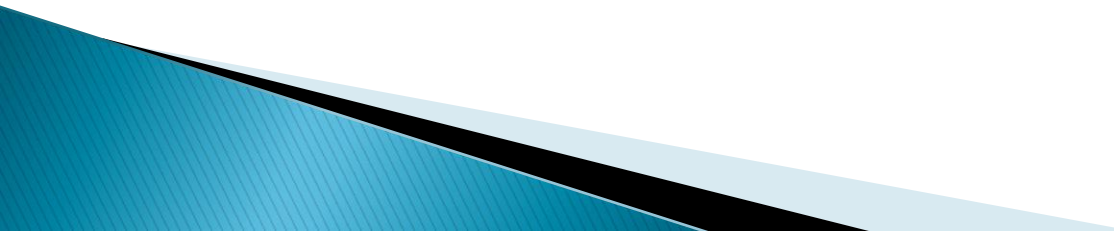


Object-Oriented Programming Concepts with C++


Prepared By:

Dr. Bakshi Rohit Prasad
(MTech, PhD, IIIT-Allahabad)

Object Oriented Programming


- ▶ A methodology of programming
 - ▶ Following major principles:
 1. Data Abstraction.
 2. Encapsulation.
 3. Information Hiding.
 4. Polymorphism (dynamic binding).
 5. Inheritance.
- 

The C++ language

- ▶ An object-oriented, general-purpose programming language, derived from C.
 - ▶ C++ adds the following to C:
 1. Inlining and overloading of functions.
 2. Default argument values.
 3. Argument pass-by-reference.
 4. **Operator new** and **delete**, instead of **malloc()** and **free()**
 5. **Operator Overloading** and **Stream I/O**
 6. Object-oriented programming via concepts of objects, classes, information hiding, encapsulation, polymorphism, inheritance.
- 

Object–Oriented Programming

Object–oriented programming paradigm is characterized by the following major concepts:

1. **Data Abstraction:** Knowing only essential details while hiding rest of details.
 2. **Encapsulation:** Combining data and operations on it in a single unit.
 3. **Information hiding:** The ability to selectively hide implementation details.
 4. **Polymorphism:** One name multiple purpose/work.
 5. **Inheritance:** The ability of objects of one class, to inherit properties of another class.
- 

OO Principles and C++ Constructs

OO Concepts

Data Abstraction

Encapsulation

Information Hiding

Polymorphism

Inheritance

C++ Construct(s) Used

Classes / Objects


Classes / Objects

Access specifiers; public, private, protected

Operator overloading,
function overloading, etc.

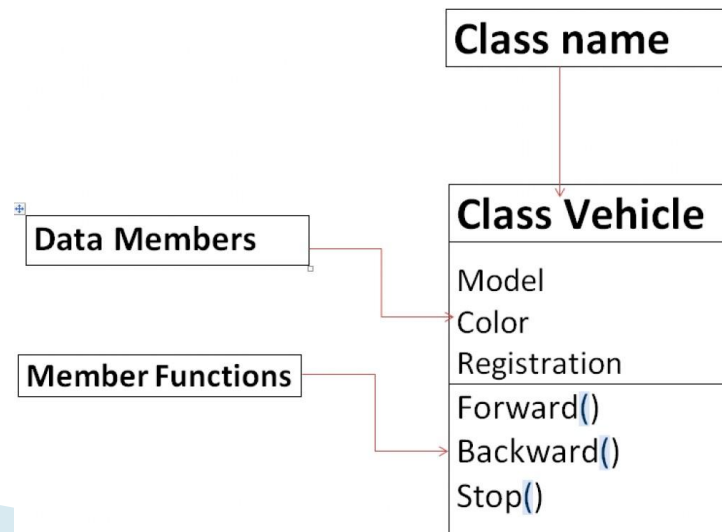
Derived Classes

Encapsulation

- ▶ *Encapsulation* refers to mechanisms that combine data and operations specified on that data into a single unit
 - ▶ Thus, it allows each object to have its own data and methods.
 - ▶ Object oriented languages provide powerful and flexible encapsulation mechanisms for restricting interactions between various software components so that data and functionality may be used only by required components in a system.
 - ▶ It has a significant impact on the ease of understanding, testing, and maintenance of components.
- 

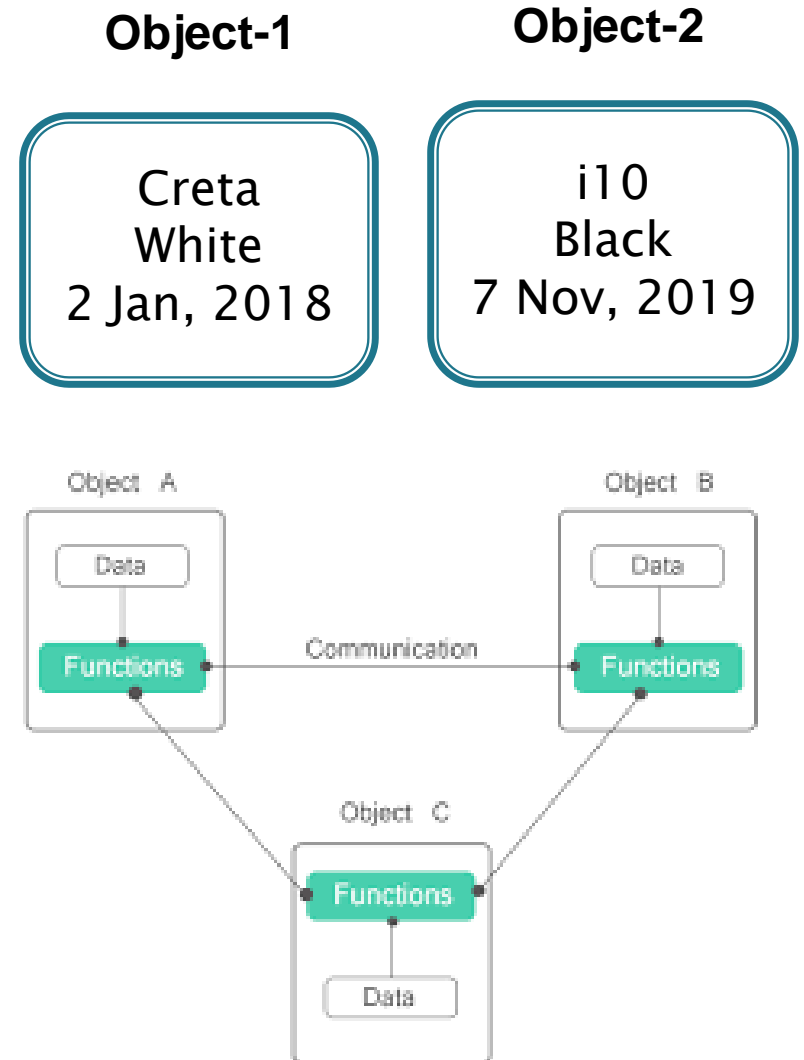
Classes

- ▶ Class is like a blueprint for objects which encapsulates the data members (attributes) and member functions (methods) into a single unit.
- ▶ In programming languages like C++, Java, etc., it is considered as a user-defined data type which can be used to create one or more instances (objects) of that class.

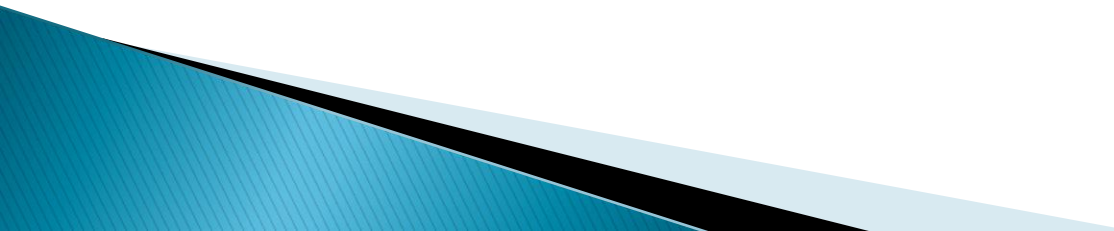


Objects

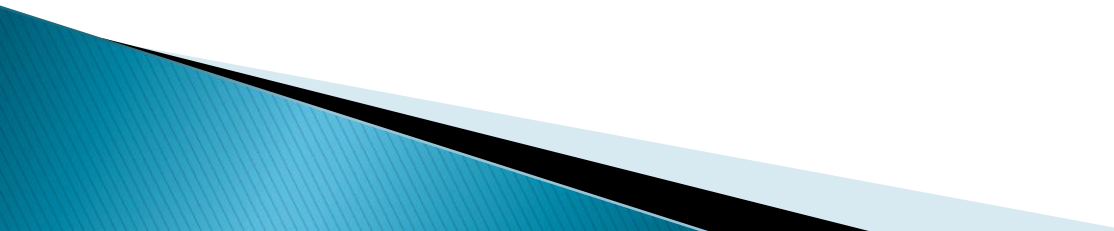
- ▶ An **object** is an encapsulation of data together with procedures/functions that manipulate the data and/or return information about the data.
- ▶ In programming languages, it is considered to be an instance of a class of that type.
- ▶ The procedures/functions are usually called as '**methods**' that provides mechanism to communicate with each other.
- ▶ A method call is called as '**message**' and the object on which it is invoked is called as the '**receiver**' of the message.




Information or Data Hiding

- ▶ Information or data hiding is a programming concept which protects the data from direct modification by other parts of the program.
 - ▶ It protects data or information from any inadvertent change throughout the program.
 - ▶ In most of the object programming language, it is achieved via the concept of class which encapsulates data and its operation in a single unit.
 - ▶ Only the operations inside the class, can access their data member and not from outside the class.
 - ▶ Access specifiers can be used to selectively hiding the information.
- 

Polymorphism

- ▶ **Polymorphism** refers to the capability of having same names and exhibiting different behavior.
 - ▶ In object oriented context, it refers to the capability of having methods with the same names but exhibiting different behavior depending on the receiver.
 - ▶ In other words, you can send the same message to two different objects and they can respond in different ways.
 - ▶ In object oriented programming like C++, polymorphism is represented via overloading of functions.
- 

Inheritance

- ▶ **Inheritance** refers to the capability of a class (called as child class) to inherit the properties of another class (called as parent class).
 - ▶ New data elements and methods can be added to the new child class, but the data elements and methods of the parent class are also made available in the new class without need to rewrite them.
 - ▶ OOP is all about real world objects and inheritance is a way of representing real world relationship among them.
 - ▶ For example: **car, bus, bike** come under a broader category called **Vehicle** and inherit the properties of class vehicles.
- 

Procedural Vs Object Oriented Programming Paradigm

► Procedural Programming:

- It can be defined as a programming model which is based upon the concept of calling procedure to solve a problem
- Procedures are also known as routines, subroutines or functions.
- Procedures simply consist of a series of computational steps to be carried out to accomplish a task.
- Some example of Procedural programming languages are:
 - FORTRAN, COBOL, PASCAL, C, etc.

Procedural Vs Object Oriented Programming Paradigm

► Object Oriented Programming

- It can be defined as a programming model which is based upon the concept of objects.
- Objects contain data in the form of attributes and code in the form of methods that manipulate its data.
- Programs are designed using the concept of objects that interact with each other to model real world problems.
- Some of Object Oriented programming languages are:
 - C++, C#, Java, Python, Ruby, Scala etc.

Procedural Vs Object Oriented Programming – Major Differences

Procedural Programming	Object Oriented Programming
Program is divided into small components called functions .	Program is divided into small components called objects
More emphasis on Functions .	More emphasis on Data
Follows Top down approach.	Follows Bottom up approach.
No concept of access specifiers.	Use access specifiers like private , public , protected .
Does not have any proper way for hiding data hence, it is less secure .	Provides data hiding hence, it is more secure .
Adding new data and function is not easy.	Adding new data and function is easy.
Not close to real world scenario.	Models real world scenario very well.
Examples: C, FORTRAN, Pascal, etc.	Examples: C++, Java, Python, etc.

Any Query??

Object Oriented Programming using C++

Topics to be covered

- Iterative Constructs (Loops)
 - for loop
 - while loop
 - do-while loop
- Functions
 - Library Vs User-defined Functions (UDFs)
 - Declaring, Defining and Calling UDFs
 - Argument passing and default arguments
 - Pass by Reference

Iterative Constructs

- Used to repeat a set of instructions again and again until a specific condition is met
- Types of loops in C++:
 - for
 - while
 - do-while

Iterative Constructs

- Used to repeat a set of instructions again and again until a specific condition is met

- Types of loops in C++:

– for

– while

– do-while

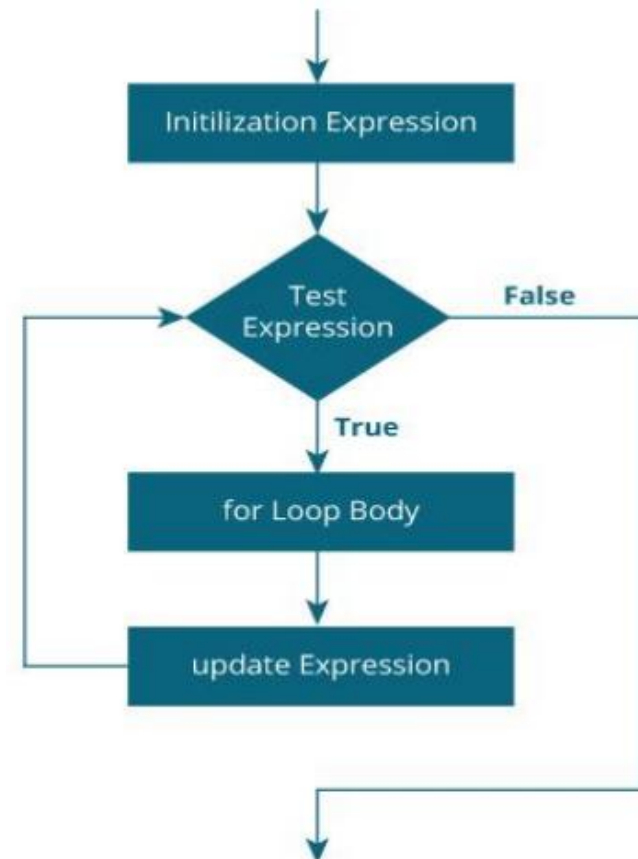
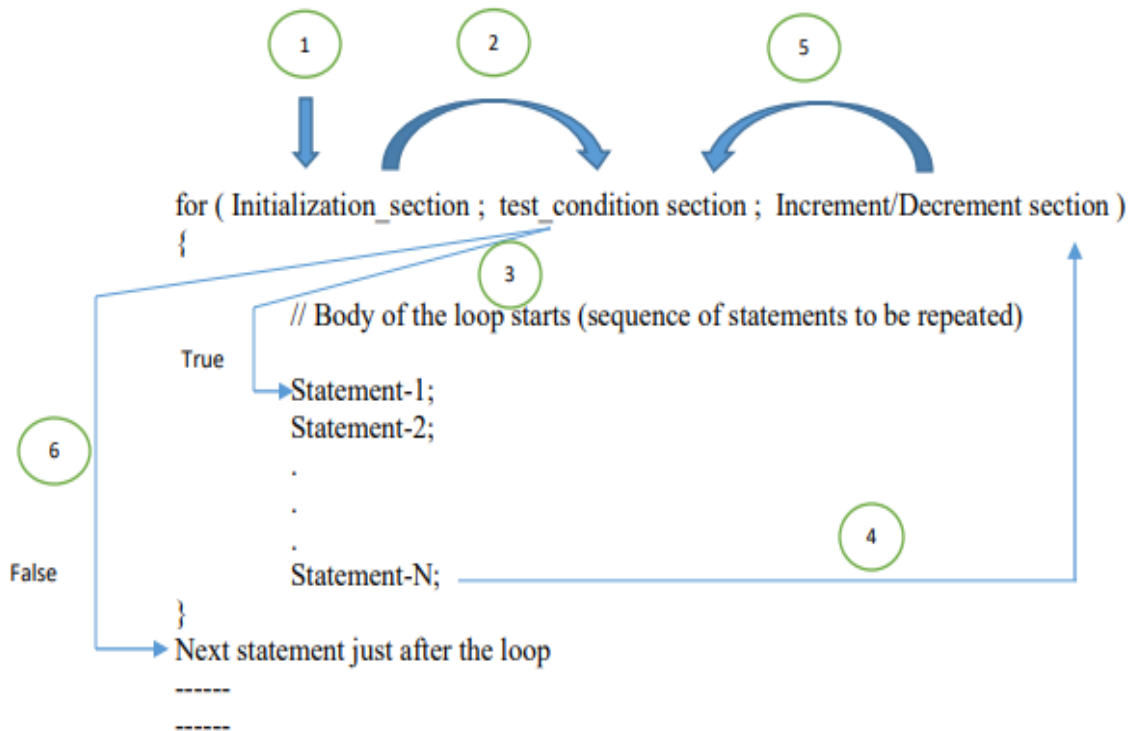


Entry controlled loops: **condition is checked before entering the loop**

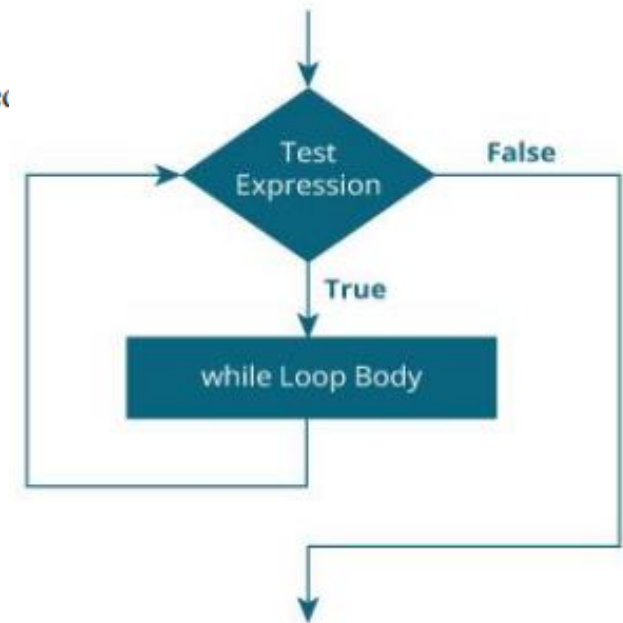
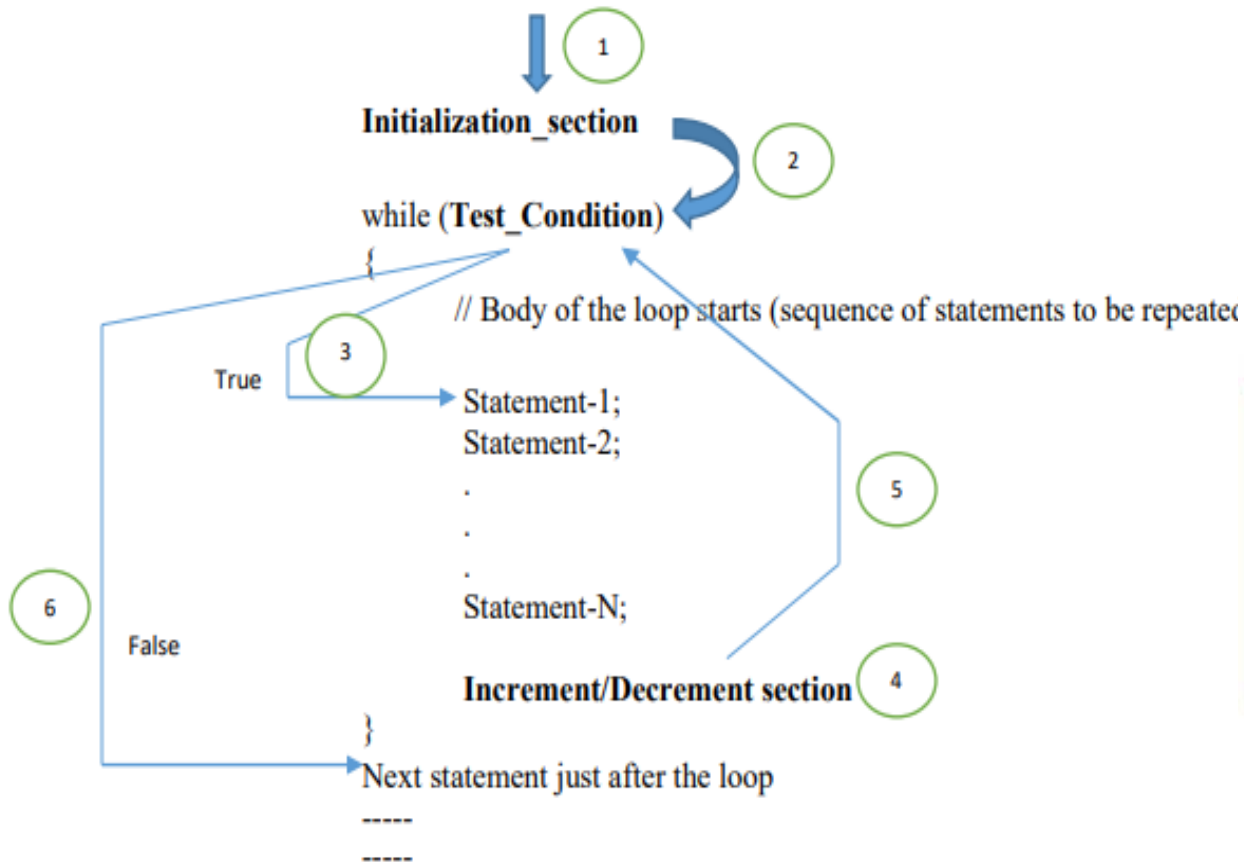
Exit controlled loops : **condition is checked at the end of the loop**

Iterative Constructs – *for loop*

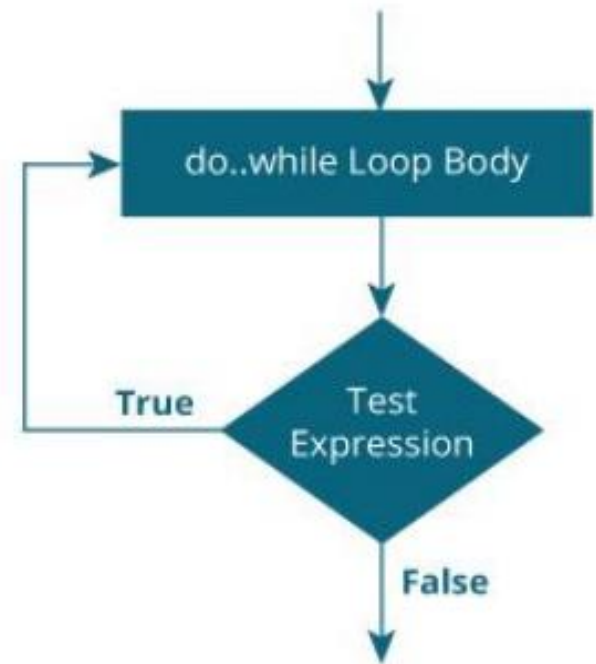
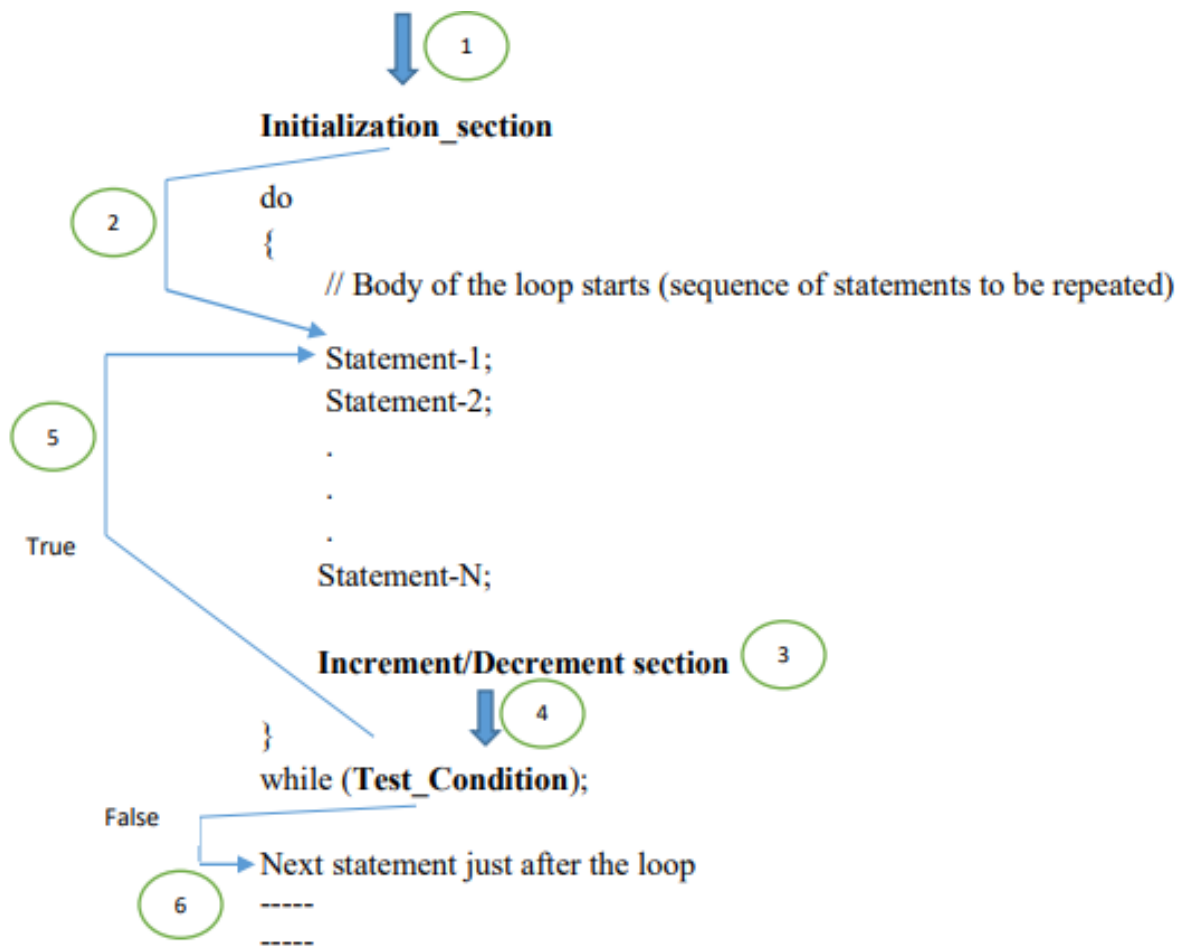
Syntax of for loop is:



Iterative Constructs – *while loop*



Iterative Constructs – *do-while* loop



Example of each Loop

for Loop:

```
for(i=1;i<=3;i++)  
{  
    cout << "hello";  
    cout << endl;  
}  
cout << "out of loop";
```

Output:

hello
hello
hello
out of loop

while Loop:

```
i=1;  
while(i<=3)  
{  
    cout << "hello";  
    cout << endl;  
    i++;  
}  
cout << "out of loop";
```

Output:

hello
hello
hello
out of loop

do-while Loop:

```
i=1;  
do  
{  
    cout << "hello";  
    cout << endl;  
    i++;  
} while(i<=3);  
cout << "out of loop";
```

Output:

hello
hello
hello
out of loop

Functions

- A function is a sequence of instructions written to perform a certain task
- **Advantages:**
 - Modular programming
 - Reusability of code
 - Easy to understand the code
 - Easy to detect bugs in the program
 - Easy to maintain and modify the program

Types of Functions

- Two broad categories:
 - **Library functions:**
 - Already written functions that are provided along with the C++ package
 - User just uses them as per the requirement
 - To use it, respective library must be included in the program
 - Example: `getline()`, `length()`, `size()` , `max()`, `min()`, etc.
 - **User Defined Functions (UDFs):**
 - Functions written by the programmer explicitly in the program
 - Programmer declare and define it according to the need
 - Further, they are call whenever required during the program

Declaring/Defining/Calling UDFs

Function declaration:

- Tells compiler about following 3 aspects:
 - ✓ *name of the function*
 - ✓ *number of arguments and their types that the function will receive*
 - ✓ *type of value to be returned by the function*

Declaration Syntax:

```
return_type func_name(parameter_list);
```

Declaring/Defining/Calling UDFs

Function definition:

- Defines the instructions to be performed by the function
- This sequence of instruction is also called as Body of the function

Defintion Syntax:

```
return_type func_name(parameter_list)
{
    // Body of the function
}
```

Declaring/Defining/Calling UDFs

Function Calling:

- Function can be called whenever required
- It can called any number of times

Calling Syntax:

```
func_name(argument_list_to_be_passed);
```

Sample Program

```
int main()
{
    int a=10; b=5, result;
    int sum(int, int);      // Function declaration
    result = sum(a,b);      // Function calling
    cout << "Addition=" << result;
    return 0;
}
```

*value (15) is
returned from
sum() function*

**Return of
control**

Jump of control

***./.* Function definition**

```
int sum(int x, int y) {
    int z;
    z = x+y;
    return z;
}
```

*value (10, 5) are passed
to the sum() function*

Default Arguments in a Function

- A default value can be provided for any parameter of the function
- If no argument is passed corresponding to that parameter, the default value is used for it
- **Example:**

```
/* Function definition of sum  
with default value for y=1 */
```

```
int sum(int x , int y=1)  
{  
    int z;  
    z = x+y;  
    return z;  
}
```

```
/* When sum() is called  
without second argument */
```

```
int main()  
{  
    int a=4,b=6;  
    // int sum(int , int =1);  
    cout << sum(a);  
    return 0;  
}
```

Reference in C++

- A reference in C++:
 - is an alias to some variable
 - shares same memory address as the variable
 - must be declared (using reference operator &) as well as initialized at the same time
 - is useful in passing arguments by reference during a function call
 - **Syntax:**
`data_type &ref_var = variable_name;`

Example program on Reference

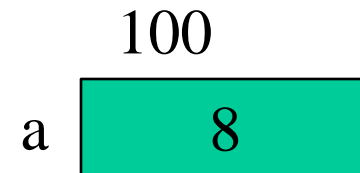
/* Example of Reference */

```
#include <iostream>
using namespace std;
int main()
{
    int a=8;
    int &b = a;
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<b<<endl;
    cout<<&b<<endl;
    return 0;
}
```


Example program on Reference

/* Example of Reference */

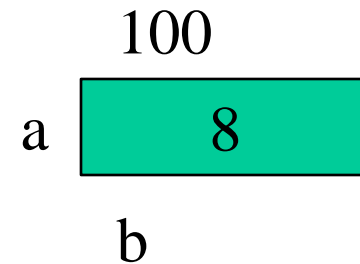
```
#include <iostream>
using namespace std;
int main()
{
    int a=8;           // normal variable a
    int &b = a;
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<b<<endl;
    cout<<&b<<endl;
    return 0;
}
```



Example program on Reference

/* Example of Reference */

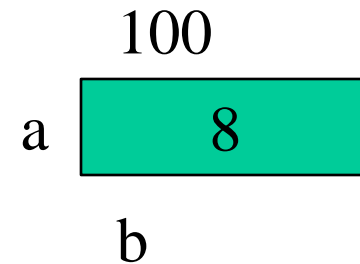
```
#include <iostream>
using namespace std;
int main()
{
    int a=8;           // normal variable a
    int &b = a;        // reference variable b
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<b<<endl;
    cout<<&b<<endl;
    return 0;
}
```



Example program on Reference

/* Example of Reference */

```
#include <iostream>
using namespace std;
int main()
{
    int a=8;           // normal variable a
    int &b = a;         // reference variable b
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<b<<endl;
    cout<<&b<<endl;
    return 0;
}
```



Output:

8
100

Example program on Reference

/* Example of Reference */

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int a=8;           // normal variable a
```

```
    int &b = a;         // reference variable b
```

```
    cout<<a<<endl;
```

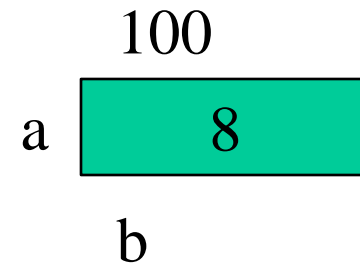
```
    cout<<&a<<endl;
```

```
    cout<<b<<endl;
```

```
    cout<<&b<<endl;
```

```
    return 0;
```

```
}
```



Output:

8
100
8
100

Example program on Reference

/* Example of Reference */

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int a=8;          // normal variable a
```

```
    int &b = a;        // reference variable b
```

```
    cout<<a<<endl;
```

```
    cout<<&a<<endl;
```

```
    cout<<b<<endl;
```

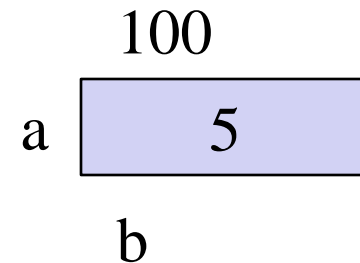
```
    cout<<&b<<endl;
```

```
    b=5;
```

```
    cout<<a<<endl;
```

```
    return 0;
```

```
}
```



Output:

8
100
8
100

Example program on Reference

/* Example of Reference */

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int a=8;           // normal variable a
```

```
    int &b = a;         // reference variable b
```

```
    cout<<a<<endl;
```

```
    cout<<&a<<endl;
```

```
    cout<<b<<endl;
```

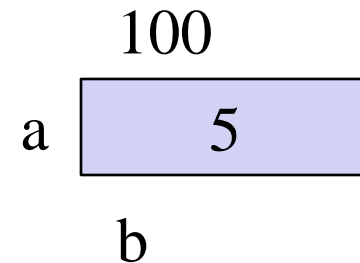
```
    cout<<&b<<endl;
```

```
    b=5;
```

```
    cout<<a<<endl;
```

```
    return 0;
```

```
}
```



Output:

8
100
8
100
5

Pointers in C++

- A pointer in C++:
 - is a variable which stores the address of another variable
 - has its own memory address other than the variable
 - is declared using * operator (can be initialized separately)
 - is useful in passing arguments via address to a function
 - is used to implement several significant data structures like: linked list, trees, graphs, etc.
 - **Syntax:**

```
data_type *ptr_var = &var_name;
```

Example program on Pointers

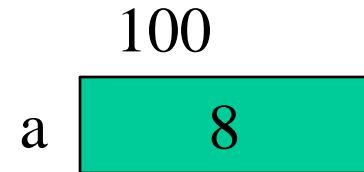
```
#include <iostream>
using namespace std;

int main(){
    int a=8;
    int *ptr;
    ptr=&a;
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<ptr<<endl;
    cout<<&ptr<<endl;
    cout<<*ptr<<endl;
    cout<<*&a<<endl;
    *ptr = *ptr + 10;
    cout<<*ptr<<endl;
    return 0;
}
```


Example program on Pointers

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

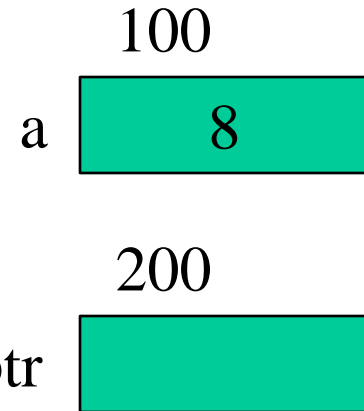
```
int main(){
    int a=8;           //normal variable a
    int *ptr;
    ptr=&a;
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<ptr<<endl;
    cout<<&ptr<<endl;
    cout<<*ptr<<endl;
    cout<<*&a<<endl;
    *ptr = *ptr + 10;
    cout<<*ptr<<endl;
    return 0;
}
```



Example program on Pointers

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

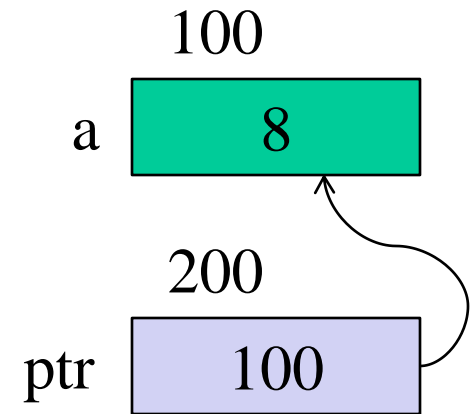
```
int main(){
    int a=8;           //normal variable a
    int *ptr;         //declaring pointer ptr
    ptr = &a;
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<ptr<<endl;
    cout<<&ptr<<endl;
    cout<<*ptr<<endl;
    cout<<*&a<<endl;
    *ptr = *ptr + 10;
    cout<<*ptr<<endl;
    return 0;
}
```



Example program on Pointers

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

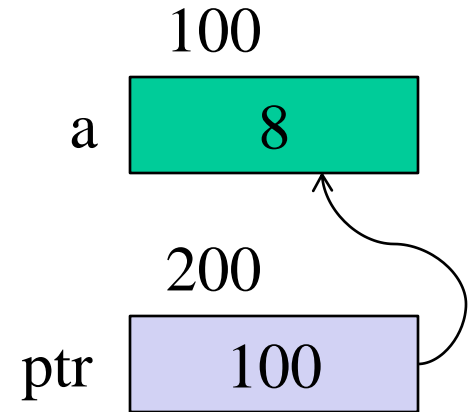
int main(){
    int a=8;           //normal variable a
    int *ptr;          //declaring pointer ptr
    ptr = &a;          //initializing pointer ptr
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<ptr<<endl;
    cout<<&ptr<<endl;
    cout<<*ptr<<endl;
    cout<<*&a<<endl;
    *ptr = *ptr + 10;
    cout<<*ptr<<endl;
    return 0;
}
```



Example program on Pointers

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

int main(){
    int a=8;           //normal variable a
    int *ptr;          //declaring pointer ptr
    ptr = &a;          //initializing pointer ptr
    cout<<a<<endl;    // printing value of a
    cout<<&a<<endl;    // printing address of a
    cout<<ptr<<endl;
    cout<<&ptr<<endl;
    cout<<*ptr<<endl;
    cout<<*&a<<endl;
    *ptr = *ptr + 10;
    cout<<*ptr<<endl;
    return 0;
}
```



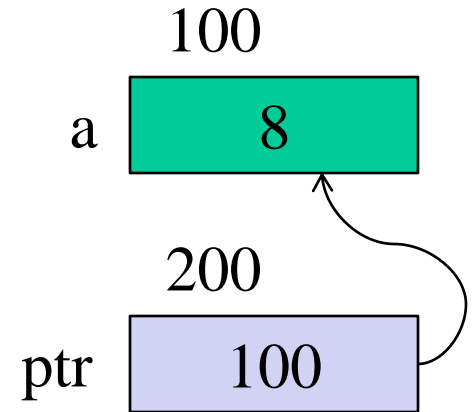
Output:

8
100

Example program on Pointers

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

int main(){
    int a=8;           //normal variable a
    int *ptr;          //declaring pointer ptr
    ptr = &a;          //initializing pointer ptr
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<ptr<<endl; // printing pointer's value
    cout<<&ptr<<endl; // printing pointer's address
    cout<<*ptr<<endl;
    cout<<*&a<<endl;
    *ptr = *ptr + 10;
    cout<<*ptr<<endl;
    return 0;
}
```



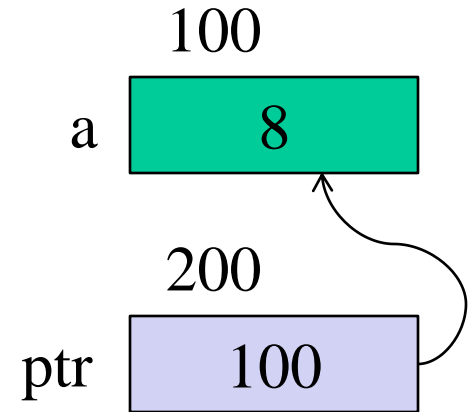
Output:

8
100
100
200

Example program on Pointers

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

int main(){
    int a=8;           //normal variable a
    int *ptr;          //declaring pointer ptr
    ptr = &a;          //initializing pointer ptr
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<ptr<<endl;
    cout<<&ptr<<endl;
    cout<<*ptr<<endl;  // printing value of a via ptr
    cout<<*&a<<endl;  // printing value of a
    *ptr = *ptr + 10;
    cout<<*ptr<<endl;
    return 0;
}
```



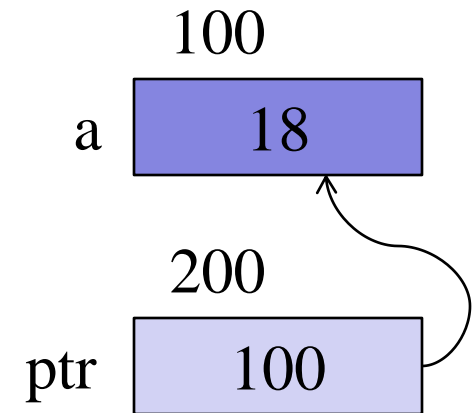
Output:

8
100
100
200
8
8

Example program on Pointers

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

int main(){
    int a=8;           //normal variable a
    int *ptr;          //declaring pointer ptr
    ptr = &a;          //initializing pointer ptr
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<ptr<<endl;
    cout<<&ptr<<endl;
    cout<<*ptr<<endl;
    cout<<*&a<<endl;
    *ptr = *ptr + 10;    // updating a using ptr
    cout<<*ptr<<endl;
    return 0;
}
```



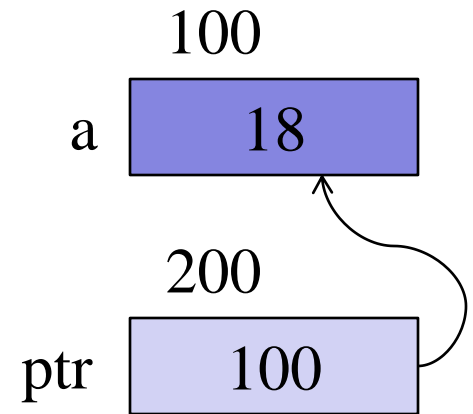
Output:

8
100
100
200
8
8

Example program on Pointers

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

int main(){
    int a=8;           //normal variable a
    int *ptr;          //declaring pointer ptr
    ptr = &a;          //initializing pointer ptr
    cout<<a<<endl;
    cout<<&a<<endl;
    cout<<ptr<<endl;
    cout<<&ptr<<endl;
    cout<<*ptr<<endl;
    cout<<*&a<<endl;
    *ptr = *ptr + 10;
    cout<<*ptr<<endl;  // reprinting the a's value
    return 0;
}
```



Output:

8
100
100
200
8
8
18

Pass by Reference

- Arguments reference is passed
- Receiving parameters are the reference variables
- **Example:**

```
/* Function definition having  
reference variables */
```

```
int sum(int &x , int &y)  
{  
    int z;  
    z = x+y;  
    return z;  
}
```

```
/* Main function */
```

```
int main()  
{  
    int a=4, b=8;  
    cout << sum(a,b);  
    return 0;  
}
```

Object Oriented Programming using C++

Prepared by:

Dr. Bakshi Rohit Prasad
(M.Tech., PhD, IIIT-Allahabad)

Topics to be covered

- Defining a class
 - Data members
 - Member functions
 - Inside the class
 - Outside the class
 - Parameterized member functions
- Creating objects and Accessing class members
- Constructors
 - Parameterized constructors
- Destructors

General C++ Program Components

- Typical C++ Programs consist of:
 - A function `main()`
 - One or more classes each containing:
 - data members and
 - member functions

Class Definition

- A class definition in C++ :
 - Tells the compiler about what **member functions** and **data members** belong to the class
 - Keyword **class** is used to define a class
 - Body of the class is enclosed in curly braces **{ }**
 - Specifies data members and member functions
 - Access-specifier **public**, **private** or **protected** used before them

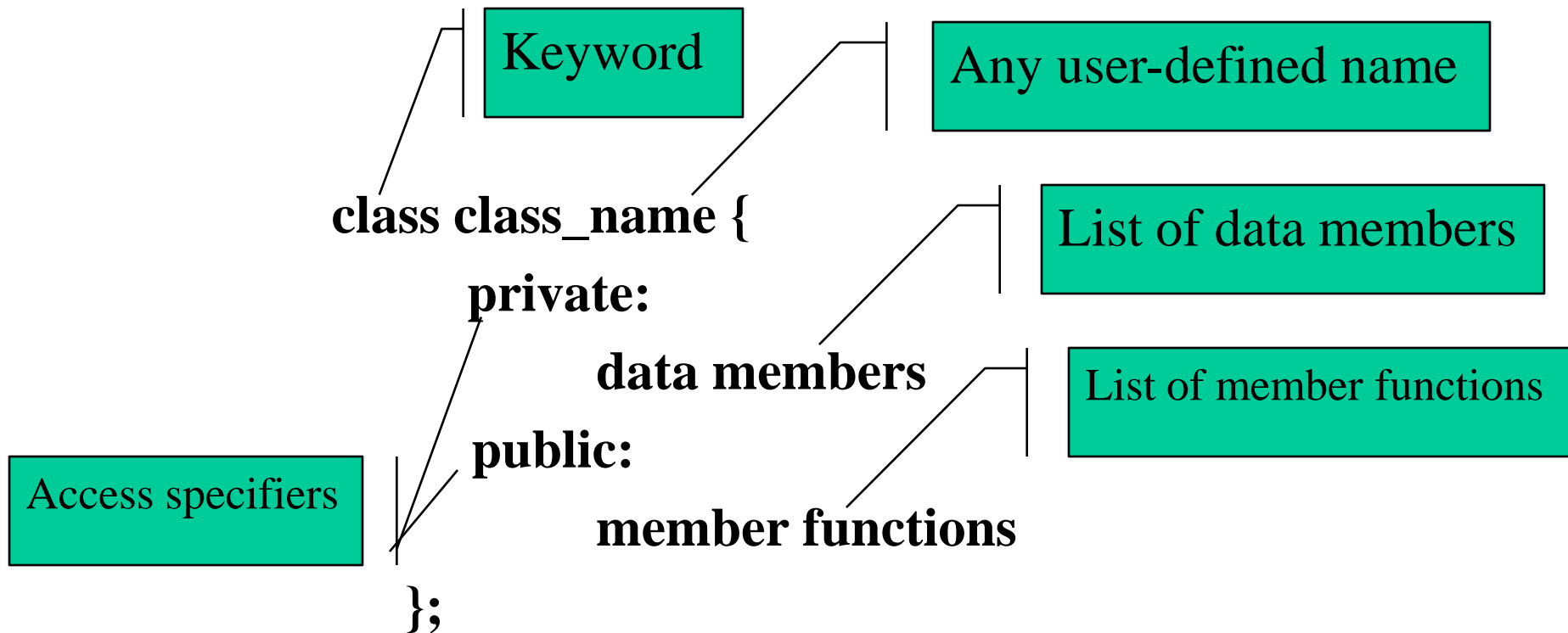
Class Definition

- A general syntax of a class definition in C++ :

```
class class_name {  
    private:  
        data members  
    public:  
        member functions  
};
```

Class Definition

- A general syntax of a class definition in C++ :



Example Class Definition

```
class Rectangle {  
    private:  
        int length;  
        int breadth;  
    public:  
        void area()  
        {  
            cout<<length*breadth;  
        }  
};
```


Creating Objects & Accessing Class members

```
class Rectangle {  
    private:  
        int length;  
        int breadth;  
    public:  
        void area()  
        {  
            cout << "area is:" << length*breadth << endl;  
        }  
};  
  
int main()  
{  
    Rectangle r1;  
    r1.length = 10;  
    r1.breadth = 5;  
    r1.area();  
    return 0;  
}
```

Creating Objects & Accessing Class members

```
class Rectangle {  
    public:  
        int length;  
        int breadth;  
  
    public:  
        void area()  
        {  
            cout << "area is: " << length*breadth << endl;  
        }  
};
```

Creating r1 object by using class name

```
int main()  
{  
    Rectangle r1;  
    r1.length = 10;  
    r1.breadth = 5;  
    r1.area();  
    return 0;  
}
```

Accessing class members using object name r1 along with dot (.) operator

Member function definition outside the class

```
class Rectangle {  
    private:  
        int length;  
        int breadth;  
    public:  
        void area();    // Function Prototype must be given  
};
```

/* Function definition outside the class using scope resolution

```
    operator :: */  
void Rectangle::area()  
    {  
        cout << length*breadth << endl;  
    }
```

Parameterized Member Functions

```
class Rectangle {  
    private:  
        int length;  
        int breadth;  
    public:  
        int area(int length, int breadth) {  
            return length*breadth;  
        }  
};  
  
int main()  
{  
    Rectangle r1;  
    cout << "area of rectangle is:" << r1.area(10, 5) << endl;  
    return 0;  
}
```

Constructors and Destructors

Constructors

- Initializes class objects (its data members)
- A constructor of a class:
 - is a method **with same name** as class
 - has **no return** value **not even void**
 - is automatically called as the class object is created
- We can have parameterized constructors also that receives arguments and set the values of data members for the object of that class

Example of Constructor

```
class Rectangle {  
    private:  
        int length;  
        int breadth;  
    public:  
        Rectangle() {  
            length = 1;  
            breadth = 1;  
        }  
  
        void area(){  
            cout << "area is:" << length*breadth << endl;  
        }  
};  
  
int main()  
{  
    Rectangle r1;  
    r1.area();  
    return 0;  
}
```

Example of Constructor

```
class Rectangle {  
    private:
```

```
    int length;  
    int breadth;
```

```
    public:
```

```
        Rectangle() {  
            length = 1;  
            breadth = 1;  
        }
```

```
        void area(){  
            cout << "area is:" << length*breadth << endl;  
        }
```

```
};
```

```
int main()  
{
```

```
    Rectangle r1;  
    r1.area();  
    return 0;
```

```
}
```

Constructor has no return type

Constructor initializes
length and breadth

Constructor is called here

Parameterized Constructor

```
class Rectangle {  
    private:  
        int length;  
        int breadth;  
    public:  
        Rectangle(int x, int y) {  
            length = x;  
            breadth = y;  
        }  
  
        void area(){  
            cout << "area is:" << length*breadth << endl;  
        }  
};  
  
int main()  
{  
    Rectangle r1(7,5);  
    r1.area();  
    return 0;  
}
```

Function Overloading

- It is one of the important features of C++
- It is defined as the ability of the programming language to have functions with same name but different argument lists and/or return type
- These argument lists may differ in terms of:
 - Number of arguments
 - Data type of arguments
- During function calling number and types of arguments passed decide which function is to be called from among multiple available functions having same name

Example of Function Overloading

```
#include <iostream>
using namespace std;
/* Overloaded sum function */
int sum(int x, int y) {
    return x+y;
}

int sum(int x) {
    return x+3;
}

int main()
{
    cout<<"Value of sum is: "<<sum(10,5)<<endl;
    cout<<"Value of sum is: "<<sum(10)<<endl;
    return 0;
}
```

Constructor Overloading

- Overloading a constructor of a class
- It is defined as the ability of the C++ language to have class constructors with same name but different argument lists
- These argument lists may differ in terms of:
 - Number of arguments
 - Data type of arguments
- During instantiation (creation of object of the class) respective constructor is invoked based on number and types of arguments passed

Example of Constructor Overloading

```
class Rectangle {  
    private:  
        int length;  
        int breadth;  
    public:  
        /* Overloaded constructor */  
        Rectangle(){  
            length = 3;  
            breadth = 2;  
        }  
  
        Rectangle(int x, int y) {  
            length = x;  
            breadth = y;  
        }  
  
        void area(){  
            cout << "area is:" << length*breadth << endl;  
        }  
};
```

```
/* main function */  
int main()  
{  
    Rectangle r1;  
    Rectangle r2(7,5);  
    r1.area();  
    r2.area();  
    return 0;  
}
```

Dynamic Memory Allocation

- Dynamic memory allocation refers to:
 - manual allocation of memory **on demand**
 - **explicitly** by the programmer
 - on **Heap data storage** in memory
- Advantages:
 - On demand memory allocation and deallocation
 - Creation of variable size data storage structures
 - Helpful in creating several important data structures like linked list, trees, graphs, etc.

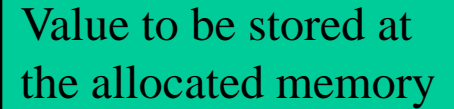
Dynamic Memory Allocation

- In C++, dynamic memory allocation operators are '**new**' and '**delete**'
- '**new**' operator is used for allocating new memory on demand during the program
- **Syntax:**
 pointer_variable = **new** data-type;

Dynamic Memory Allocation

- **Declaration with initialization:**

`pointer_variable = new data_type (value);`

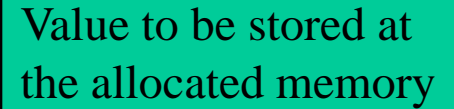


Value to be stored at
the allocated memory

Dynamic Memory Allocation

- **Declaration with initialization:**

`pointer_variable = new data_type (value);`



Value to be stored at
the allocated memory

Dynamic Memory Allocation

- **Declaration with initialization:**

`pointer_variable = new data_type (value);`

Value to be stored at the allocated memory

- **Example-1 :**

`int *ptr;`

`ptr = new int;`

Declaration of a pointer variable

100  ptr

Dynamic Memory Allocation

- **Declaration with initialization:**

`pointer_variable = new data_type (value);`

Value to be stored at the allocated memory

- **Example-1 :**

`int *ptr;`

`ptr = new int;`

Declaration of a pointer variable

100

200

ptr

200

Memory allocated by new operator

Dynamic Memory Allocation

- **Declaration with initialization:**

Value to be stored at the allocated memory

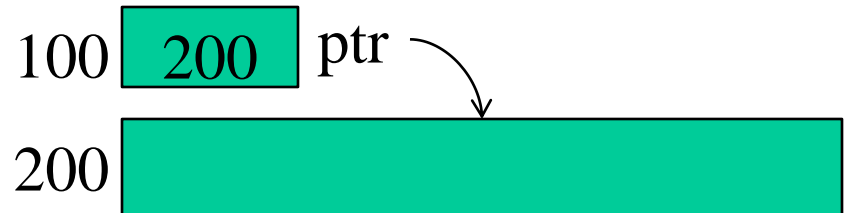
`pointer_variable = new data_type (value);`

- **Example-1 :**

Declaration of a pointer variable

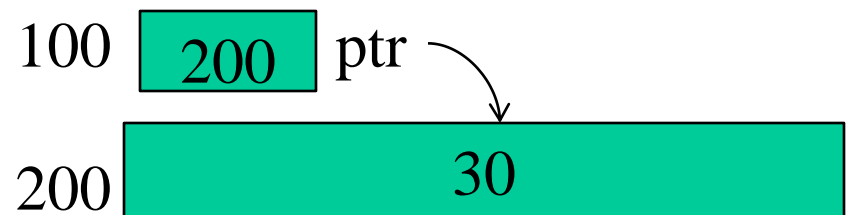
`int *ptr;`

`ptr = new int;`



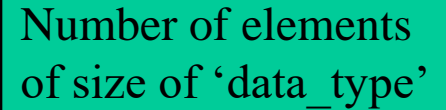
- **Example-2 :**

`int *ptr = new int(30);`



Dynamic Memory Allocation

- **Allocating block of memory:**



Number of elements
of size of 'data_type'

```
pointer_variable = new data_type[block_size];
```

Dynamic Memory Allocation

- Allocating block of memory:

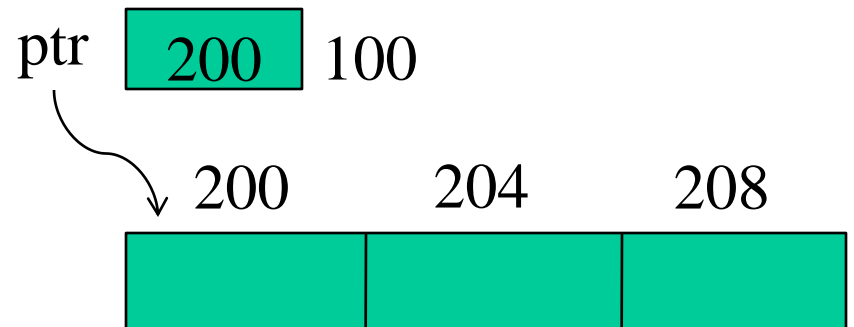
Number of elements
of size of 'data_type'

```
pointer_variable = new data_type[block_size];
```

- Example-3 :

```
int *ptr;
```

```
ptr = new int[3];
```



Memory for 3 elements of size 'int'

Delete operator

- ‘**delete**’ operator is used for deallocating existing allocated memory

- **Syntax:** Two different use:

```
int *ptr1 = new int;
```

```
delete pointer_to_allocated_memory;
```

Example : delete ptr1;

```
int *ptr2 = new int[3];
```

```
delete[] pointer_to_block_of_memory;
```

Example: delete[] ptr2;

Destructors

- A Destructor is special member function that:
 - is invoked automatically to release the resources held by the object (before the object is reclaimed by the system)
 - has the same name as the class, preceded by a tilde (~) character
 - cannot take arguments and cannot return values
 - cannot be overloaded
- When not defined by programmer, the compiler provides a default destructor which is sufficient in general.
- A custom destructor is required when a class keeps:
 - handles to system resources that need to be released or
 - pointers that own the memory they point to