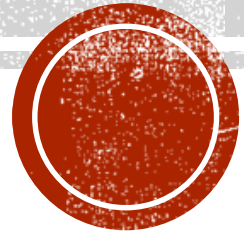
OBJECT ORIENTED PROGRAMMING...

Working with Classes, Constructors and Destructors



CONSTRUCTORS/DESTRUCTORS

ACTIVITY

Write a simple C++ class to represent a rectangle, including methods to calculate its perimeter and area

In Object-Oriented Programming (OOP), initializing variables when creating objects is generally a good practice.

Proper initialization ensures that the object is in a valid, predictable state from the moment it is instantiated.

CONSTRUCTORS

- Special class member functions **automatically called** when an object of a class is **created/instantiated**.
- Purpose: initialise the newly created object.
- A constructor's name: Same as the name of the class.
  - defined inside or outside the class definition
- A constructor CANNOT return a value and has NO return type (NOT even void).
- To create a constructor, use the same name as the class, followed by parentheses ():
- There are 3 types of constructors:
  - Default constructors
  - Parametrized constructors
  - Copy constructors

DEFAULT CONSTRUCTOR: EXAMPLE

The constructor which does NOT take any argument (has no parameters).

ACTIVITY

Write a default constructor to initialize the variables to Zero.

DEFAULT CONSTRUCTOR: EXAMPLE

The constructor which does NOT take any argument (has no parameters).

Example:

```
Rectangle()
{
        width=0;
        length=0;
}
```

DEFAULT CONSTRUCTOR

- A constructor that can be invoked **without any arguments** e.g. a constructor with an **empty parameter list**, or a constructor with **default values** for all of its arguments.

ACTIVITY:

Is it possible to define more than one default constructor for a class? [YES/NO]

Justify…

Even if any constructor is NOT defined explicitly, the compiler will automatically provide a default constructor implicitly that leaves all data members un-initialized.

PARAMETERIZED CONSTRUCTOR: EXAMPLE

Pass arguments to constructors. The initial values are passed as arguments to the constructor function.

The arguments passed to a constructor are typically used to initialise the object's data members

Used to initialize the various data elements of different objects with different values when they are created.

Example:

```
Rectangle(int w, int l)
{
        width=w;
        length=l;
}
```

INVOKE CONSTRUCTORS

- A constructor CANNOT be called directly like a normal member function.

Example:

```
int main()
{
Rectangle rect1;
        rect1.Rectangle();
}
```

PASSING ARGUMENTS TO CONSTRUCTOR

- Constructor is not invoked like a normal function.
- Arguments can ***only*** be passed to it when an object is declared.
- The *comma-separated list* of arguments must appear in parentheses after the object name.

Example:

```
int main()
{
        Rectangle rect1(1,1);
}
```

CONSTRUCTORS

Whenever one or more non-default constructors (with parameters) are defined for a class …

A default constructor(without parameters) should be explicitly defined as the compiler will not provide a default constructor.

CONSTRUCTORS VS. MEMBER FUNCTIONS

Differences

- **Constructor** name must be same as class name but **Functions** cannot have same name as class name.
- **Constructor** does not have a return type but **Functions** must have a return type.
- **Constructors** are invoked automatically at the time of object creation and cannot call explicitly using class objects. **Functions** are called explicitly using class objects.

Similarities

- **Constructors** and **Member Functions** can be defined with different parameters.
- Parameters can be passed into **Constructors** and **Member Functions** as arguments.

DEFINING CONSTRUCTOR VS MEMBER FUNCTION

Constructor Syntax:

```
className :: className(… )
   {
       …
    }
```

Member Function Syntax:

```
ReturnType className::MemberFunctionName(… )
   {
     ……
   }
```

CONSTRUCTOR INITIALIZATION

```
//A constructor initializing variables to zero
Rectangle::Rectangle()
{
        width=0;
        length=0;
}

//A parameterized constructor
Rectangle::Rectangle(int w, int l)
{
        width=w;
        length=l;
}
```

```
//A constructor initializing variables to zero

Rectangle::Rectangle(): width(0),length(0)
{ }


//A parameterized constructor

Rectangle::Rectangle(int w, int l): width(w),length(l)
{ }
```

CONSTRUCTOR INITIALIZATION LIST

- A list in the definition of a constructor that specifies initial values for the data members of that class.
- Initialise its data members by invoking constructors

```
ClassName(parameterList) : member1(valueList1),member2(valueList2)…
{
  // …
}
```

OVERLOADED CONSTRUCTOR

- Different functions can have the same name as long as they differ in their parameter lists. This practice is called overloading.
- Constructors are often overloaded to provide alternative ways of creating an instance of a class.

//Constructor signature

❶ Rectangle();

❷ Rectangle(int,int);

❸ Rectangle(int);

DESTRUCTORS

- A special member function that is automatically called whenever an object of its class ceases to exist.
- Destructs or deletes an object.
- Purpose: release any resources that the corresponding constructor allocated.
- A destructor's name is the same as the name of the class it belongs to, except that the name is preceded by a tilde symbol (~).

Syntax:

```
~constructor-name();
```

DESTRUCTOR: EXAMPLE

```
Constructor is called
Hello World!
Destructor is called
```

```
1  #include <iostream>
2  using namespace std;
3
4  class HelloWorld{
5  public:
6      HelloWorld(){
7      cout<<"Constructor is called"<<endl;}
8
9      ~HelloWorld(){
10     cout<<"Destructor is called"<<endl;}
11
12     void display(){
13       cout<<"Hello World!"<<endl;
14    }
15 };
16
17 int main(){
18    //Object created
19    HelloWorld obj;
20    //Member function called
21     obj.display();
22 }
```

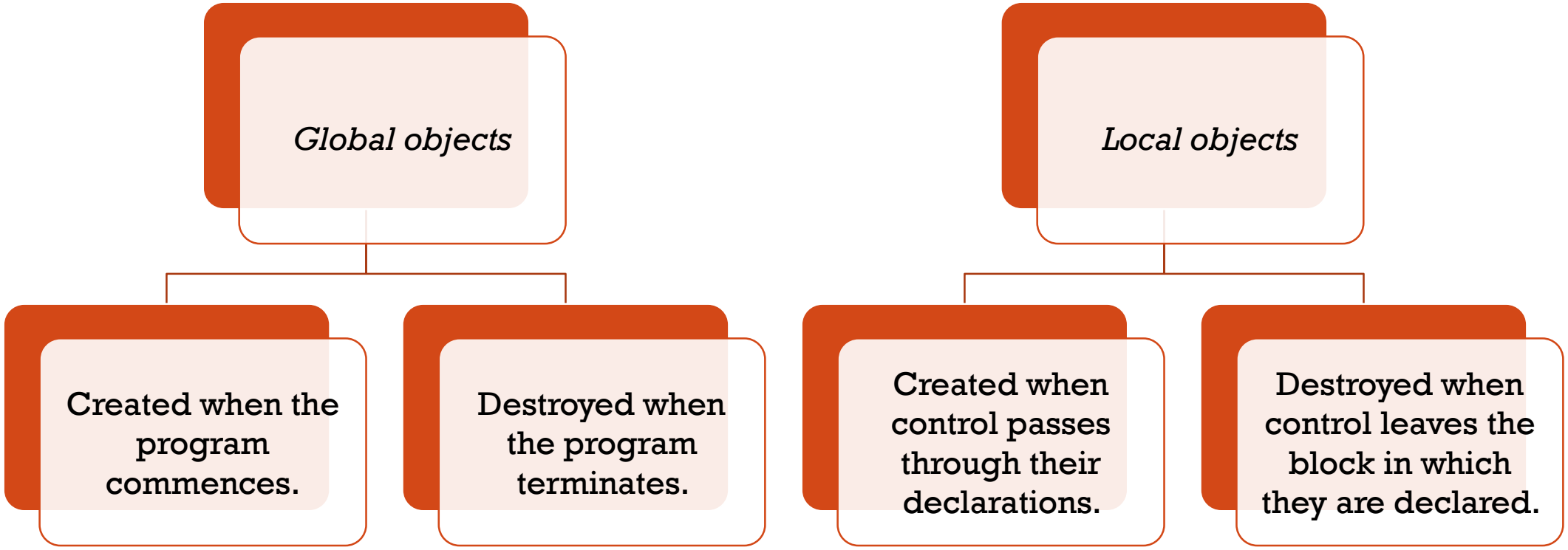
DESTRUCTORS

- Should begin with tilde sign(~)
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- A destructor should be declared in the public section of the class.
- There cannot be more than one destructor in a class.
- When a destructor is NOT specified in a class, compiler generates a default destructor.

CONSTRUCTORS/DESTRUCTORS: CALL ORDER

- Depends on order of execution (when execution enters and exits scope of objects)
- Destructor calls reverse order of constructor calls
- Global scope objects
  - Constructors are called before any other function (including main)
  - Destructors are called when main terminates (or exit function called)
    - Not called if the program terminates with abort
- Local scope objects
  - Constructors are called when objects are defined and each time execution enters the scope
  - Destructors are called when objects leave scope / execution exits block in which object defined
    - Not called if the program ends with exit or abort

OBJECT LIFETIMES



ACTIVITY

OUTPUT?

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

class Rectangle
{
    int width;
    int length;

    public:

        Rectangle()
        {
        width=0;
        length=0;
        }

        Rectangle(int w,int l){
            width=w;
            length=l;
        }

        Rectangle(int w){
            width=w;
            length=0;
        }

        ~Rectangle(){
            cout<<"Destructor is called"<<endl;
            cout<<"Object with "<<width<<" "<<length<<endl;
        }

        void getWL();
};
```

```cpp
void Rectangle::getWL()
{
    cout<<"Width is "<<width<<endl;
    cout<<"Length is "<<length<<endl;
}

int main()
{
    Rectangle Rec3;
     Rec3.getWL();

    Rectangle Rec1(4);
     Rec1.getWL();

    Rectangle Rec2(1,1);
     Rec2.getWL();

}
```

```
Width is 0
Length is 0
Width is 4
Length is 0
Width is 1
Length is 1
Destructor is called
Object with 1 1
Destructor is called
Object with 4 0
Destructor is called
Object with 0 0
```

```
class Rectangle
{
        int width;
        int length;
         public:
          //constructor signature
            Rectangle();
            Rectangle(int,int);
            Rectangle(int);
          //destructor signature
            ~Rectangle();
          //function signature
            void getWL();
            int areaRect();
};
```

ACTIVITY: RECTANGLE 2

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

class Rectangle
{
    int width;
    int length;

    public:
        //constructor signature
        Rectangle();
        Rectangle(int,int);
        Rectangle(int);
        //destructor signature
        ~Rectangle();
        //function signature
        void getWL();
        int areaRect();
};
```

```cpp
//A function to calculate the area of the rectangle
int Rectangle::areaRect()
{
    return width*length;
}

int main()
{
    Rectangle Rec1(4);
     Rec1.getWL();

    Rectangle Rec2(1,1);
     Rec2.getWL();

    Rectangle Rec3;
     Rec3.getWL();

    cout<<"Rectangle 1 Area: "<<Rec1.areaRect()<<endl;
    cout<<"Rectangle 2 Area: "<<Rec2.areaRect()<<endl;
    cout<<"Rectangle 3 Area: "<<Rec3.areaRect()<<endl;
}
```

```
Width is 4
Length is 10
Width is 1
Length is 1
Width is 0
Length is 0
Rectangle 1 Area: 40
Rectangle 2 Area: 1
Rectangle 3 Area: 0
Destructor is called: 0
Destructor is called: 1
Destructor is called: 40
```

ACTIVITY :

```
class CreateDestroy{
      private:
       int objectID;
       string message;

      public:
      CreateDestroy(int,string);
      ~CreateDestroy();
};
```

```
CreateDestroy::CreateDestroy(int ID,string msg){
  objectID = ID;
  message=msg;
  cout << "Object "<< objectID << " constructor runs "
       << message << endl;
}

CreateDestroy::~CreateDestroy()
{
  cout << "Object " << objectID << "   destructor runs    "
       << message << endl;
}
```

```
int main()
{
  cout << "MAIN FUNCTION: EXECUTION BEGINS" << endl;

  CreateDestroy obj1( 1, "(local automatic in main)" );
  CreateDestroy obj2( 2, "(local automatic in main)" );
  CreateDestroy obj3( 3, "(local automatic in main)" );

  cout <<"\nMAIN FUNCTION: EXECUTION ENDS" << endl;
}
```

```
MAIN FUNCTION: EXECUTION BEGINS
Object 1 constructor runs (local automatic in main)
Object 2 constructor runs (local automatic in main)
Object 3 constructor runs (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 3   destructor runs    (local automatic in main)
Object 2   destructor runs    (local automatic in main)
Object 1   destructor runs    (local automatic in main)

--------------------------------
Process exited after 0.04353 seconds with return value 0
Press any key to continue . . .
```

ACTIVITY :

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

class List{
    private:
        int length;
        int *array;
    public:
        List();
        List(int size);
        ~List();

        void displayList();
        void SetList(int a);
    };
```

```
int main()
{
    cout<<"START"<<endl;

    List l1;
    l1.displayList();
    l1.SetList(2);
    l1.displayList();

        List l2(3);
        l2.displayList();
}
```

```
List::List()
{
  length = 10;
  array = new int [length];
  assert(array != 0);
  for (int i = 0; i < length; i++)
     array[i] = 0;
}

List::List(int size)
{
  length = size;
  if (length <= 0)
       array = 0;
   else
   {
      array = new int [length];
      assert(array != 0);
      for (int i = 0; i < length; i++)
        array[i] = 3;
   }
 }
```

```
void List::displayList(){
      for (int i = 0; i < length; i++)
         cout<<array[i]<<",";
   cout<<endl;
}

void List::SetList(int a){
      for (int i = 0; i < length; i++)
      array[i] = i+a;
}

// Destructor
List::~List()
{
  delete [] array;
  cout<<"Called the destruct"<<endl;
}
```

COPY CONSTRUCTOR

- A *copy constructor* is an overloaded constructor that initializes an object using another object of the same class.
- Initialize the members of a newly created object by copying the members of an already existing object.
- A copy Constructor may be called, when
  - an object of the class is returned by value.
  - an object of the class is passed (to a function) by value as an argument.
  - an object is constructed based on another object of the same class.
  - the compiler generates a temporary object.
- A copy constructor cannot return a value and has no return type (not even void).
- Allows instantiating an object as an exact copy of another in terms of its attribute values.
- Default or User Defined copy constructor

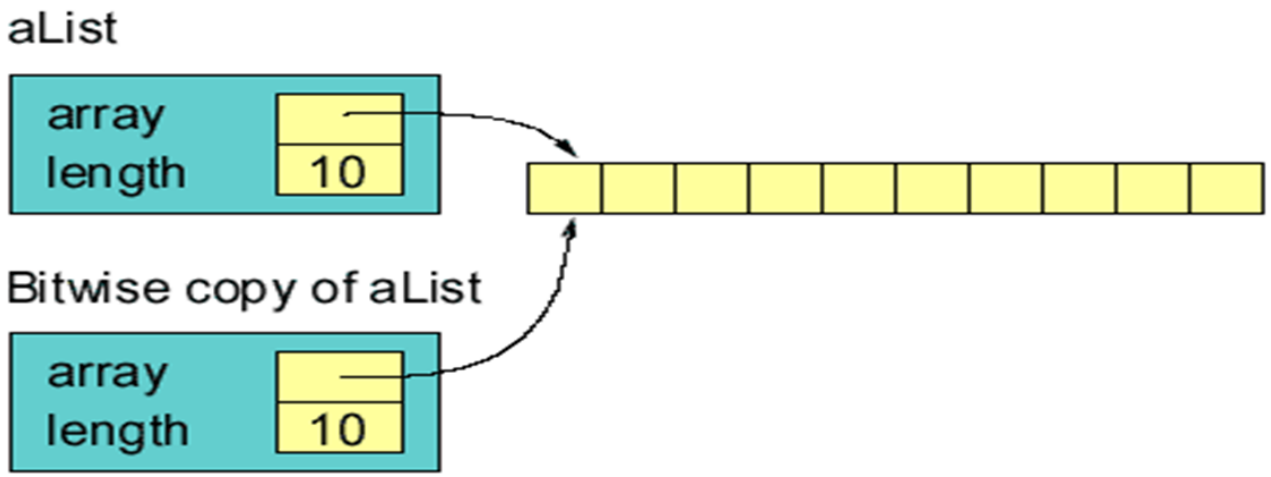
DEFAULT COPY CONSTRUCTOR

If a copy constructor for a class is NOT defined explicitly, then the compiler will *automatically* provide a copy constructor that makes a *bitwise* copy of the class instance.

Bitwise copying is a way to get a copy of a class object by copying every bit (byte) of a particular class object (instance). Bitwise copying is used when it is necessary to obtain an identical destination object from the source object.

DEFAULT VS USER DEFINED COPY CONSTRUCTOR

- Works fine in general.
- HOWEVER, if an object has pointers or any runtime allocation of the resource like a file handle, a network connection, etc.
- Need to define a user-defined copy constructor.



USER DEFINED COPY CONSTRUCTOR

Copy Constructor Syntax

```
className::className(const className& old_Object)
{
   // ...
}
```

USER DEFINED COPY CONSTRUCTOR

```
List::List(const List &original)
{
 length = original.length;
 if (length > 0)
 {
  array = new int [length];
  assert(array != 0);
  for(int i = 0; i < length; i++)
      array[i] = original.array[i];
    }
    else
      array = 0;
 }
```