Object Oriented Programming[210243]
SE Computer Engineering

UNIT-I
Fundamentals of Object Oriented Programming

Unit Contents

Introduction to object-oriented programming, Need of object-oriented programming, Fundamentals of object-oriented programming: Namespaces, objects, classes, data members, methods, messages, data encapsulation, data abstraction and information hiding, inheritance, polymorphism. Benefits of OOP, C++ as object oriented programming language.

C++ Programming- C++ programming Basics, Data Types, Structures, Enumerations, control structures, Arrays and Strings, Class, Object, class and data abstraction, Access specifiers, separating interface from implementation.

Functions- Function, function prototype, accessing function and utility function, Constructors and destructor, Types of constructor, Objects and Memory requirements, Static members: variable and functions, inline function, friend function.

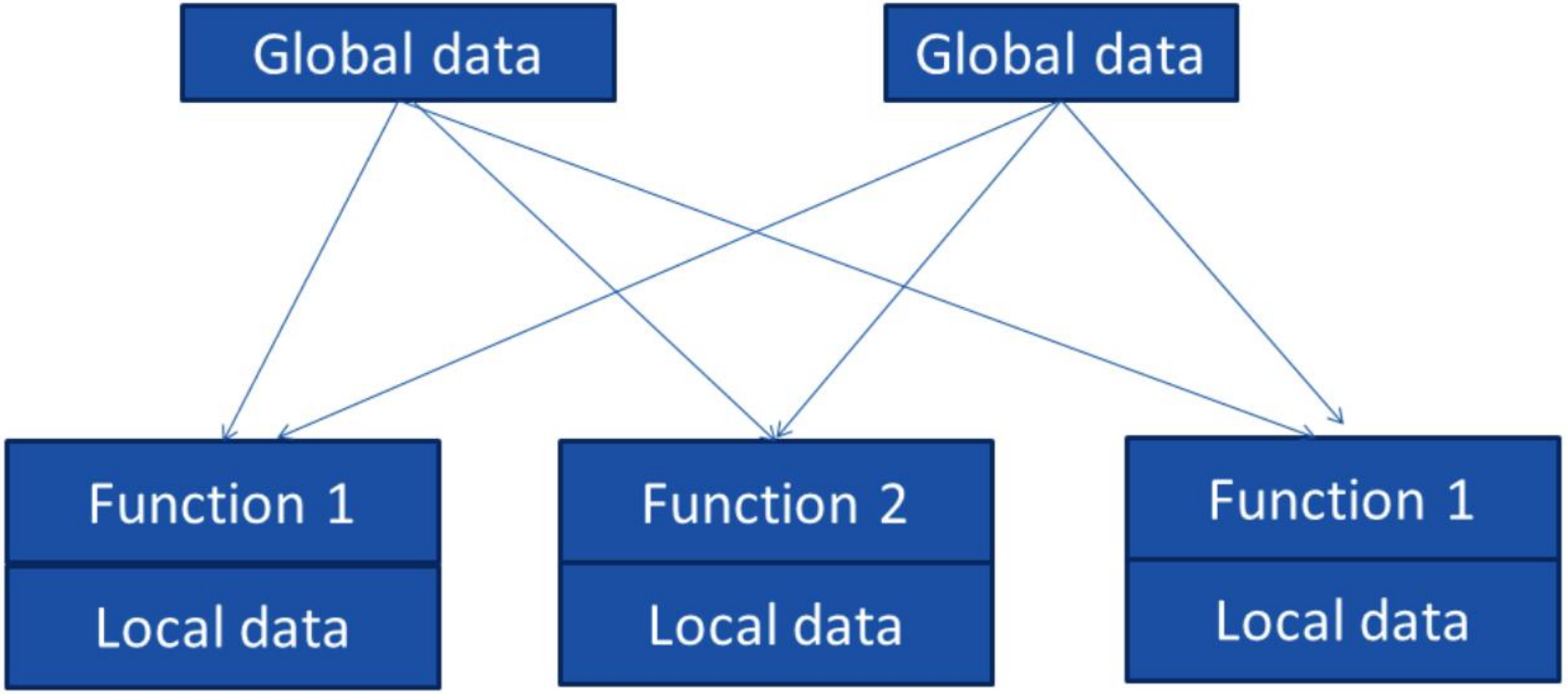
Introduction to OOP paradigm

- **Object-oriented** programming (**OOP**) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as **methods or functions**).

Introduction to Procedure-Oriented Programming

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

Procedure-Oriented Programming



Relationship of data and functions in POP

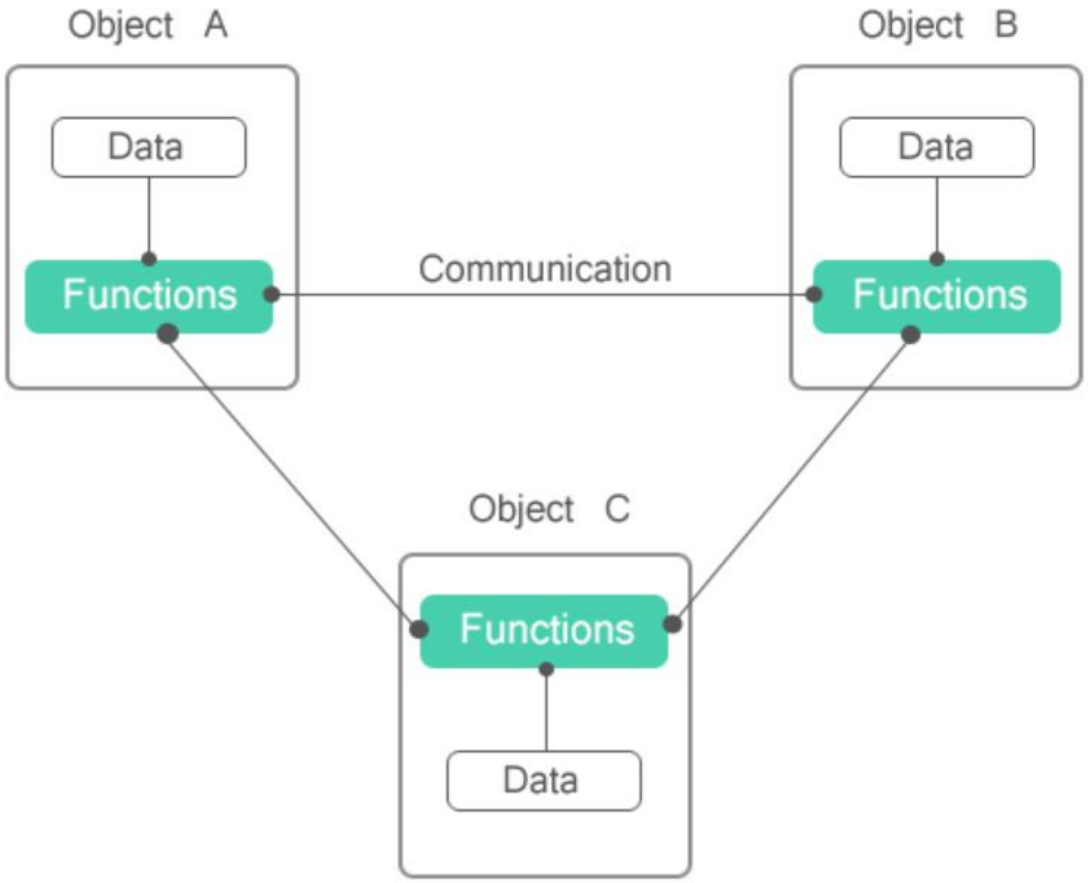
Limitations of Procedural Programming

- The program code is harder to write
- The Procedural code is often not reusable, which may pose the need to recreate the code if is needed to use in another application
- Difficult to relate with real-world objects
- The importance is given to the operation rather than the data, which might pose issues in some data-sensitive cases
- The data is exposed to the whole program, making it not so much security friendly

Object-Oriented Programming Benefits

- OOP mimics the real world, making it easier to understand
- OOP codes are reusable in other programs
- Security is offered due to the use of data hiding and abstraction mechanism
- Due to modularity and encapsulation, OOP offers ease of management

OOP Paradigms



Organization of Data and Functions in OOP

Fundamentals of object oriented Programming

Namespaces:

- A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it.
- Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.
- eg. using namespace std;

Here std is the namespace where ANSI C++ standard class libraries are defined.

Fundamentals of object oriented Programming

- Defining namespace

  syntax-

```
namespace namespace_name
{
        // code declarations
}
```

Fundamentals of object oriented Programming

```
#include <iostream>
using namespace std;
// first name space
namespace first_space
{
   void func() {
      cout << "Inside first_space" << endl;
   }
}
// second name space
namespace second_space
{
   void func() {
      cout << "Inside second_space" << endl;
   }
}
```

```
int main ()
{
   // Calls function from first name space.
   first_space::func();

   // Calls function from second name space.
   second_space::func();

   return 0;
}
```

Fundamentals of object oriented Programming

Objects:

- Objects are the basic run time entities in an object-oriented system.
- When a program is executed, the objects interact by sending messages to one another.
- Objects take up space in the memory.
- We can create ‘n’ number of objects belonging to particular class

Object Example

```
class example
{
        -------
        -------
};
int main()
{
        example obj;        //object of class example
}
```

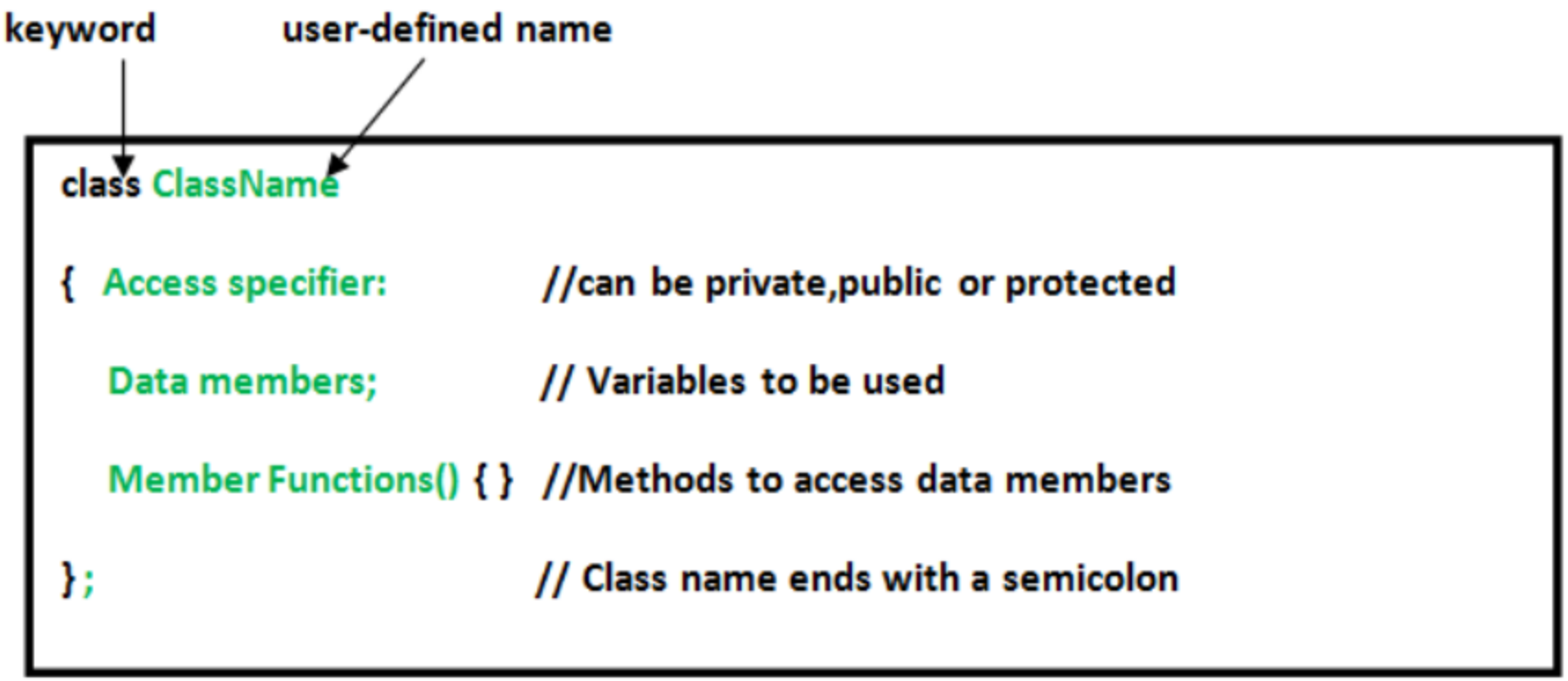
Fundamentals of object oriented Programming

Classes:

- A class in C++ is the building block, that leads to Object-Oriented programming.
- It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.
- A C++ class is like a blueprint for an object.

Fundamentals of object oriented Programming

Classes:



Fundamentals of object oriented Programming

Data Members:

- Data members include members that are declared with any of the fundamental types, as well as other types, including pointer, reference, array types, bit fields, and user-defined types.

Fundamentals of object oriented Programming

- **Data Members:**

```
#include<iostream>
using namespace std;
class student
{
    private :
        int id;
        char name[20];
    public :
        Void Getdata(void);
        Void display (void)
        {
            cout << id <<'\t' << name << endl;
        }
};
int main( )
{
```

Data Members

Member Functions

Fundamentals of object oriented Programming

Methods:

- **Methods** are functions that belongs to the class.
- There are two ways to define functions that belongs to a class:
  - Inside class definition.
  - Outside class definition.

Inside class definition

```
class example
{
        public:
                void add()                      //method defined inside class
                {
                        ------
                        ------
                }
};

Int main()
{
        example obj;
        obj.add();
        return 0;
}
```

Outside class definition

```
class example
{
        public:
                void add()                      //method declared inside class
};

void example::add()                     //method defined outside class
{
        ------
        ------
}

Int main()
{
        example obj;
        obj.add();
        return 0;
}
```

Fundamentals of object oriented Programming

Messages:

- 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.

e.g. employee.salary(name)

(object) (Message) (information)

Fundamentals of object oriented Programming

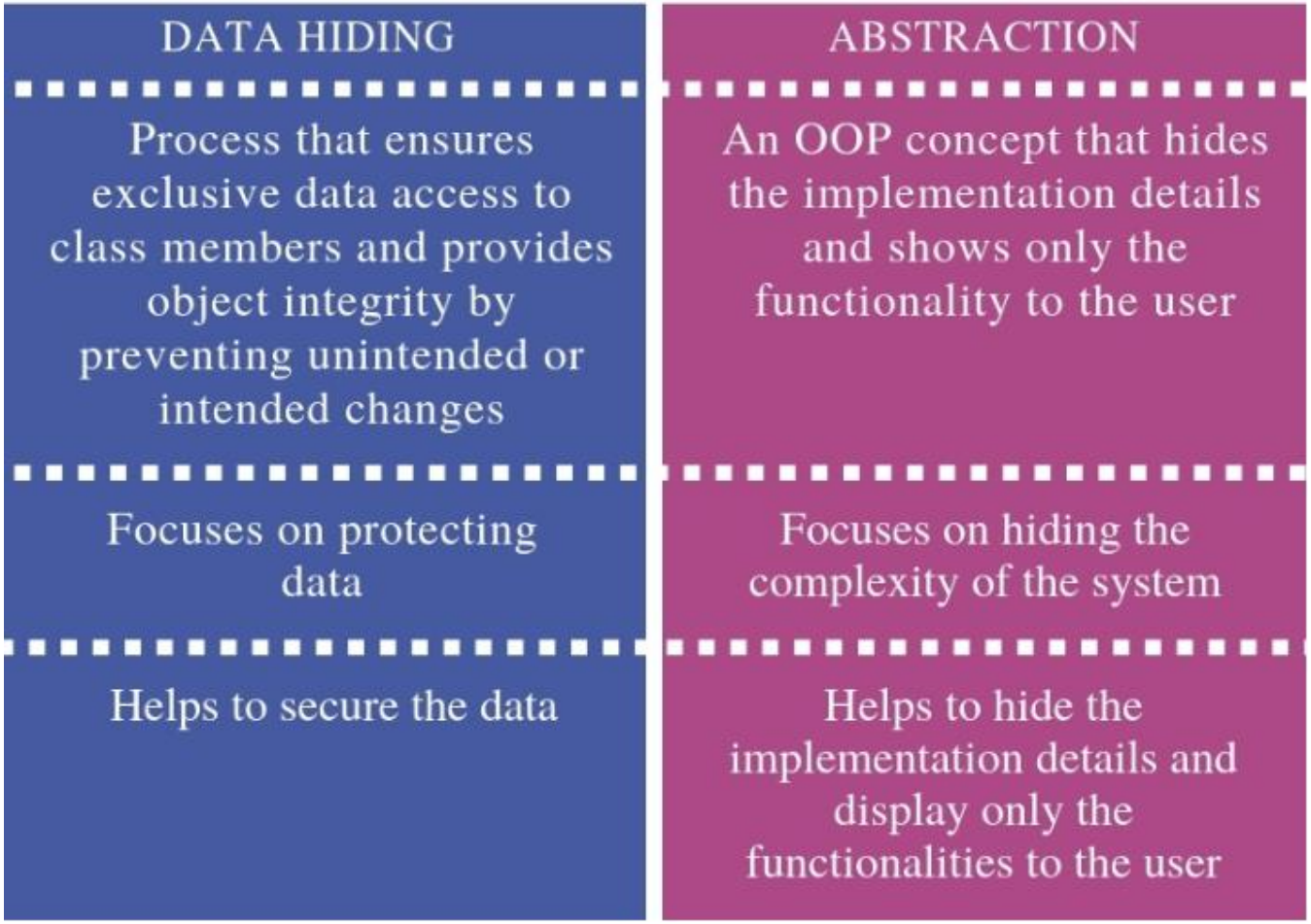
Data Encapsulation:

- **Encapsulation** is an Object Oriented Programming concept that binds together the **data** and functions that manipulate the **data**, and that keeps both safe from outside interference and misuse.

Fundamentals of object oriented Programming

- **Data Abstraction** refers to providing only essential **information** to the outside world and **hiding** their background details, i.e., to represent the needed **information** in program without presenting the details.
- **Data hiding** is an object-oriented programming technique of **hiding** internal object details i.e. **data** members.

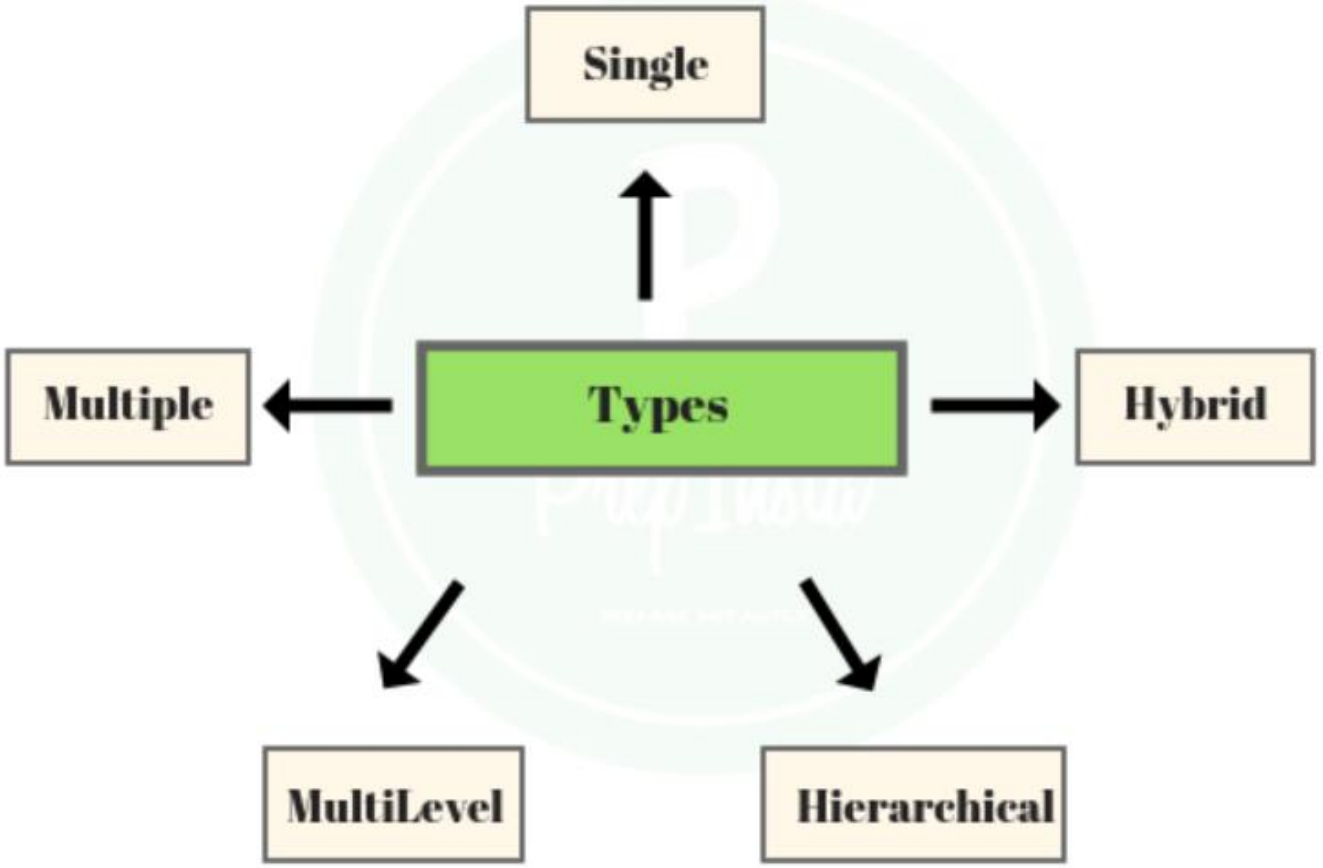
Fundamentals of object oriented Programming



Fundamentals of object oriented Programming

- **Inheritance** is a process in which one object acquires all the properties and behaviors of its parent object automatically.
- In C++, the class which inherits the members of another class is called derived class and the class whose members are **inherited** is called base class.

Types Of Inheritance



Fundamentals of object oriented Programming

- Inheritance Example

```
// Base class
class Vehicle
{
   public:
   string brand = “Tata";
   void honk()
  {
    cout << "Tuut, tuut! \n" ;
   }
};
```

```
// Derived class
class Car: public Vehicle
{
  public:
    string model = “Altroz";
};
int main()
{
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " +  myCar.model;
  return 0;
}
```

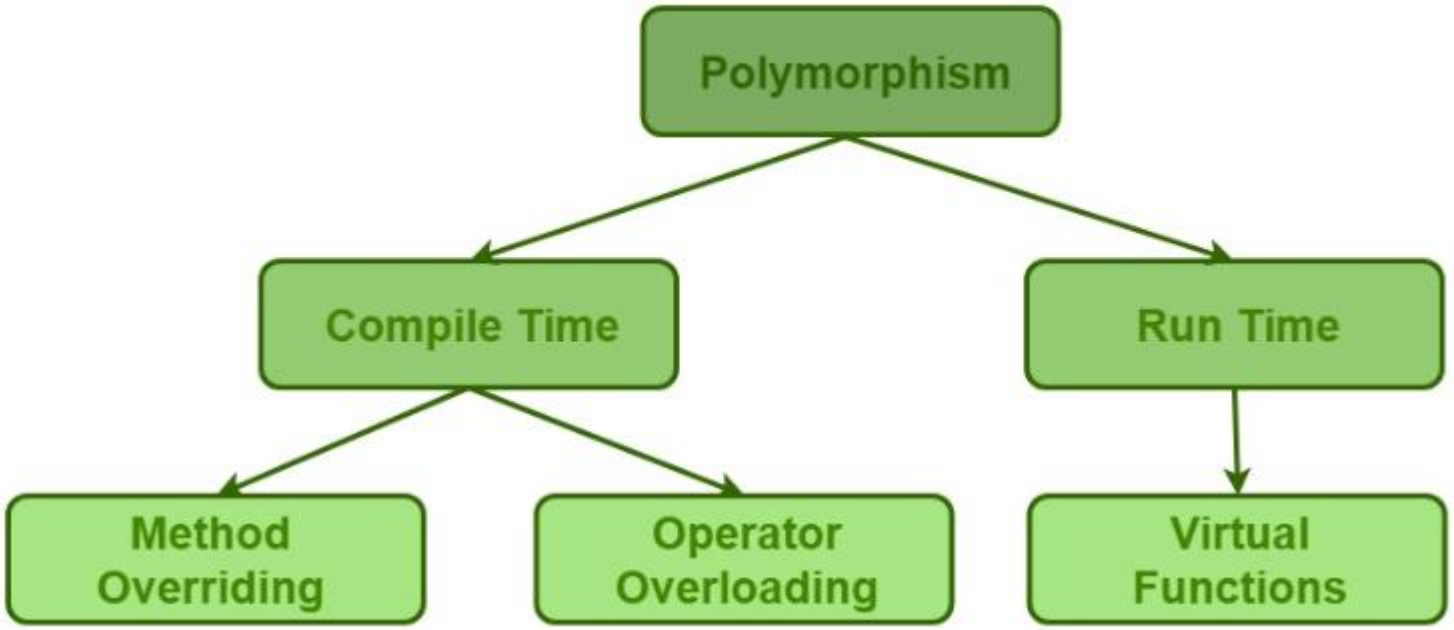
Fundamentals of object oriented Programming

Polymorphism:

- The word polymorphism means having many forms.
- In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism.
- It takes place when there is a hierarchy of classes and they are related by inheritance.

Fundamentals of object oriented Programming

Types of Polymorphism



Fundamentals of object oriented Programming: Polymorphism

```
class Shape {
  protected:
    int width, height;

  public:
    Shape( int a = 0, int b = 0){
      width = a;
      height = b;
    }
    int area() {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
```

```
class Rectangle: public Shape {
  public:
    Rectangle(int a = 0, int b = 0):Shape(a, b) { }

    int area () {
          return (width * height);
    }
};

class Triangle: public Shape {
  public:
    Triangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
          return (width * height / 2);
    }
};
```

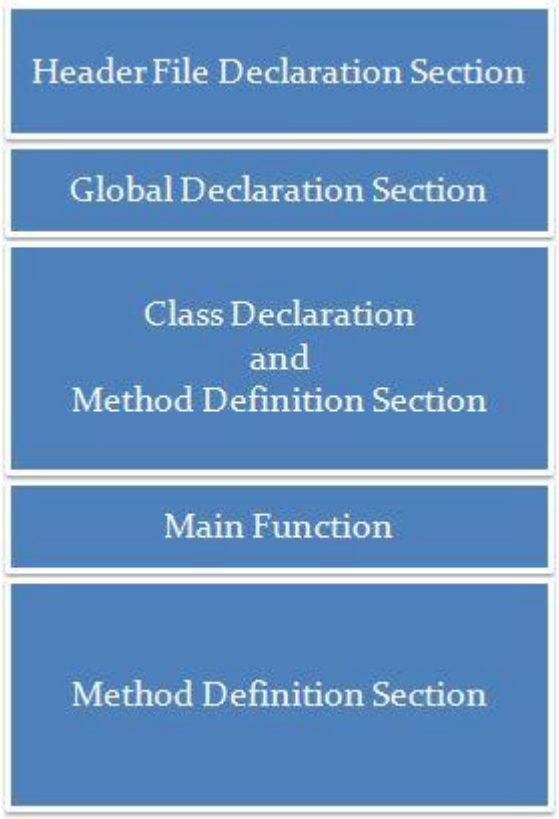
Benefits of OOP

- Modularity for easier troubleshooting
- Reuse of code
- Flexibility through polymorphism
- Effective problem solving
- Security
- Code maintenance

C++ As OOP Language

- C++ is called object oriented programming (OOP) language because C++ language views a problem in terms of objects involved rather than the procedure for doing it.
- Object oriented programming is always good for writing large business logics and large applications or games.
- OOPs is also very much desired for maintenance and long term support.

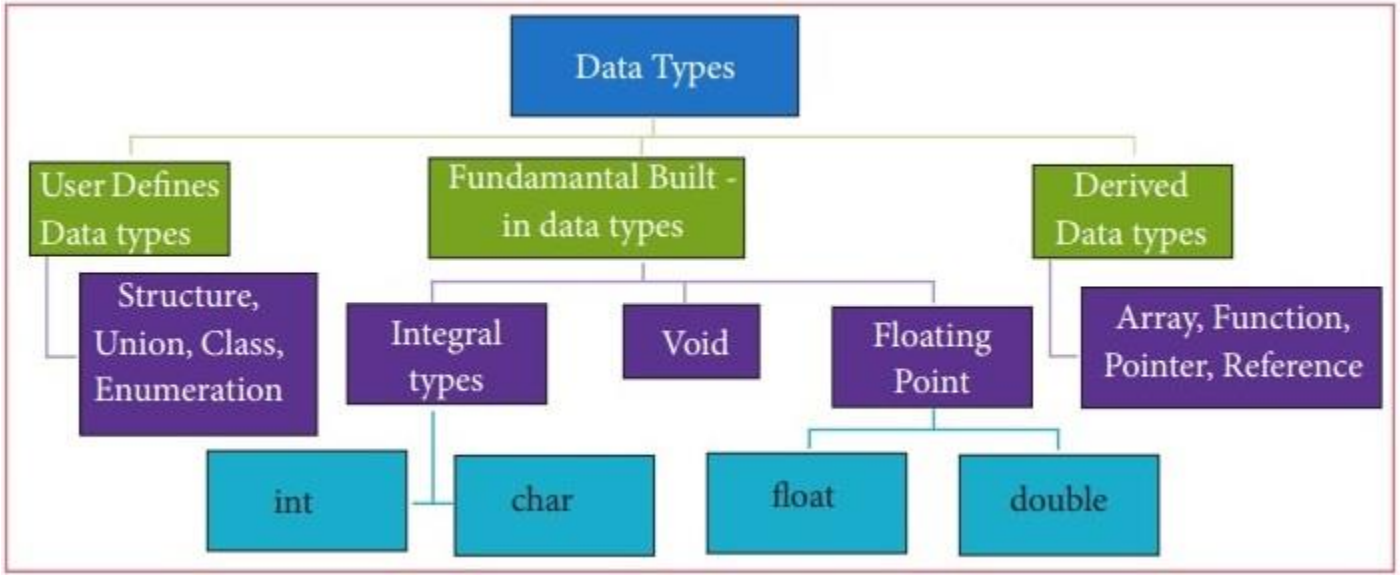
C++ Program structure



C++ Programming Basics

```
 using namespace std;
Class Example
{
        //Class  declaration, if any along with its member variables and functions
};
//global variables, if any
int main()                  //This is where the execution of program begins
{
         //code
        return 0;
 }
```

C++ Data Types


Data Types
User Defines Data types
Fundamantal Built - in data types
Derived Data types
Structure, Union, Class, Enumeration
Integral types
Void
Floating Point
Array, Function, Pointer, Reference
int
char
float
double

More about Built-in Data Types

Type	Size (in bytes)	Range
char	1	-127 to 127 or 0 to 255
unsigned char	1	0 to 255
int	4	-2147483648 to 2147483647
unsigned int	4	0 to 4294967295
short int	2	-32768 to 32767
unsigned short int	2	0 to 65,535
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	+/- 3.4e +/- 38 (~7 digits)
double	8	+/- 1.7e +/- 308 (~15 digits)

Derived Type

- These data types are derived from fundamental data types.
- Variables of derived data type allows us to store multiple values of same type in one variable but never allows to store multiple values of different types.
- e.g. Array of integer values

```
int string[10];
```

- Function
- Pointer
- Reference

User-defined data types

- Structure - defines a new data type, with more than one member variable

  e.g.

```
struct Books
{
        char title[50];
        char author[50];
        char subject[100];
        int book_id;
};
```

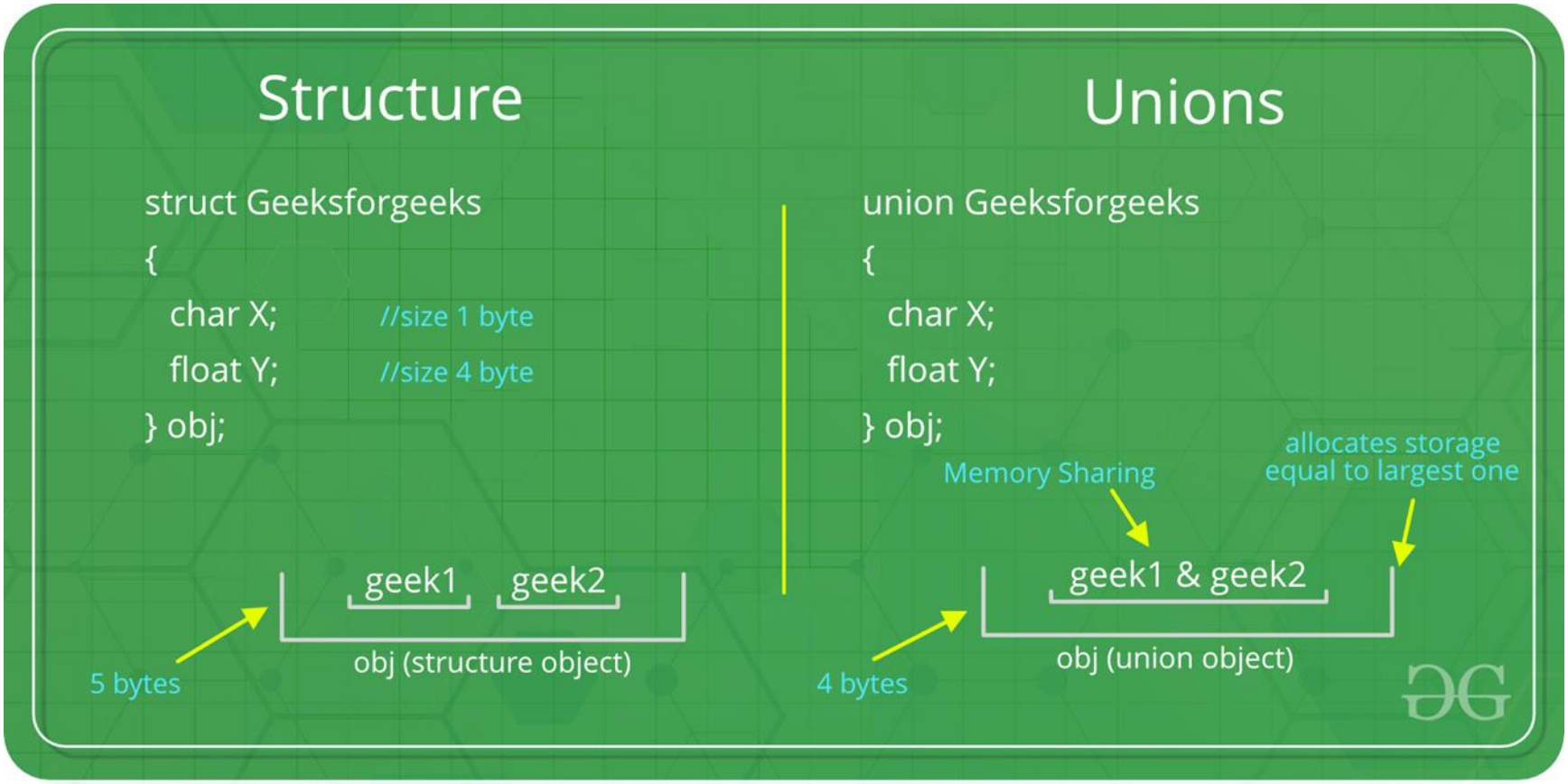
Union

- Union - user-defined type that uses same block of memory for each of its list member

e.g.

```
union student
{
   int roll_no;
   int phone_number;
   float percentage;
};
```

Structure Vs Union



Example of Union

```
#include<iostream>
#include<cstring>
using namespace std;
union student
{
        int roll_no;
        int phone_number;

};
int main()
{
        union student p1;
        p1.roll_no=12;
        p1.phone_number=1234567822;

        cout<<"roll_no:"<<p1.roll_no<<endl;
        cout<<"phone_number:"<<p1.phone_number<<endl;

        return 0;
}
```

Output:

roll_no:1234567822

phone_number:1234567822

User-defined data types

- Class - defines a new data type, with more than one member variables and member functions.

  e.g.

```
class Books
{
        char title[50];
        char author[50];
        public:
                void get_book();
                void show_book();
};
```

Enumerations

- Enumerations are used to define symbolic constants.

e.g.

```
enum direction {East, West, North, South};
int main()
{
    direction dir = North;
    cout<<dir<<endl;
    cout<<East;
    return 0;
}
```

Output: 2
0

Enumerations

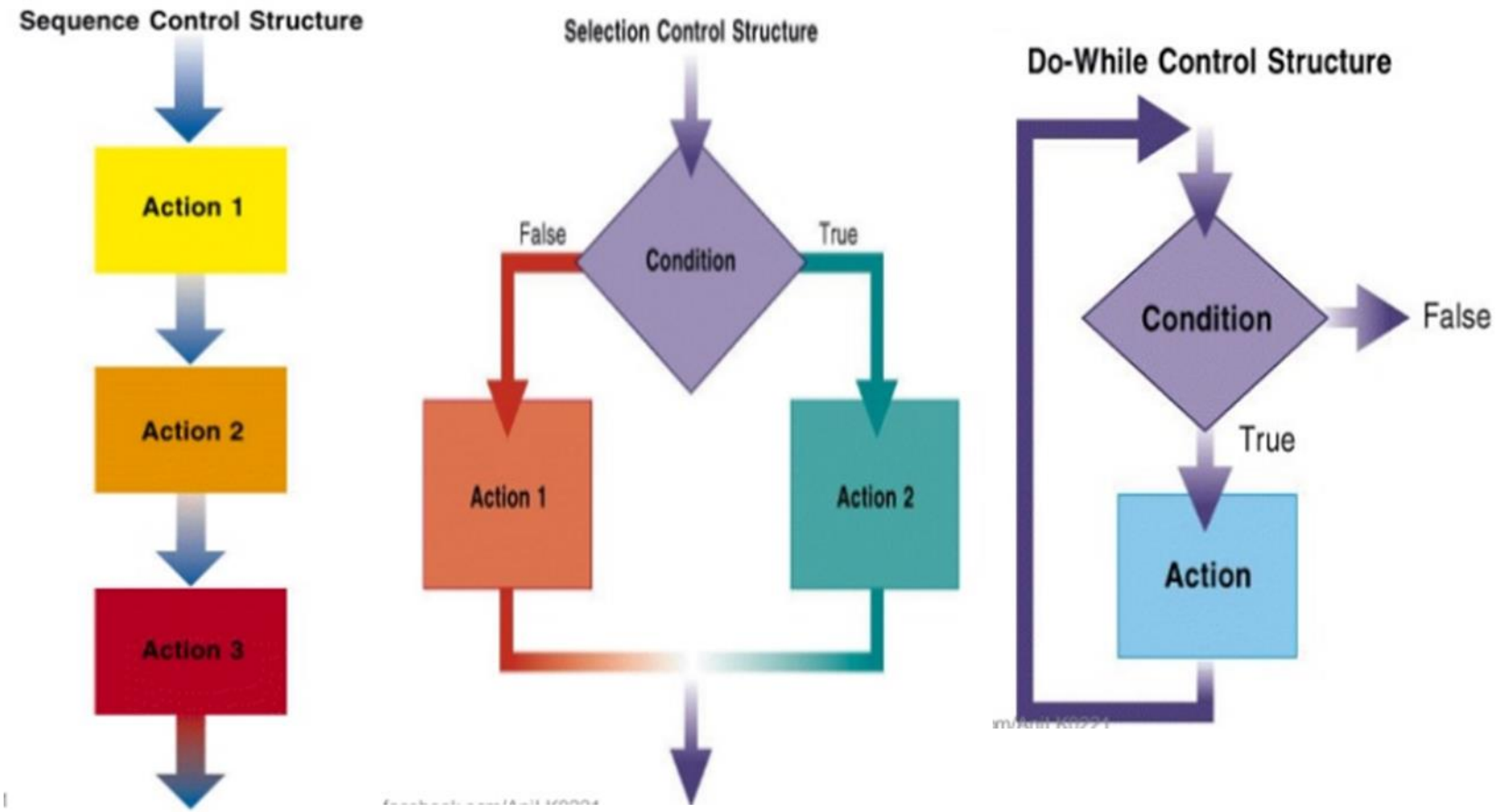
```
int main()
{
	// Defining enum Gender
	enum Gender { Male, Female };

	// Creating Gender type variable
	Gender gender = Male;
	switch (gender)
	{
		case Male: cout << "Gender is Male";  break;
		case Female: cout << "Gender is Female"; break;
		default:   cout << "Value can be Male or Female";
	}
	return 0;
}
```

Control Structures

- C++ has only three kinds of control structures, which we refer to as control statements:
- Sequence statement

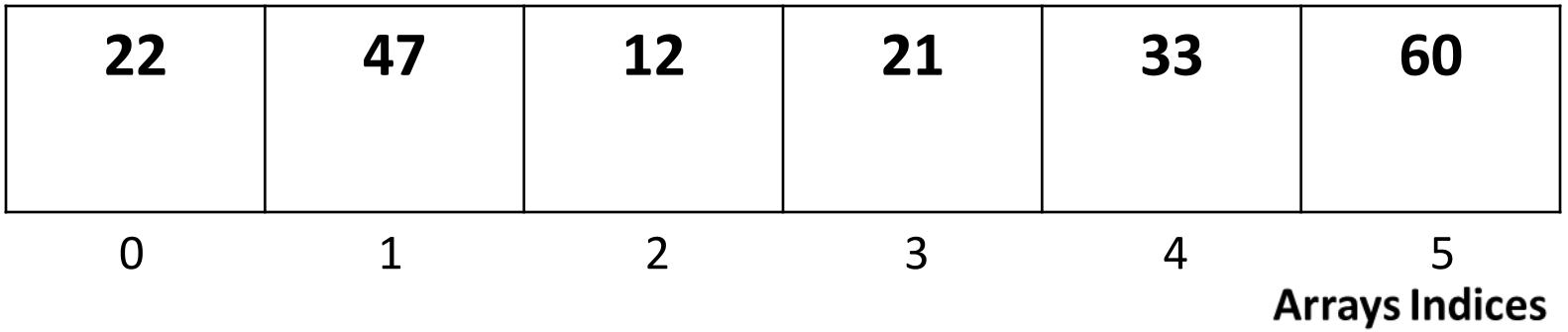
  “sequence is a series of statements that executes one after another”
- Selection statements
  - If
  - if...else and
  - switch
- repetition statements
  - While
  - for and
  - do... while

Control structures



Arrays and Strings

- Arrays - An array in C++ is a collection of items stored at contiguous memory locations and elements can be accessed randomly using indices of an array.



Array Length : 6
First Index: 0
Last Index: 5

Strings

Strings represent sequences of characters

```
#include <iostream>
using namespace std;
int main ()
{
    char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
    cout << "Greeting message: ";
    cout << greeting << endl;
    return 0;
}
```

Output- Greeting message: Hello

Class

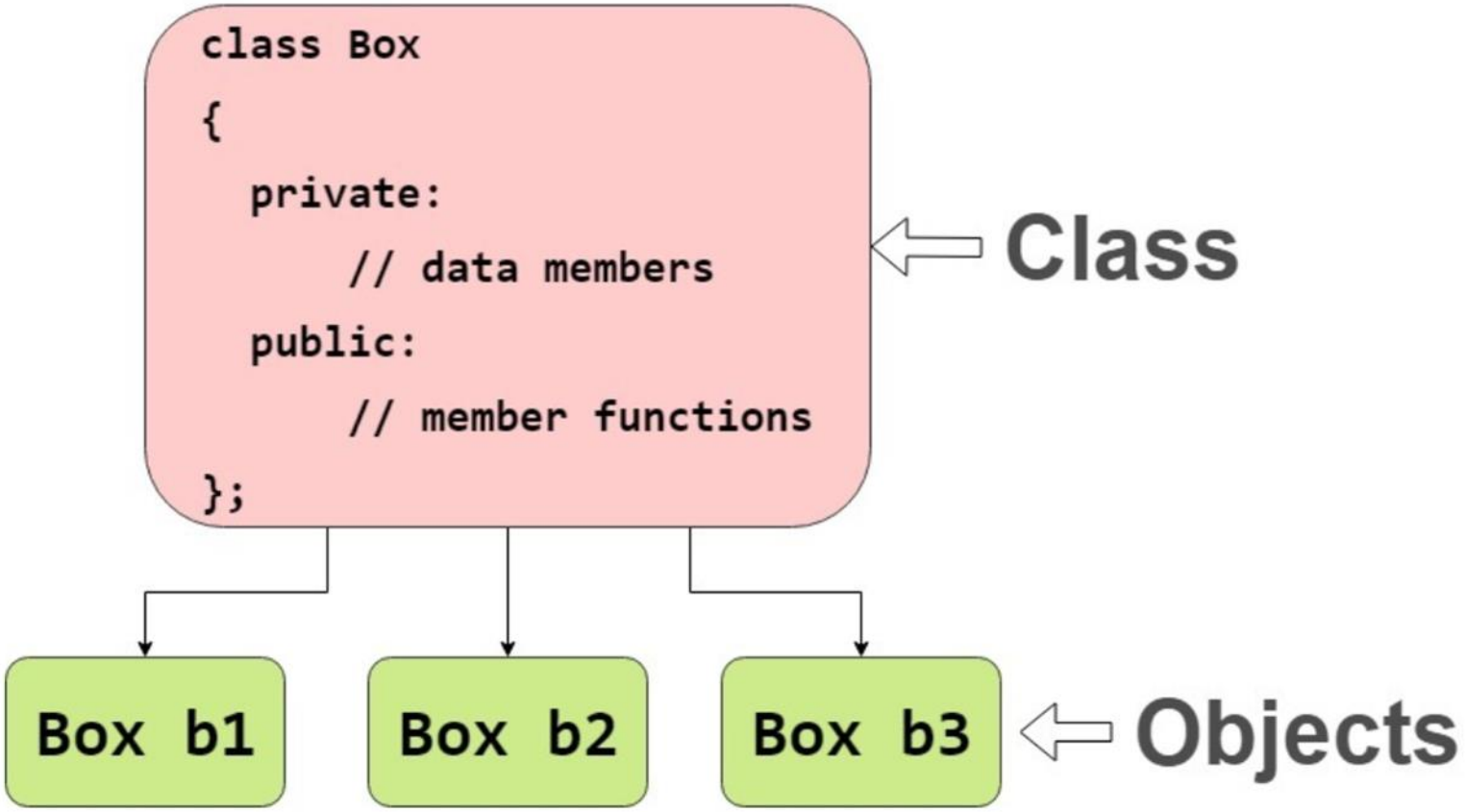
- Class - It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

```
class Books
{
        char title[50];
        char author[50];
        public:
                void get_book();
                void show_book();
};
```

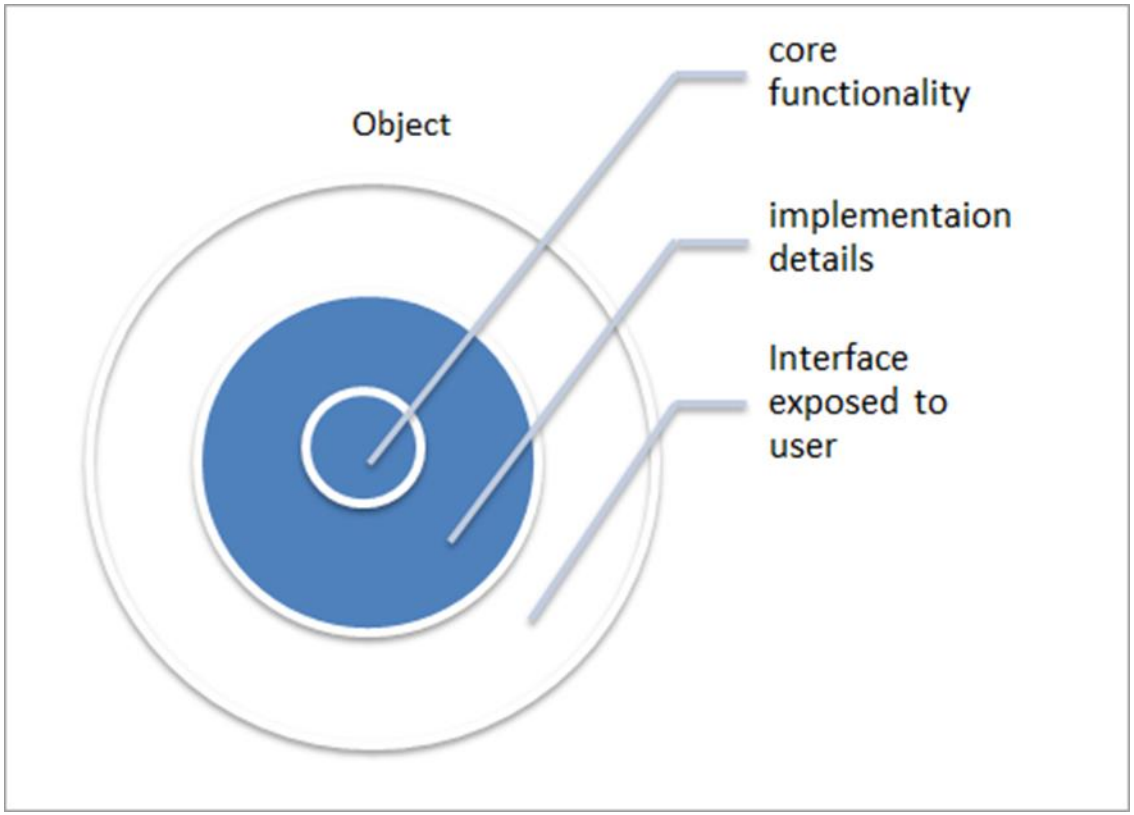
Object

- Everything in C++ is associated with classes and objects, along with its attributes and methods.
- For example: in real life, a car is an **object**. The car has attributes, such as weight and color, and methods, such as drive and brake.

Class & Object



Data Abstraction



Abstraction means displaying only essential information and hiding the details.

Class & Data Abstraction

```
#include <iostream.h>
using namespace std;
class AbstractionExample
{
private:
        int num;
        char ch;
public:
void setMyValues(int n, char c)
{
        num = n;
        ch = c;
}
void getMyValues()
{
cout<<"Numbers is: "<<num<< endl;
cout<<"Char is: "<<ch<<endl;
}
};

int main()
{
        AbstractionExample obj;
        obj.setMyValues(100, 'X');
        obj.getMyValues();
        return 0;
 }
```

Advantages of Abstraction

- The programmer need not write low-level code.
- It protects the internal implementation from malicious use and errors.
- The Programmer can change the internal details of the class implementation without the knowledge of end-user thereby without affecting the outer layer operations.

Accessing Class Members

- The data members and member functions of class can be accessed using the dot('. ') operator with the object.

Accessing Class Members

```
class MyClass
    {
        public:                         // Access specifier
            int myNum;                  // Attribute (int variable)
            string myString;            // Attribute (string variable)
    };

int main()
    {
        MyClass myObj;                  // Object of MyClass

        // Access attributes and set values
        myObj.myNum = 15;
        myObj.myString = "Some text";

        return 0;
    }
```

Access Specifiers

public

```
class PublicAccess
{
 public:
        int x;
        void display();
}
```

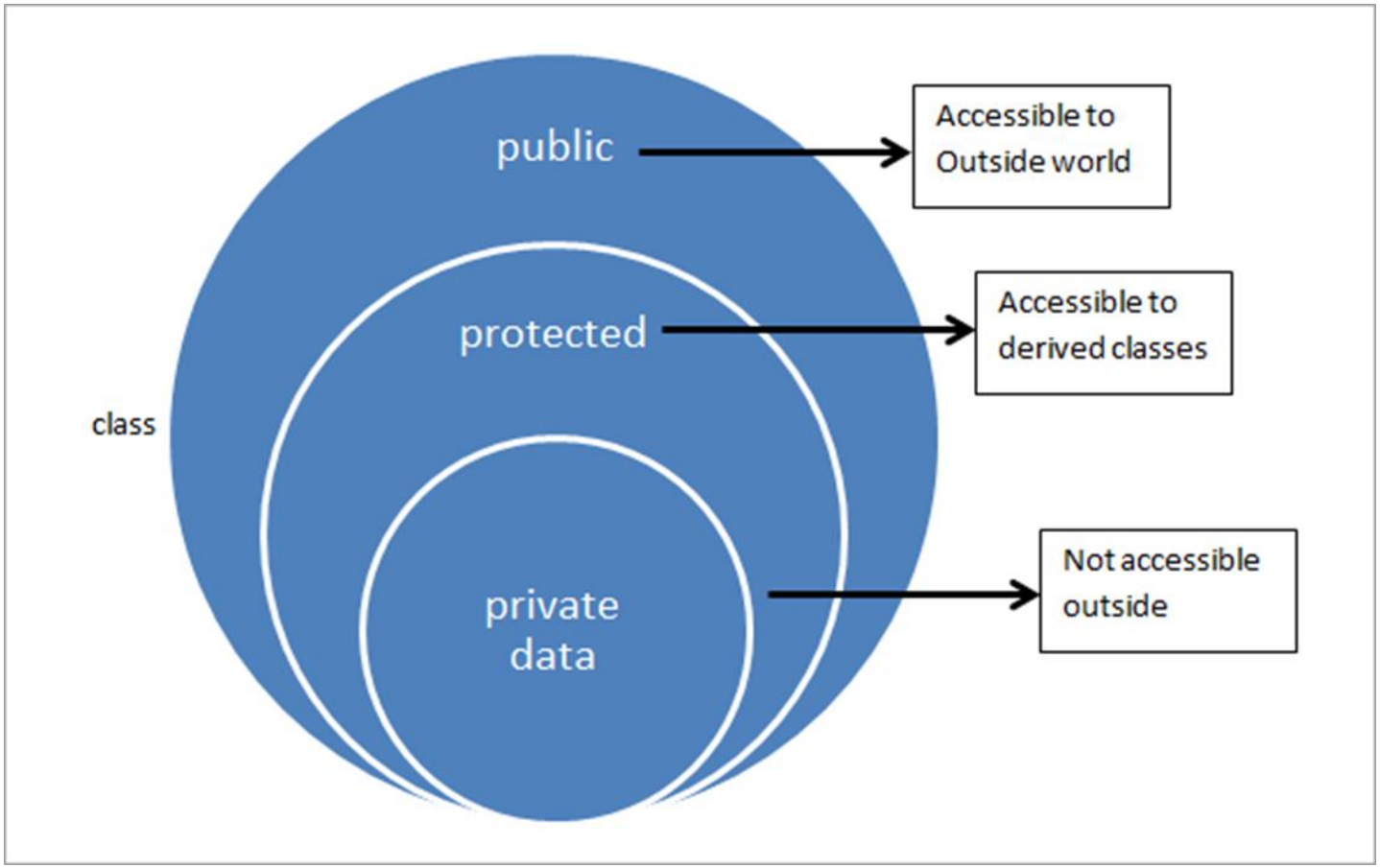
private

```
class PrivateAccess
{
 private:
        int x;
        void
display();
 }
```

protected

```
class ProtectedAccess
{
 protected:
        int x;
        void display();
}
```

Access Specifiers



Separating Interface from Implementation

- Class declaration is the abstract definition of interface.
- The functions include the details of the implementation.
- It is possible to separate interface from the implementation by placing them in separate files.
- The class declaration goes in a header file with a .h extension whereas implementation goes in the .cpp file which must include the user defined header file.

Interface Vs Implementation

- Interface – the declaration of the class
- Implementation – the definition of the member functions
- interface and implementation are typically separated into different files (also separate from main driver file)

- For Rectangle Class
  - Interface – Rectangle.h
  - Implementation – Rectangle.cpp

Functions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Functions

```
// function creation

void myFunction()
{
        cout << "I just got executed!";
}

int main()
{
        myFunction();                   // call the function
        return 0;
}
```

Output:
"I just got executed!"

Function Prototype

- Function prototype serves the following purposes:

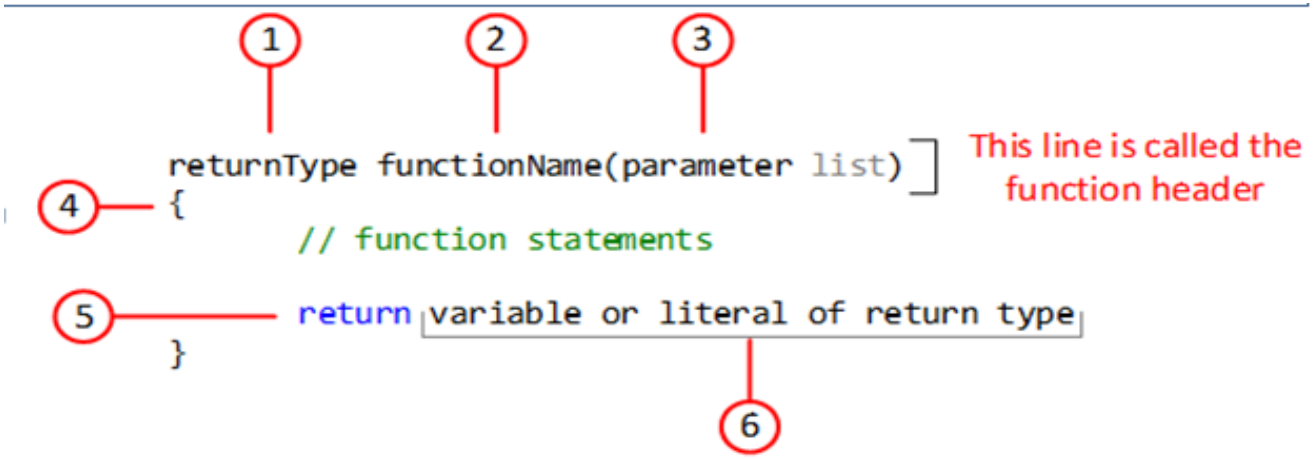
  1) It states the return type of the data that the function will return.

  2) It states the number of arguments passed to the function.

  3) It states the data types of the each of the passed arguments.

  4) It states the order in which the arguments are passed to the function.

Function Prototype




```
int add(int a, int b);  //Function Prototype

void add(int a, int b) //Function Defination
{
  return a+b;
}
```

Constructors

- A constructor is a special method that is automatically called when an object of a class is created.
- To create a constructor, use the same name as the class, followed by parentheses ().
- Constructor doesn't have a return type.
- One can pass parameters to the constructors.
- Constructors are of different types: default, parameterized, copy.

Constructors

```
class MyClass                          //Class
{
  public:
            MyClass()                  //Constructor
            {
                        cout << "Hello World!";
            }

};

int main()
{
            MyClass myObj;    // object of MyClass will call the constructor
            return 0;
}
```

Destructors

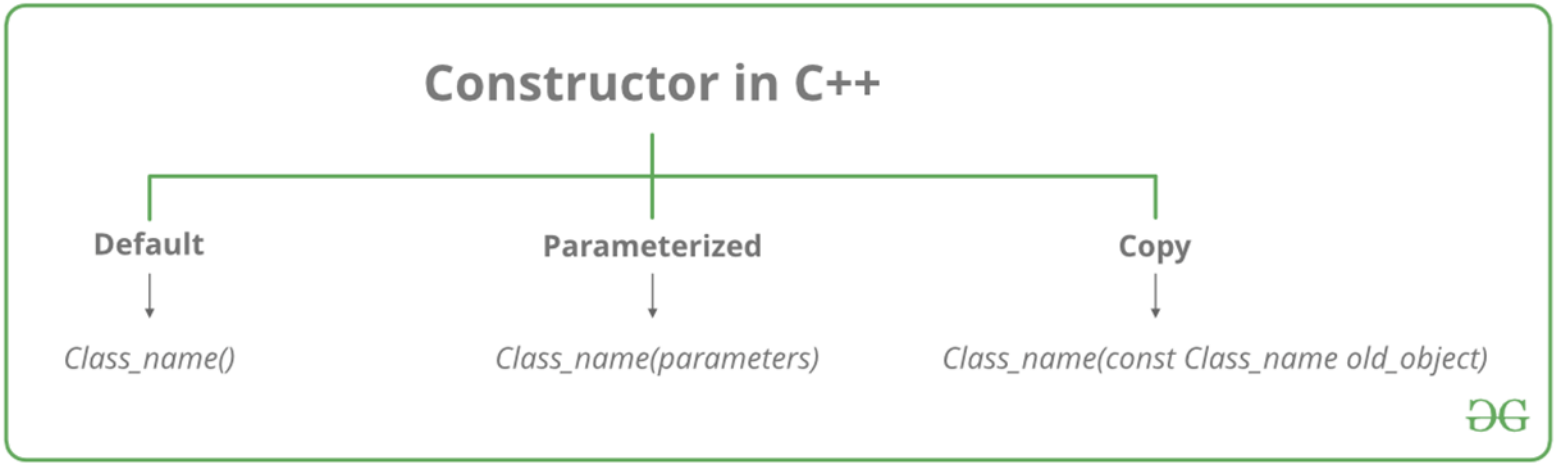
- A destructor is a member function that is invoked automatically when the object goes out of scope or is explicitly destroyed.
- A destructor has the same name as the class, preceded by a tilde (~).
- For example, the destructor for class String is declared: ~String() .

Destructors

```
class Demo
{
        private:
        int num1, num2;
        public:
        Demo(int n1, int n2)
         {
                cout<<"Inside Constructor"<<endl;
                num1 = n1;
                num2 = n2;
        }
        ~Demo()
        {
                cout<<"Inside Destructor";
        }
};
```

```
int main()
{
        Demo obj1(10, 20);
        return 0;
}
```

Types of Constructor



Default Constructor

Default constructor is called with no arguments.

```
class MyClass
{
  public:
            MyClass()            // default Constructor
            {
                        cout << "Hello World!";
            }
};
int main()
{
            MyClass myObj;
            return 0;
}
```

Parameterized Constructor

- It accepts a specific number of parameters. To initialize data members of a class with distinct values.
- With a parameterized constructor for a class, one must provide initial values as arguments, otherwise, the compiler reports an error.

Parameterized Constructor

This constructor is called with arguments.

```
class MyClass
{
  public:
        MyClass(int p)  // parameterized Constructor
        {
                -------
                -------
        }
};
int main()
{
        MyClass myObj(10);
        return 0;
}
```

Copy Constructor

- Used to create new object as a copy of existing object.
- A copy constructor has the following general function prototype:

```
ClassName (const ClassName &old_obj);
```

Copy Constructor

```
class Point
{
private:
        int x, y;
public:
        Point(int x1, int y1)
        {
                x = x1;
                y = y1;
        }
        // Copy constructor
        Point(const Point &p)
        {
                x = p.x;
                y = p.y;
        }
};
```

```
int main()
{
        Point p1(10, 15);
        Point p2 = p1; // Copy constructor
        return 0;
}
```

Objects & Memory Requirements

- The memory space is allocated to the data members of a class only when an object of the class is created, and not when the data members are declared inside the class.
- Since a single data member can have different values for different objects at the same time, every object declared for the class has an individual copy of all the data members.

Static Members: Variables & Functions

- A static member function can be called even if no objects of the class exist
- Static functions are accessed using only the class name and the scope resolution operator ::.
- A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static Members

```
class Test
{
private:
static int n;
public:
static void show()
{
cout<<"n = "<<n<<endl;
}
};
```

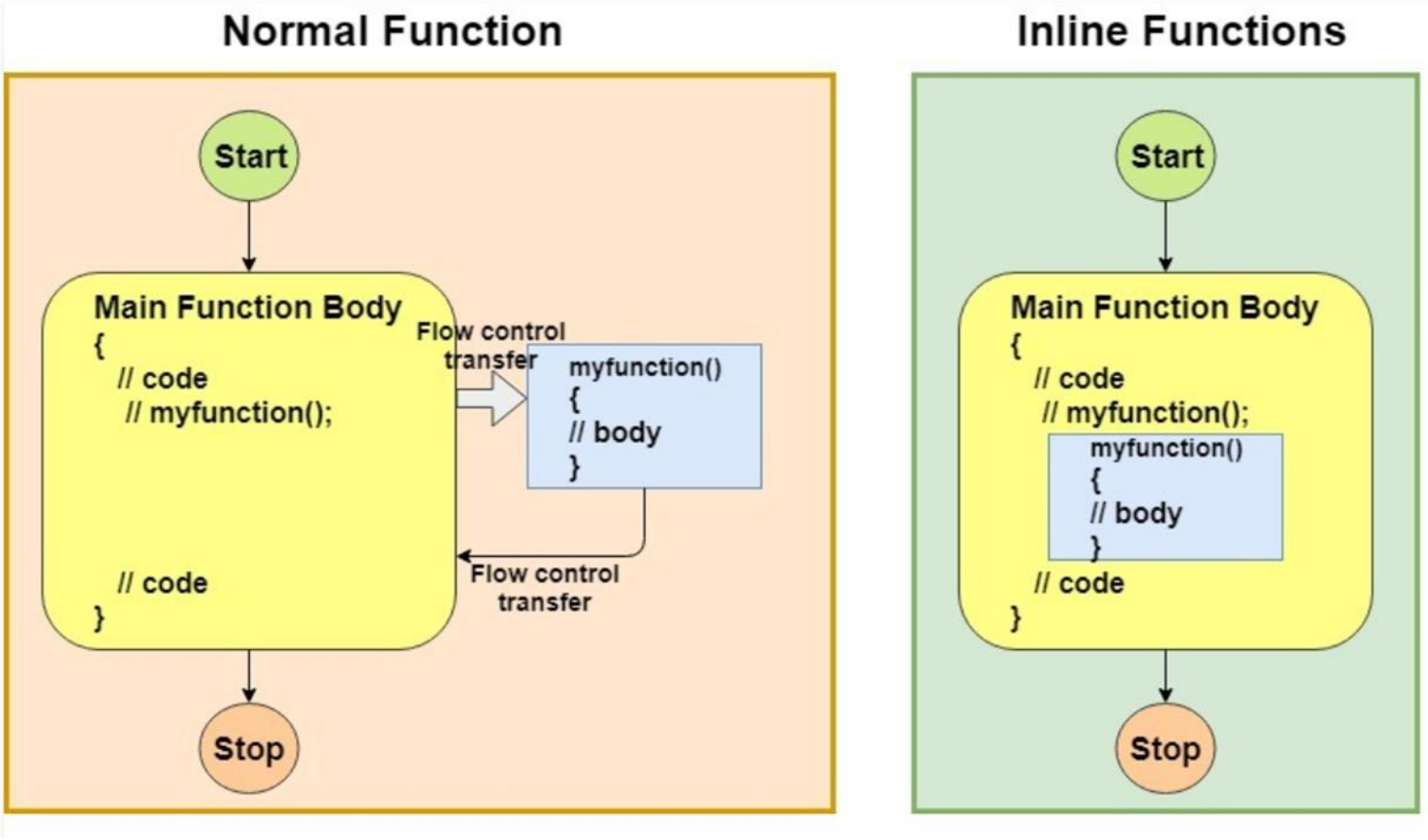
```
int Test::n=10;
void main()
{
Test::show();
getch();
}
```

Output:
n = 10

Inline Function

- The inline functions are the C++ enhancement feature to improve the execution time of a program.
- These functions can instruct the compiler to make them inline so that compiler can replace the function calls by the function definitions.
- Function call overhead doesn’t occur.

Inline Function



Inline Function

```
inline int cube(int s)
{
   return s*s*s;
}
int main()
{
   cout << "The cube of 3 is: " << cube(3) << "\n";
   return 0;
}
//Output: The cube of 3 is: 27
```

Friend Function

- A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.
- Even though the prototypes for friend functions appear in the class definition, friends are not member functions of the class in which they are declared.

Friend Function

```
class Temperature
{           int celsius;
public:
            Temperature()
            {    celsius=0;
            }
            friend int temp(Temperature);
};

int temp(Temperature t)
{           t.celsius=40;
            returnt.celsius;
}
```

```
int main()
{
            Temperature tm;
            cout<<temp(tm)<<endl;
            return 0;
}
```

Modular Programming

- **Modular programming** is the process of subdividing a computer **program** into separate sub-programs.
- A module is a separate software component. It can often be used in a variety of applications and functions with other components of the system.

Generic Programming

- **Generics** in **C++** is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces.
- **Generic Programming** enables the programmer to write a general algorithm which will work with all data types.

Difference Between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP) 4M

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Story of C++ Invention by Bjarne Stroustrup: Case Study

Why I Created C++

https://www.youtube.com/watch?v=JBjjnqG0BP8

The Essence of C++

https://www.youtube.com/watch?v=86xWVb4XIyE