Course: C++ Programming

Chapter 3: Constructors and Destructors

Directed by:
Dr. Sakka Rouis Taoufik

https://github.com/srtaoufik/coursCpp

1

I.Object Initialization

It is possible to assign initial values to the attributes of an object through a list of values specified between two curly brackets and separated by commas (as for thetables in C).

Example:

```
class Point2D {
public:
   int x;
   int y; };
class Point3D {
public:
   int x;
   int y;
   int z;
};
```

```
Point2D P1={5,6};
Point3D P2={5,6,9};
Point3D P3={4,,9} ;
// the second attribute is initialized to 0.
Point2D T[3] = {1,5,6,8,9,10};
Point2D T[3] = {{1,5},{6,8},{9,10}};
```

II.Limitations of initialization with lists of values

- All members of the object to be initialized must be public.
- The object must be of a base class and have no virtual members.
- Some initialization operations are complex and require multiple steps such as the following example **:**

```
class TabIntegers {
public :
   int Nb;
   int* T; };
```

To be able to initialize member T, the latter must first be allocated.

II.Limitations of initialization with lists of values

Solution 1: Using an initialization method

+ Ability to perform allocation operations for dynamic members.
+ Ability to access private and protected members.

```
class TabIntegers {
 int Nb;
  int* T;
public :
  void Init(int Ni) {
     Nb=Ni;
     T=new int [Nb]; }
  void Init(int* Ti, int Ni) {
     Nb=Ni;
     T=new int [Nb];
     for(int k=0;k<Nb;k++)
        T[k]=Ti[k]; }
};
```

```
void main() {
  TabIntegers TE;
  TE.init(5);
}
```

II.Limitations of initialization with lists of values

Solution 1: Using an initialization method

Disadvantages:

-To use the object, it is necessary to explicitly call the Init method each time. This explicit call can cause errors, especially in large programs where the risk of forgetting becomes high.

-Using Init cannot be considered initialization in the true technical sense because initialization is usually done in the same instruction asthe statement.

II.Limitations of initialization with lists of values

Solution 2: Using C++ Constructors

C++ offers a solution for creating and initializing objects that solves all the above-mentioned problems (initialization in the true sense of the word, implicit calls, initialization of dynamic members with memory allocation, etc.).

Definition:

A constructor is a particular method of a class that is executed automatically in an implicit manner, when an object of that class is created.

II.Limitations of initialization with lists of values

Solution 2: Using C++ Constructors

Basic role: A constructor of a class ensures the reservation of the memory space necessary for the creation of any object of this class. The space referred to here refers to the basic space required to create the object. It does not include the dynamic spaces associated with any dynamic members of theclass.

For the class *TabIntegers* defined previously, the basic space required to create an object is equal to the sum of the sizes of *Nb* and the pointer *T*.

II.Limitations of initialization with lists of values

Solution 2: Using C++ Constructors

Usual exploitation of manufacturers: Constructors are usually used by programmers to perform the following operations:

- Initialization of class attributes (public, private or protected).
- Allocation of memory spaces required by dynamic members of the class.

III.Declaration and definition of a constructor

- A constructor is a method of the class.
- A constructor must have the same name as its class.
- A constructor can take 0, one, or more parameters.
- A constructor does not return any value. (Not putting a return type in front of the constructor does not mean that the constructor returns an integer by default (*int*) as is the case for classical methods).
- A class can have one or more constructors. If there are multiple constructors, they must follow the constructor overloading rules.methods.
- Like any method a constructor can be defined inside the class orthe outside.

III.Declaration and definition of a constructor

Example:

```
class Time {
 int Hour;
 int Minute;
 int Second;
 public :
  Time (int H, int M, int S){
     Hour = H;
     Minute = M;
     Second = S;
 }
```

```
void Show(){
  cout<<"The time is: ";
  cout<<Hour<<':'<<Minute<<':'
<<Second<<endl;
}
};
```

III.Declaration and definition of a constructor

Remarks:

•The public, private, and protected access specifiers apply to constructors in the same way as to any class method.

•To be able to call a constructor and sub-sequently create an object, this constructor must be mandatory public.

IV.Call from a builder

Any construction of an object of a class must be accompanied by a callto its manufacturer.

1. Case of an object of type auto

Creating an object of type auto can be done through an explicit or implicit call to thebuilder.

Explicit call:

The constructor is called by name as a classic method.

ClassName ObjectName = ClassName(<effective parameters>);

Time T = Time (16, 30, 25);

Implicit call:

ClassName ObjectName(<effective parameters>);

Time T (16, 30, 25);

IV.Call from a builder

2. Case of a dynamic object

For dynamic object creation, only explicit call of the constructor is possible.

ClassName* ptrObject = new ClassName(<effective parameters>);

Time* T = new Time(16,30,25);

Example :

Time T1(16,30,25);
Time* T2 = new Time(17,44,59);
T1.Display();
T2->Display();

V.Constructor overloading

•It is possible to define multiple constructors for the same class. These constructors are then said to be overloaded.

•Constructors in the same class must follow C++ function overloading rules. In other words, they must have different signatures.

V.Constructor overloading

```
class Time {
 int Hour;
 int Minute;
 int Second;
public:
Time(int H,int M,int S)  {
 Hour = H;
 Minute = M;
 Second = S;
}
Time() {
 Hour = 0;
 Minute =0;
 Second = 0;
}
```

```
void ToInput() {
 cout<<"Give the time: "
 cin>>Hour>>Minute>>Second;
}
void Display() {
 cout<<"The time is: ";
 cout<<Hour<<':'<<Minute<<':'<<Second;
 cout<<endl;
}
};
void main() {
 Time T1;
 T1.ToInput();
 TimeT2(16,15,30);
 T1.Display();
 T2.Display();
}
```


V.Constructor overloading

Rk1:If a class has multiple constructors, then when creating an object from that class, one of those constructors will be called. In the previous example, object T1 is created using the constructor without arguments. This object is uninitialized. Its contents will then be entered using the Enter method. Object T2 is created using the parameterized constructor. Its contents are initialized at 16:15:30.

Rk2:Any object can only be created through a constructor call. Therefore, it is the constructors defined in a class that determine how objects from that class can be instantiated. For example, if the no-argument constructor Time() was not defined in the Time class then the instantiation Time T1; would be syntactically incorrect (compile error).

V.Constructor overloading

Rk3:It is always recommended to define multiple constructors in a class to provide users of the class with more possible scenarios and greater flexibility regarding object instantiation.

VI.The Destroyers

A destructor plays the symmetric role of the constructor. It is automatically called to free the basic memory space necessary for the object's very existence (space occupied by attributes). This freeing is implicit. It is performed by any destructor. It is generally recommended to use the destructor to do:

•The release of any dynamic memory spaces associated with the object's attributes. Unlike the release of base space, this release is not done automatically. It must be done explicitly by the programmer.
•Closing any files used by the object.
•Data recording, etc.

VII.Declaring and defining a destructor

•A destructor has the same name as the class name but preceded by a tilde ~ (for thedistinguish from the manufacturer).
•A destructor does not return any value.
•A destructor never takes parameters.
•A destructor is a method of the class. All the rules that apply to methods (scope, access qualification, declaration, definition) therefore apply to the destroyer.
•A class can only have one destructor. It is always unparameterized.

VII.Declaring and defining a destructor

Example 1:

```
class Time {
int Hour;
int Minute;
int Second;
public :
Time (){ .... }
Time(int H, int M, int S){ .... }
Time (const char* str){ .... }
~Time(){ } // Empty destructor
Display(){ .... }
} ;
```

Rk. The class *Time* has ***auto'***attributes. So destruction of an object of this class does not require any special additional processing.
Therefore its destructor isdefined empty.

VIII.Calling a Destroyer

1. Explicit call:

ObjectName.~ClassName();Or**ObjectPointer -> ~ClassName();**

This call, however, remains rarely used.

2. Implicit call:

A destructor is implicitly called when the object's lifetime has elapsed. This isthe case:

•For a local object of the auto memory class, when the object's validity domain (instruction block in which it is declared) is left.

•For a global or local object of the memory class "*static*" when the program isfinished.

•For a dynamic object created by *new* when the operator *delete* is applied to the pointer who designates it,

VIII.Calling a Destroyer

Example :The following example shows the constructor and destructor call times for auto and dynamic type objects.

```
class X {
  char ObjectName[20];
public:
  X(char* Name) {
  strcpy(ObjectName, Name);
  cout<<"Execution of the object's constructor: "<<ObjectName<<endl;
}
~X(){
   cout<<"Executing the object's destructor: "<<ObjectName<<endl;}
};
void main() {
  X x1("x1");
  X* x2 =new X("x2");
  delete x2;
}
```

VIII.Calling a Destroyer

Example :The following example shows the constructor and destructor call times for auto and dynamic type objects.

```
class X {
  char ObjectName[20];
public:
   X (char* Name) {
   strcpy(ObjectName, Name);
   cout<<"Execution of the object's constructor: "<<ObjectName<<endl;
}
~X (){
   cout<<"Executing the object's destructor: "<<ObjectName<<endl;}
};
void main() {
  X x1("x1");
  X* x2 =new X("x2");
  delete x2;
}
```

```
Executing the object's constructor: x1
Executing the object's constructor: x2
Executing the object's destructor: x2
Executing the object's destructor: x1
```


IIX. Ddefault destructor

If no destructor has been explicitly defined for a class, then the compiler generates a default destructor, public. This destructor performs by default the freeing of the base memory space (of the attributes). It is therefore always defined empty as follows:next: **~ClassName(){}**

Noticed :

The default generated destructor is generally suitable for simple classes (Example: the class*Time*). However, complex classes that require additional processing when they are released require an explicit and custom definition of their destructor as shown in the following paragraph.

IX. Application exercises

Exercise 1

1 - Propose a specification and an implementation of a class *Date* comprising three attributes: *day*,*month* and *year.* The class must include:
- Two constructors, one without parametrers and the other allowing to initialize a date through three integer type parameters.
- A destroyer.
- Two methods for entering and displaying a date.
- A method for comparing two dates. Propose two versions of this method:
    - o A version that returns the most recent date.
    - o A version that just returns an integer indicating the most recent date.

2 - Propose a main function showing the possibilities of using this class.

IX. Application exercises

Exercise 2

1 - Propose a specification and an implementation of a class n*o body c*comprising three attributes: *Name*, *First name a*nd *Date of birth.* The class must include:
- Constructors allowing you to create an object of type in different ways *Person*.
- A destroyer.
- Two methods of entering and displaying information about a *Person*.

2 - Propose a main function showing the possibilities of using this class.