (Conti…)

- For Example:

```
//A program to find cube, using inline function.
using namespace std;
inline double cube(double c) //Inline Function
   {
        return(c*c*c);
    }
int main()
   {
        double a;
        cout<<"Enter the value for finding cube: ";
        cin>>a;
        cout<<“The cube of a given number is: "<<cube(a)<<endl;
        return 0;
   }
```

1.5.3. Inline Function: (Conti…)

- For Example:

```
//A program to convert pounds to kilogram, using inline function.
    using namespace std;
    inline float pdtokg(float pounds) //Inline Function
      {
          return 0.453592*pounds;
       }
    int main()
      {
          float lb;
          cout<<"Enter the weight in pounds: ";
          cin>>lb;
          cout<<"Weight in KG: "<<pdtokg(lb)<<endl;
          return 0;
      }
```

Class Work

- Write a program to calculate area of circle, using inline function.

1.5.4. Array and Functions:

- Arrays can be passed to a function as an argument, just like passing variables as the arguments.

- For example: Passing One-dimensional Array to a Function

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

void display(int arr[5])
{
    cout << "Displaying elements: ";
    for (int i = 0; i < 5; ++i)
    {
        cout<<"\nArray Elements: "<< i + 1 <<": "<< arr[i] << endl;
    }
}

int main()
{
    int a[5] = {10, 20, 30, 40, 50};
    display(a);
    return 0;
}
```

1.5.4. Array and Functions: (Conti…)

- For example:

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

void sum(int arr1[], int arr2[])
   {
       int arr3[5];
       for(int i=0; i<5; i++)
        {
            arr3[i] = arr1[i]+arr2[i];
            cout<<arr3[i]<<" ";
        }
    }

int main()
 {
    int a[5] = {10, 20, 30, 40 ,50};
    int b[5] = {1, 2, 3, 4, 5};
    sum(a, b); //Passing arrays to function
    return 0;
 }
```

1.5.4. Array and Functions: (Conti…)

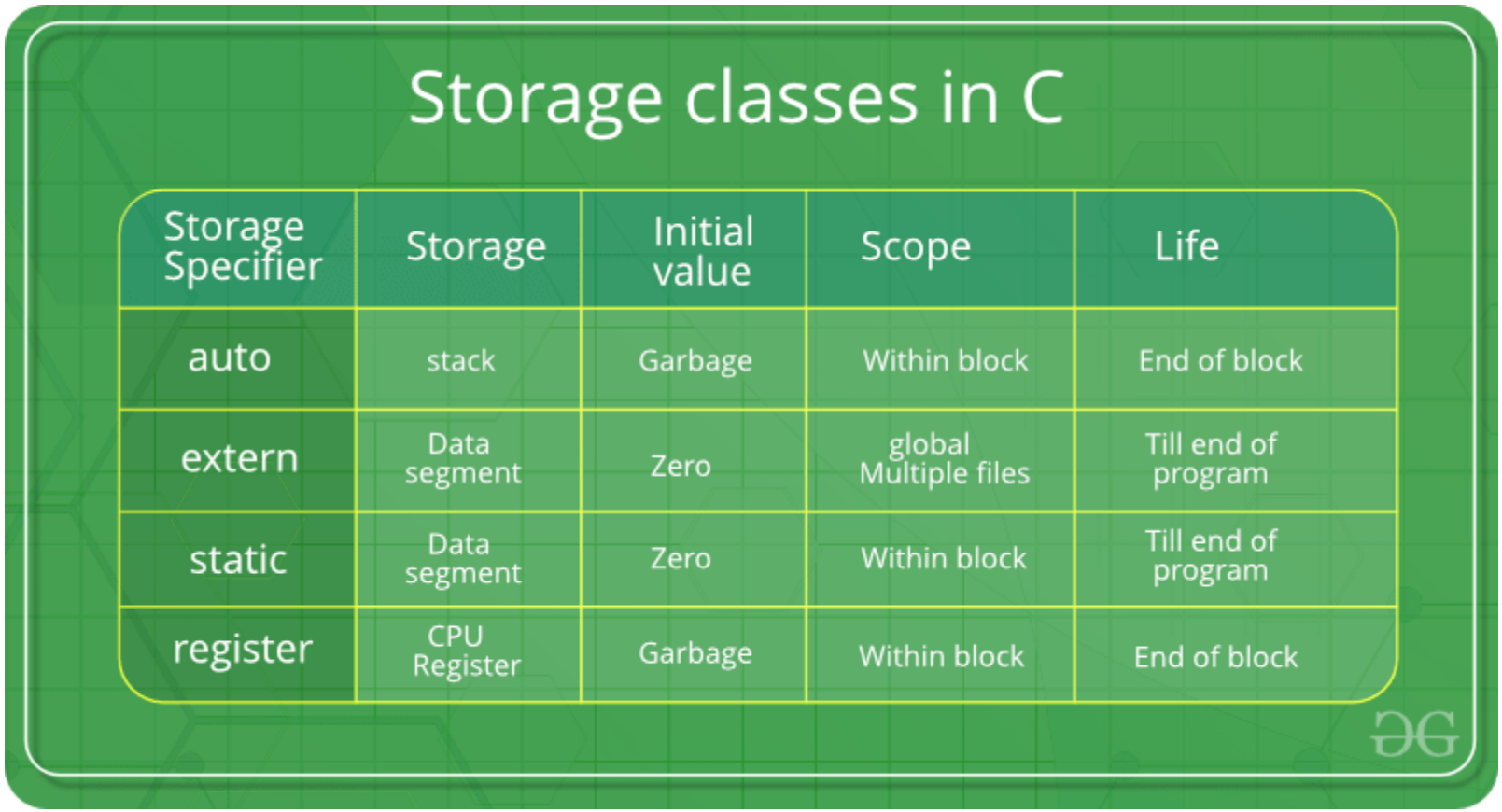
- For example: Passing Multidimensional Array to a Function

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

void display(int n[3][2])
{
   cout <<"Displaying Values: ";
   for(int i = 0;  i < 3; ++i)
   {  cout <<"\n";
      for(int j = 0; j < 2; ++j)
      {
         cout <<"\n"<<n[i][j] << " ";
      }
   }
}

int main()
{
   int num[3][2] = { {3, 4}, {9, 5},
{7, 1} };
   display(num);

   return 0;
}
```

1.5.5. Storage Class:

- We have seen that every variable has a data type.
- To fully define a variable, we need a ***storage class*** as well apart from data type.
- A ***storage class*** defines the scope and life-time of variables and/or functions within a C++ Program.
- The storage classes, which can be used in a C++ Program:
  - auto
  - register
  - static
  - extern

1.5.5. Storage Class: (Conti…)



1.5.5. Storage Class: (Conti…)

1. auto:

- The **auto** storage class is the default storage class for all local variables.

- For Example:
  - ✓ Two variables with the same storage class, auto can only be used within functions, i.e., local variables.

```
{
   int rno; // by default, storage class is auto
   auto int rno;
}
```

1.5.5. Storage Class: (Conti…)

2. register:

- The **register** storage class is used to define local variables that should be stored in a CPU register instead of RAM, which allows faster access.

- This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

- For Example:

```
{
   register int rno;
}
```

1.5.5. Storage Class: (Conti…)

3. static:

- The scope of **static** variable is local to the function in which it is defined but it doesn't die when the function execution is over.
- The value of a static variable continues between function calls.
- The default initial value of static variable is 0.
- For Example:

```
{
   static int count=0;
}
```

1.5.5. Storage Class: (Conti…)

4. extern:

- Variables of **extern** storage class have a global scope.
- It is used, when we want a variable to be visible outside the file in which it is declared. *So, an extern variable can be shared across the multiple files.*
- An extern variable remains alive as long as program execution continues.
- A *static global variable is visible only in the file*, but an *extern global variable is visible across all the files of the program*.

1.5.5. Storage Class: (Conti…)

4. extern: (Conti…)

- For Example:

```
extern int count; // global variable declaration

void func1()
{
    ….;
}
```

- The **extern** modifier is most commonly used when there are two or more files sharing the same global variables or functions.

1.5.5. Storage Class: (Conti…)

4. extern: (Conti…)

- For Example:

✓ **File1.cpp:**

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

int count;
extern void func1();

int main()
  {
        count = 5;
        func1();
  }
```

✓ **File2.cpp:**

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

extern int count;

void func1(void)
  {
      cout <<"Count is: "<< count;
  }
```

1.5.6. Function with variable number of arguments:

- To use a function with variable number of arguments, "**cstdarg**/stdarg" header file is used.

- To use this, four macros provide access in a standard form:
  1. va_list, which stores the list of arguments,
  2. va_start, which initializes the list,
  3. va_arg, which returns the next argument in the list, and
  4. va_end, which cleans up the variable argument list.

- Whenever a function is declared to have an unknown number of arguments, in place of the last argument you should place an ellipsis (which looks like '**...**').

- For example:

```
int add(int x, ... );
```

1.5.6. Function with variable number of arguments:

- int add(int x, ...); ---- would tell the compiler that the function should accept many arguments that the programmer uses, as long as it is equal to at least one, the one being the first i.e. x.

1.5.6. Function with variable number of arguments:

- Syntax of macros handling variable number of arguments:

  void va_start(va_list ap, lastfix);
  type va_arg(va_list ap, type);
  void va_end(va_list ap);

  - **void va_start(va_list ap, lastfix);**
    - ✓va_start is used before the first call to *va_arg* and *va_end*.
    - ✓va_start takes two parameters: ap and lastfix of type va_list.

    - ✓va_list – is a type to declare a variable.
    - ✓va_list array holds the information needed by *va_arg* and *va_end*.

    - ✓ap – is a pointer to the variable argument list.
    - ✓lastfix – is the name of the last fixed parameter passed to the caller.

1.5.6. Function with variable number of arguments:

- For Example: To find maximum value.

```
int FindMax (int n, ...)
  {
      int i,val,largest;
      va_list vl;
      va_start(vl,n);
      largest=va_arg(vl,int);
      for (i=1;i<n;i++)
          {
              val=va_arg(vl,int);
              largest=(largest>val)?largest:val;
          }
      va_end(vl);
      return largest;
  }
```

```
int main ()
 {
   int m;
   m=
   FindMax(7,702,422,631,834,892,104,772);
   cout<<"The largest value is: "<<m;
   return 0;
 }
```


Thank You