University of Monastir

Course: C++ Programming

Chapter 4: Constructor Types in C++

Directed by:
Dr. Sakka Rouis Taoufik
https://github.com/srtaoufik/coursCpp

I.Introduction

There are three types of constructors in C++ that are a bit unusual. Their uniqueness lies in the fact that they can be called automatically in ways that are sometimes hidden from the compiler. These constructors are:

•the default constructor,
•the Constructor by copy,
•the cast constructor.

Chapter 4: Constructor Types in C++

II.Default constructor

Definition:A default constructor is any constructor that can be called without passing itof arguments.

Example :

```
#include <iostream.h>
#include <stdlib.h>
#include <iomanip.h>
class Time {
    int Hour;
    int Minute;
    int Second;
public :
   Time(){
       Hour = 12; Minute = 0; Second = 0;
   }
   Time(int H, int M, int S){
       Hour = H; Minute = M; Second = S;
   }
```

```
void Display() {
  cout<<"The time is: ";
  cout<<setw(2)<<setfill('0')<<Hour<<':'
 <<setw(2)<<setfill('0')<<Minute<<':'
 <<setw(2)<<setfill('0')<<Second<<endl;
 }
};
…
Time T;
// T contains
//Hour=12, Minute=0, Second=0;
```

Chapter 4: Constructor Types in C++

III.Constructor by copy

Definition

•A copy constructor is a constructor that allows you to create an object and initialize it with the contents of another object of the same class.

•Using copy constructors is of great importance especially with objects that have **dynamics** attributes.

4

III.Constructor by copy

Simplified syntax:

ClassName(<const> ClassName &){....}

Example :

```
class Time { ...
public :
Time(const Time& T){...} // Copy constructor
...
};
```

Remarks:

- A copy constructor can have other additional parameters. These must have default values.
- A copy constructor is always called with a single argument (the first argument). Any additional arguments are called in aimplicit.

III.Constructor by copy

Why a first parameter of reference type?

The question this remark answers is: why do we pass a reference to the object to be copied to the copy constructor and not a copy of that object? In other words, why do we use pass-by-reference instead of pass-by-value? seems sufficient?

III.Constructor by copy

Justification of the passage by reference:

Consider the class*Time* defined previously. We add to this class a copy constructor which takes a Tm as parameter.

```
class Time { ...
public :
Time(){...}
Time(Time Tm){...} // We assume it's possible
...
};
```

III.Constructor by copy

Justification of the passage by reference:

Let *f* a function that takes a parameter *Tf* of type *Time* and *X* an instance of this *Time class*:

```
void f (Time Tf){...}
Time X;
```

The call of *f* with the effective parameter *X* generates the copy of *X* in the formal parameter *Tf*. This copy is made with the Constructor by copy of *Time*.

```
f(X) // Tf=X ⇔ Tf(X)
//Call to Time's copy constructorand consequently:
f(X) ⇔ f(Time(X)) // instantiation of the formal parameter using the
// copy constructor.
```

III.Constructor by copy

Justification of the passage by reference:

The passage of *X* to the constructor *Time* is done by value. This constructor will there forework on a copy of *X*. This copy will be created by another call to the copy constructor. The latter itself works on a copy of *X*, it will also make a call to itself. This recursive call of the copy constructor will continue in definitely.

Time(X) // Tm=X ⇔ Tm(X)

and consequently Time(X) becomes Time(Time(X))

and f(X) becomes f(Time(Time(Time ………)))

III.Constructor by copy

Justification of the passage by reference:

To solve this problem, simply avoid the implicit call to the copy constructor that is generated by passing by value and replace it with passing by reference.

So with Time(Time& Tm), f(X) reduces only to f(Time(X))

because the assignment of the actual parameter to the formal parameter (Time& Tm=Time(X))

represents a copy of references and not a copy of objects.

III.Constructor by copy

Why the recommendation const

Using a constant reference as the first parameter in copy constructors is a recommendation.

The interest of this recommendation lies in the protection of the object passed as an argument against any modification which can occur by mistake in the constructor, especially since its passage is done by reference (we do not work on a copy of the object but directly on the latter).

III.Constructor by copy

Note: Default copy constructor

If an explicitly defined copy constructor is not present, the compiler automatically generates one by default. This constructor performs a field-by-field copy of the contents of both objects.

Limitations of the default copy constructor

Example 1:

```
class TabIntegers {
public:
  intNb;
  int* T;
 TabIntegers(int Neither) {...}
 TabIntegers(int* You, int Neither){...}
 voidToinput(){...}
 void Display(){...}
};
```

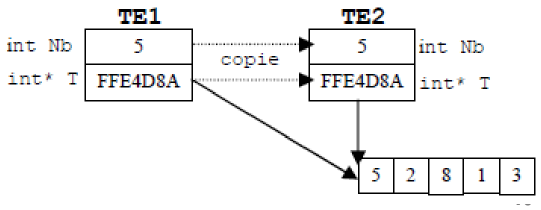
```
void hand() {
  intTi[4]={5,2,8,1,3};
 TabIntegersTE1(Ti,4);
 TabIntegersTE2(TE1);
 TE2.Display();
}
```

Chapter 4: Constructor Types in C++

III.Constructor by copy

The class*TabIntegers* does not have an explicitly defined copy constructor. Therefore, the default copy constructor is used in the instruction:TabIntegers**TE2** (TE1);

➔ This constructor performs a field-by-field copy of the contents of*TE1*In*TE2*. Therefore, for the dynamic member*T*, it is a copy of addresses which is carried out between*TE1.T*And*TE2.T*.

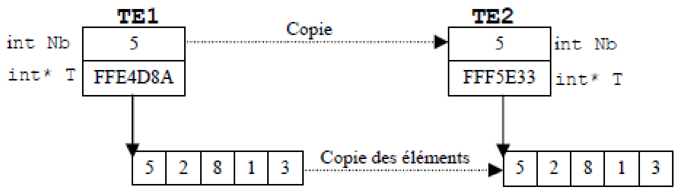


Chapter 4: Constructor Types in C++

III.Constructor by copy

Solution:it is necessary to define explicitly the Constructor by copy as follows:

```
TabIntegers::TabIntegers (const TabIntegers & Tc) {
  Nb=Tc.Nb;
 T=new int[Nb];
  for(intk=0;k<Nb;k++)
   T[k]=Tc.T[k];
}
```



14

Chapter 4: Constructor Types in C++

III.Constructor by copy

Where to call an object copy constructor

A copy constructor is called when:

•of creating an object using another object of the same class.

Example:

```
Point P1; // default constructor
Point P2=P1; Point P3=Point(P1); // copy constructor
```

•from the transmission of the value of an actual parameter (of object type) to a formal parameter following the call of a function.

•of returning an object type value from a function. In fact, the function creates a temporary object without a name using the copy constructor, from the object passed to*return*and it is this temporary object that is assigned to the variable receiving the return valueof the function.

Chapter 4: Constructor Types in C++

III.Constructor by copy

Where to call an object copy constructor

```
class NoData {
public:
  NoData() {cout<<"default constructor";  }
  NoData (const NoDat a& ND) {cout<<"copy constructor";  }
};
void f1(NoData D){ }
void f2(NoData& D){ }
NoDataf3() { NoData D; ...return D; }

void hand(){
 NoData X;// Default constructor
 f1(X); // Copy constructor (due to pass-by-value)
 f2(X); // Nothing is displayed
 X=f3(); // Default constructor (for local object D)
           // Copy constructor (for the returned object)
}
```

III.Constructor by copy

Where to call an object copy constructor

Question :

The following prototype for*f3*: NoData & **f3**() allows to achieve a correct result even in the absence of an explicit copy constructor?

Answer :

The copy constructor is not called but we obtain a reference to an object which no longer exists in memory since it is local to the function.

IV.Typecasting constructors

A typecast constructor, also called a conversion constructor, is a constructor that allows you to create and initialize an object of type T1 from data.

(object or not) of another type T2 different from T1.

Features of the typecast constructor

•A typecast constructor can only be called with a single parameter: as is the case with the copy constructor, any parameters that follow the first formal parameter must have default values.

•The first parameter of the cast constructor contains the data to be converted. This parameter can be of any type (prefinite, struct, class, etc.) except the type of theconstructor class.

Chapter 4: Constructor Types in C++

IV.Typecasting constructors

General syntax:

ClassName(<const> Type1 Param1<,Type2 Param2=DefVal2,..,Type_n Param_n=DefVa_n>);

Example :

```
class Time {
  int Hour;
  int Minute;
  int Second;
public :
  Time(){Hour = 12; Minute = 0; Second = 0;}
  Time(int H,int M, int S){Hour = H; Minute = M; Second = S;}
  Time(const char* str){
    cout<<"Call the cast constructor";
    Hour = 10*(*str-'0') + (*(str+1)-'0');
    Minute = 10*(*(str+3)-'0') + (*(str+4)-'0');
    Second = 10*(*(str+6)-'0') + (*(str+7)-'0');
   }
  void Show(){...}
};
```

Chapter 4: Constructor Types in C++

IV.Typecasting constructors

General syntax:

The constructor Time(const char* str) can be considered as a typecast constructor since it meets the two characteristics of this type of constructor, namely a call using a single parameter, in addition the parameter has a different type than

Time (const char*).

IV.Typecasting constructors

Scopes of use of typecast constructors

Explicit use:A typecast constructor can be explicitly called by the programmer whenever the programmer needs to work with an object of a certain type and has adata of another type.

Example

Time X = Time("18:30:12"); or also Time X("18:30:12");

Implicit use:A typecast constructor from a type T1 (object or not) to an object type T2 different from T1 is implicitly called by the compiler:

•When passing an actual parameter of type T1 to a function having a formal parameter of type T2.

•In an assignment operation having a right-hand member (object, variable, or expression) of type T1 and a left-hand member of object type T2.

X.Constructor Initialization Lists

A constructor initializer list represents a syntactic alternative for initializing the attributes of a class.

➔ Instead of doing this initialization in the body of the constructor, it becomes possible thanks to this alternative to do it just before the constructor header using the following syntax:

ClassName(arguments): A1(V1), A2(V2), …, An(Vn)

When :

•ClassName(arguments) represents a constructor call,

•Ai: represents the attributes of ClassName to be initialized

•Vi: initial value assigned to the Ai attribute.

X.Constructor Initialization Lists

Example 1:

```
// Definition of the constructor of the Time class
Time():Hour(12),Minute(30),Second(45) { }
```

Example 2:

```
// Definition of the constructor of the TabEntiers class
TabIntegers (int N, int* Tab): Nb(N),T(new int[N]) {
   for(int i = 0;i<Nb;i++)
     T[i] = Tab[i];
}
```

X.Constructor Initialization Lists

Initialization of constant attributes and references

For constant and reference attributes, initializer lists are not a simple code writing alternative but rather the only way to assign their initial values.

Consider the class *ConstRef* which has the following three attributes:

x(entire), *r* (reference to an integer) and *y* (integer constant).

```
class ConstRef {
 int x;
 int& r = x ; // ERROR
 const int y = 5; // ERROR
… … …
};
```

X.Constructor Initialization Lists

Initialization of constant attributes and references

The fact that*r*And*y*are respectively a reference and a constant makes their initializations mandatory. However, their initializations as carried out in the class*ConstRef*are prohibited. Indeed, the definition of a class constitutes a type definition without any instantiation in memory.

➔ At the time of definition of *ConstRef* no attribute has a physical existence in memory. Therefore, initializing r with the reference of an x variable, which does not exist or also assign 5 to a constant *y* which do not exist in memory represent prohibited operations.

X.Constructor Initialization Lists

Initialization of constant attributes and references

The use of a constructor defined as follows is also prohibited:

```
ConstRef::ConstRef( ) {
  x = 3; // OK
  r = x; // ERROR
  y = 5; // ERROR
}
```

In fact, the instruction r=x; represents an assignment of the value 3 to the variable referenced by *r*. Gold*r*does not reference any variable because it has not been initialized. Furthermore, the instruction y=5; represents an assignment of the value 5 to the constant *y* and not an initialization. However, assignment operations cannot be applied to constants.

Chapter 4: Constructor Types in C++

X.Constructor Initialization Lists

Initialization of constant attributes and references

The only way inC++ allows for initializing such members by using lists initialization.

Example :

```
class ConstRef {
  int x;
  int & r ;
  const int y ;
  ConstRef( ) : r(x),y(5) {
    x=3;
   }
};
```

27

Chapter 4: Constructor Types in C++

X.Constructor Initialization Lists

Initializing Member Objects

Initializer lists are also used to initialize members that are of class type (object members) mainly when they have constructorsconfigured.

Initializing member x as follows: x(55) is possible because a constructor with prototype X (int i) is defined for class X.

Example :

```
class X {
  int a;
public :
  X(int i) {
    a=i;
    cout<<"Constructor of X\n";
  }
};
```

```
class Y {
  int b;
  X x;
public :
  Y():b(5),x(55) {
    cout<<"Constructor of Y\n";
   }
};
```

28

X.Constructor Initialization Lists

Initializing Member Objects

Rq. Considering the definition of class X, the following definition of the constructor of Y is not correct:

```
Y() {
   b=5;
   x=X(55);
   cout<<"Y constructor";
}
```

Indeed, the instruction x=X(55); does not syntactically constitute an initialization in the true sense of the term. Rather, it is considered to be a first assignment to x of the contents of the temporary object X(55). This assumes that x has been previously created using a default constructor, something that is not possible because there is no such constructor in the definition of class X.

XI.Exercisesapplication

Écrivez une classe Vecteur comportant :
comme membres données privées : trois composantes de type double, et un donnée membre **statique** nb représentant le numéro du vecteur (le premier créé portera le numéro 1, le suivant le numéro 2, et ainsi de suite, …) ;

- ✓ trois constructeurs :
- l'un, sans arguments, initialisant chaque composante à 0,
- l'autre, avec 3 arguments, représentant les composantes,
- le dernier par recopie
- ✓ un destructeur qui affiche le numéro du Vecteur détruit ;
- ✓ une fonction homothétie pour multiplier les composants par une valeur fournie en argument.
- ✓ une fonction prod-scal qui fournit en résultat le produit scalaire de deux vecteurs,
- ✓ une fonction somme permettant de calculer la somme de deux vecteurs.
- ✓ une fonction amie module qui fournit le module d'un vecteur $\sqrt{x^2 + y^2 + z^2}$
- ✓ une fonction affiche qui affiche le numéro, les coordonnées et le module d'un vecteur

Écrivez ensuite une fonction main qui permet de les tester.

XI. Application exercises

Let the class ***Matrix*** whose specification is given as follows:

```
class Matrix {
 const int NbLines;
 const int NbColumns;
 int** Elements;
public :
  Matrix (int L, int C);
  Matrix (int L, int C, int** Elts);
  Matrix (const Matrix & M);
  ~Matrix();
  void Set (int LineIndex, int ColumnIndex, int Value);
  int Get(int LineIndex, int ColumnIndex);
  void Show(); // Displays the contents of the matrix
  void Set(); // performs the capture of all elements
};
```

1- Give the definitions of the class methods ***Matrix***.

2- Give a main function that shows the instantiation of the class ***Matrix*** and the use of its different methods.

XI. Application exercises

1- Propose a function ***Multiply*** external and friend to both classes ***Vector*** and ***Matrix*** which performs the product of a matrix and a vector.

2- Transform the friend function ***Multiply*** so that it becomes a method of the class ***Matrix.*** Indicate how the code organization needs to be changed to compile successfully.