4. Constructors and destructors

The creation and the destruction of the objects represent an important operation in order to realize for safety and stable programs. The programmer which designs a set of class related to a program must take in consideration all the possibilities in which the class instances can be created, initialized, and destroyed, in order to write a simple and safety code.

The ***constructors*** and ***destructors*** are special member functions of the classes which are responsible for the actions as: object *initialization* of the data members of the class instances, object *copying*, and memory *deallocation* for the additional allocated memory. Because of their particularities, these functions present characteristics of the common member functions of the classes, but they have also in addition some specific properties.

The first specific characteristic of these functions is represented by their name. In the standard C++ language it is used the Stroustrup solution in which the constructors are named by the class name where they belong, and the destructors are named by the class name preceded by the '~' character. This solution is natural, because the constructors and the destructors are called in automatic mode by the compiler and their name must be predetermined.

Another particularity is that if a class does not contain in its declaration constructors and/or destructors, some of these functions are automate generated by the compiler.

The constructors and destructors do not return values, not even of the `void` type, which made them special by comparing with the others functions; otherwise the compiler should know what to do with the returned value.

4.1 Constructors

There are two distinct situations when the class constructors are called: (a) when an object, instance of a class is initialized and some of data members of the newly created object are initialized with certain values, or (b) when a newly created object is initialized with the values of the data members of another already existent object from the same class.

Remark. The last variant should not be confused with the case of an assigning statement; as in the C language, the initialization of an object can realized at its definition:

```
// i,j variable definition and j variable initialization
int i, j = 3;
// assignment operator
i = 4;
```

In fact the notions of defining and initializing an object are related one with another and they can not exist separately.

Example 4.1. The class *time* allows the determination of a time interval passed from an initial date of the form year-month-day-hour-minute-second, to the current date, considering the time measured in seconds.

```
class time {
  int hour, minute, second;
  double t;
  static int hour_0, minute_0, second_0;
  void SetTime() {
    t = 3600 * (hour - hour_0) +
          60 * (minute - minute_0) + second - second_0;
  }
public:
  time (int Hour = 0, int Minute = 0, int Second = 0)
  {
    hour = Hour;
    minute = Minute;
    second = Second;
  }
  double GetTime() {
    SetTime();
    return t;
  }
};

int time::hour_0 = 0;
int time::minute_0 = 0;
int time::second_0 = 0;

void Problem() {
  time m1(7, 3, 24);
  time m2(20, 4, 12);
  cout << ″t1= ‟ << m1.GetTimp() << endl;
  cout << ″t2= ‟ << m2.GetTimp() << endl;
}
// ...
```

The creation of an object has two distinct parts: the allocation by the compiler of a uninitialized memory block having an appropriate size, and the calling of a constructor of respective object class. The allocation operation is transparent to the programmer. The memory zone where the memory is allocated depends on the mode of creating the object (in the static data zone, stack zone, or in the heap zone).

The role of a constructor is to initialize certain data members of the object. To perform this action, the memory address of allocated zone for the object is passed to the constructor by using the hidden parameter `this`.

In the previous example there exist two implicit calls at the constructor of the class *time*, what means in fact that the compiler inserted in the place of the two definitions the following calls:

```
m1.time(&m1, 7, 3, 24);
m2.time(&m2, 20, 4, 12);
```

When an object is defined as a global variable defined outside any function, the constructor of such an object is called before the execution of the *main* function.

Related to the memory allocation for a newly object, we distinguish three types of allocation:

a) in the *static allocation zone*, for the external objects defined outside any function of a program; in this case the constructor is called before the *main* function, and the lifetime of the object corresponds to the execution time of the program;
b) in the *stack zone* of the program, in the case of local objects defined inside the blocks; the allocation and the constructor call is performed when the program execution reaches the respective object definition; the lifetime of such an object is the time for which the defined block the object is active on the stack;
c) in the *heap zone* of the program, in the case of dynamic created objects by using the `new` operator; the lifetime of such an object corresponds to the time between the consecutive call of the pair operators `new` and `delete`, related to the same pointer.

In the first two cases the object destructors are automatic called by the compiler, while in the last case the destructor is implicitly called with the help of `delete` operator. In the third case, the call of the constructor for the dynamic created object is automatic performed by the compiler from the actions performed by the `new` operator. For example, the object initialization of a dynamic object of the *time* class can be realized as follows:

```
// pointer declaration; no 'time' object is created
time *pt;
// the creation and calling of the 'time' constructor
pt = new time (7, 3, 1);
```

or:

```
// both pointer declaration and object creation
time *pt = new time (7, 3, 1);
```

All the `new` operator advantages referring to classic functions of allocation memory from the C language specified in chapter two are valid also in the case of dynamic allocation for objects.

4.2 Constructor types

There is also one situation when objects are implicitly created by the compiler, in the case of *function call*, when: (a) the actual values of parameters are passed to the called function, and (b) the calculated value of the called function is returned to the calling function, by using the `return` statement.

The parameter passing is performed in the C++ language in two distinct ways: by value and by reference. The *passing-by-reference* does not create supplementary objects, while the *passing-by-value* involves the passing of a copy of the actual parameter of the calling function to the formal parameter of the called. In other words, a new (temporary) object is created, which is a copy of the object representing the actual parameter, and this object is used as the initializing value for the corresponding formal parameter.

The inverse operation, returning from a called function, involves the creation of a new object, which represents the returned value by the function.

In the cases described above (but not only in these cases), another type of constructor is used, which is called ***copy constructor***. This is called any time an object creation imposes its initialization with another object from the same class.

In general, the constructors can be divided in the following categories:

a) general constructors;
b) default constructors;
c) copy constructors;
d) conversion constructors.

Usually a class may have more than one different constructor, which allows the creation of objects in different cases.

4.2.1. General constructors

The ***general constructors*** are constructors that have at least one argument, which is not a reference at the respective class type, the argument values being used for initialization of the data members of the created object. The great majority of constructors from the previous example have been general constructors.

Denoting with *X* the current class name and with *T1*, *T2*, …, etc., the data types of the arguments, the declaration of general constructor has the following form:

```
X(T1, T2, /*...*/);
```

Because the constructors are C++ language functions, all the specific properties of functions are valid in the case of constructors. For example, it is allowed for these functions to have parameters with default values (a useful way, which increases the efficiency of the class design and implementation).

The constructor of the class *time* is an example of parameter with implicit values for parameters. The next definition creates three objects of *time* type:

```
time o1(7, 3, 2);
time o2(7, 3);
time o3;
```

Remark. In the case when a general constructor has default values for arguments, these values must be specified in the class definition and not in the implementation part.

4.2.2 Default constructors

Default constructors do not have arguments, having the following form:

```
X (void);
```

for a class having the name *X*.

Default constructors, as the copy constructors, are *the only constructors* that can be automatic generated by the compiler in the case when a class does not have any constructor.

Example 4.2:

```
class time {
  // ...
public:
  time();
  // ...
};

time::time() {
  cout << ”Fill in with values for hour, minute, second: “;
  cin >> hour >> min >> sec;
}

void processing() {
  // ...
  time t;
  // ...
}
```

Remarks:

1. The class constructors can be overloaded.
2. A general constructor with all the arguments having default values it is not a implicit constructor.
3. The compiler does not generate a default constructor for a class that has at least one other constructor.

Because a default constructor and a general one with default values for all parameters are called with the same syntax, they do not have to be defined together in the same class.

The next sequence contains an error related to the definition of the constructors:

```
class time {
   // ...
public:
  time(int h = 0, int m = 0, int s = 0);
  time();
  // ...
};
```

because the next definition is not clear:

```
time t;
```

A special attention is imposed for the classes having no constructor (not even one), because the default generated constructor by the compiler do not perform any member initializing.

Example 4.3. The next sequence has an error, because the *s* data member is not initialized at the creation of the *String* class objects.

```
#define MaxString 100
class String {
  char s[MaxString + 1];
public:
  void set(const char str[]);
  const char* get() { return s; }
};
// ...
int main() {
  String s1;                     // ‘s’ it is not initialized
  cout << s1.get() << endl;      //memory access error!!
  // ...
}
```

A correct variant of the precedent sequence is writing a default constructor, which creates an empty string:

```
#define MaxString 100
class String {
  char s[MaxString + 1];
public:
  String() { s[0] = ‘\0’; }
  void set(const char str[]);
  const char* get();
};
```

Another used utilization of the default constructors refers the initialization of the array of objects. When a program contains a definition of an array of objects belonging to a certain class, if the array is not explicit initialized, for each component of the array the default constructor of the respective class is automatic called by the compiler.

Example 4.4. The next program:

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

unsigned int n = 0;

class A {
public:
  A() { cout << ”Constructor for A object” << ++n << endl; }
};

A v[7];

int main() { return 0; }
```

generates the following output:

```
Constructor for A1 object
Constructor for A2 object
Constructor for A3 object
Constructor for A4 object
Constructor for A5 object
Constructor for A6 object
Constructor for A7 object
```

Remark. An important attention is imposed when a class has at least one constructor, but none constructor is a default constructor. In this case the compiler will not generate a default constructor for this class. So, when it is needed a default constructor (as in the precedent example), the compiler will generate an error.

Example 4.5. The next program will generate an error:

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

class A {
public:
  A(char *str)
  {
    cout <<″ A object″ << ″ and ″ << str << ‟ string ‟ << endl;
  }
};

//…
A a;
```

4.2.3 Copy-constructors

The ***copy-constructors*** represent an important class of constructors. In the case when into a class declaration it is not specified any copy-constructor, the compiler will automatically generate such a constructor.

The role of such a constructor can be judged by analogy with initialization when defining variable. For example, the definition:

```
int k = 3;
```

involves to two distinct actions: the allocation of a memory zone for the variable *k*, and also the initialization of the respective memory zone with the value 3.

In a similar way an object of a class can be initialized with the date members of another created object belonging to the same class. For example:

```
time t(1, 0, 0);
time t1 = t;
```

In the above case, initializing of the *t1* object is performed by copying values of the data members of the *t* object by using the copy-constructor:

```
class time {
  // ...
public:
  time(const time& t) {
    hour = t.hour;
    min = t.min;
    sec = t.sec;
  }
  // ...
};
```

Remarks:

1. Always, the first argument of a copy-constructor must be a *reference* to an object of the current class, or a reference to a constant object of the current class.
2. In the case when a copy-constructor has in addition other parameters, all these parameters must have default values; if this is not the case we have a general constructor. This restriction is due to the syntax of the call of a copy-constructor:

   ```
   ⟨class⟩ ⟨object1⟩ = ⟨object2⟩ ;
   ```

Example 4.6. In the next sequence, at the initialization of the *x2* object with the values of the *x1* object cannot be specified other initializing parameters.

```
class X {
  // ...
  int a;
public:
  X(){ a = 0; }
  X(X& x, int k = 0) {
    a = x.a;
    // ...
  }
  // ...
};

// ...
X x1;
X x2 = x1;
X x3(x2, 5);
// ...
```

A copy-constructor generated by a compiler, usually, will do a member by member copy of the data members of the object. In the case of *time* class, the copy-constructor generated automatically by the constructor is identical with the one explicit defined in the code.

There are cases when a copy-constructor, implicitly generated by the compiler it is not sufficient for a correct initializing of the current object, especially in the cases when exists pointer type member data, or certain member data are objects of other classes.

Example 4.7 The class *list* implements a simple single linked list, and the class *node* implements the structure of the elements of the list.

```
struct node {
  int val;
  node* next;
  node() {val = 0; next = 0;}
```

```
  node(int v, node* n = 0) {
    val = v;
    next = n;
  }
  // copy-constructor implicitly generated
  ~node(){ next = 0; }
  void Add (int);     // adds a node after the current node
  void Print() const { cout << val << endl; }
  // ...
};

struct list {
  node* first;
  void Copy(list& l);
  void Delete();
  list() { first = 0; }
  list(list&);
  ~list();
  list& operator=(list&);
  node* Last() const;
  void Add(int);      // adds an element at the end of the list
void Print() const;
  // ...
};

void node::Add(int k) {
  node* p = new node(k);
  next = p;
};

void list::Copy (list& l) {
  node* p = new node(l.first->val);
  first = p;
  for (node*q=p->next; p; p=p->next)
    Last()->Add(q->val);
}

node* list::Last() const {
  node* p;
  for(p=first; p->next; p=p->next);
    return p;
}

void list::Add(int k) {
  if (first)
    Last()->Add(k);
  else {
    node *p = new node(k);
    first = p;
  }
}

void list::Print() const {
  for (node* p=first; p; p=p->next)
    p->Print();
}
```

```
void list::Delete() {
  // will be further implemented (to destructors)
}

list::list(list& l) {
  Copy (l);
}

list::~list() {
  Delete();
  first = 0;
}

list& list::operator=(list& l) {
  if(&l != this) {
    Delete();
    Copy(l);
  }
  return *this;
}

// ...
```

Remark. A copy-constructor of a class *X* must have as parameter a reference to an object of *X* class (having the type *X&*) or to a class itself (a parameter of the *X* type).

A copy-constructor is not called only at object initialization with values of other objects, but also in the case of parameter passing mechanism at the function call.

In the case of *passing-by-value*, a temporary copy of the object which is actual parameter is created, which is then passed to the corresponding formal parameter in the called function. When the called function returns to the calling function by using the `return` statement, the value that represents the returned object is passed back to the calling function by returning a copy of that object (so another object is created by the help of the copy-constructor).

Example 4.8. A function which creates a new list formed from the first and the last element of an existent list.

```
list FirstLast (list l) {
  list l1;
  l1.Add(l.first->val);
  l1.Add(l.Last()->val);
  return l1;
}
void Processing() {
  list l1, l2;
  l1.Add(3);
  l1.Add(7);
  l2 = FirstLast(l2);
  // ...
}
```

When the function *FirstLast* is called, the copy-constructor for the *l* parameter is called, which has as parameter a reference of the *l2* object. This temporary object will be destroyed after the exit from the *FirstLast* function.

The statement `return` has the following effect: the automatic creation of an additional object of the type *list* by copying the object *l1*. This new created object represents the object which will be returned to the *Processing* function and which is taken by the assignment operator.

4.2.4 Conversion constructors

A ***conversion constructor*** is usually a constructor with only one argument (as the copy-constructor), but its type is different to the current class. In the case when exists more parameters, these parameters must be all with default values (in order to be considered as a conversion constructor).

For example, the second constructor of *node* class from the previous example is a conversion constructor. So, a general constructor is considered to be either a constructor with all the parameters with default values, or a constructor having at least two parameters with no default values.

The conversion constructors are frequently used by the compiler for doing ***the default conversion of data types.*** Usually, any time when an operand from an expression does not respect the data type of the respective expression, an automatic conversion of the operand type to the expression type is tried. For example, for predefined types, in the next expression a conversion from `int` to *`double`* is performed:

```
int n = 3;
double x, y = 2.5;
x = y + n;
```

A similar conversion is performed also for objects; the compiler is trying to find a appropriate conversion constructor.

Example 4.9. A conversion constructor for the previous *String* class is defined:

```
#include <string>
#include <iostream>
using namespace std;
#define MaxString 100

class String {
  char s[MaxString + 1];
public:
  String() { s[0] = '\0'; }
  // conversion char* -> String
  String(const char str[]) { strcpy(s, str); }
  void set(const char str[]);
  const char* get() { return s; }
};
// ...

void f(String s) { cout << s.get() << endl; }
```

```
int main() {
  String s1;
  f(s1);         // copy constructor
  f("abc");      // conversion constructor
  // ...
}
```

At the second call of the function *f*, the conversion constructor is used to convert the '*abc*' string to an object of the class *String*, which will be passed as parameter. At the first call of *f* the copy-constructor is used.

4.3 Destructors

The ***destructors*** are used to free the additional memory zones occupied by the members of certain objects, before freeing the memory for the respective object. As in case of constructors, the deallocation of the memory of an object does not represent an action of the destructor.

The destructor is used usually in the case when objects use dynamic allocation for certain data members of them. In the case when a class does not contain an explicit definition of a destructor, the compiler will implicitly generate a destructor for it.

The destructors, unlike constructors, cannot have arguments. In addition, the destructors cannot be overloaded; each class must have exactly one destructor.

The destructor call for an object, when it does not have allocated a dynamic memory zone (it was not allocated by using the `new` operator), is performed automatically by the compiler: (a) for local objects defined inside the blocks, the destructor is called at the exit from the current block where the object where defined, and (2) for global objects defined outside any function, the destructors are called after the exit from the *main* function, or when it is explicit called by the function *exit*.

Example 4.10.

```
#include <iostream>
using namespace std;
class X {
  int k;
public:
  X(int i) {
    k = i;
    cout << ″x() for ″ << k << endl;
  }
  ~X() { cout << ″~x() for ‶ << k << endl; }
};

X ob1(5);

void f() {
  cout << ″starts the function f″ << endl;
  static X ob2(7);
```

```
    X ob3(9);
    Cout << ”finishes the function f” << endl;
  }

  int main() {
    cout << ”starts the main function” << endl;
    X ob4(11);
    f();
    cout << ”finishes the main function” << endl;
    return 0;
  }
```

The program execution generates the following sequences:

```
x() for 5
starts the main function
x() for 11
starts the function f
x() for 7
x() for 9
finishes the function f
~x() for 9
finishes the main function
~x() for 11
~x() for 7
~x() for 5
```

From the previous example it can be seen that in the case when there are several elements to be destroyed, the destructors are called in reverse order as for constructors.

Also, the constructors for a local static object is called at the first meeting with the object definition, but its destructor is called after the exit from the *main* function (which corresponds to the lifetime rule for static object).

In the next example it is observed the constructors and destructors call in the case of *pass-by-value* of the objects as arguments in the function call.

Example 4.11. A class which counters its object instances:

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

class Contor {
  char c;
  static int contor;
public:
  void Print()
    { cout << "object " << c << " contor " << contor << endl; }
  Contor(const char& ch) {
    c = ch;
    ++contor;
    cout << "Conversion constructor: ";
    Print();
  }
  Contor(const Contor& h) {
    c = h.c;
    ++contor;
    cout << "Copy-constructor: ";
    Print();
  }
  ~Contor() {
    --contor;
    cout << "Destructor: ";
    Print();
  }
};

int Contor::contor = 0;

Contor f(Contor x) {
  cout << "Starts the f function” << endl;
  cout << "Finishes the f function" << endl;
  return x;
 }

int main() {
  Contor o1('a');
  cout << "Before the f call with return value" << endl;
  Contor o2 = f(o1);
  cout << "After the f call" << endl;
  cout << "Before the f call without return value" << endl;
  f(o1);
  cout << "After the f call without return value" << endl;
  return 0;
}
```

Program output:

```
Conversion constructor: contor 1 object
Before f call with return value
Copy-constructor: contor 2 object
f function starts
f function finishes
Copy-constructor: contor 3 object
Destructor: contor 2 object
After the f call with return value
Before the f call without return value
Copy-constructor: contor 3 object
f function starts
f function finishes
Copy-constructor : contor 4 object
Destructor : contor 3 object
Destructor : contor 2 object
After the f call without return value
Destructor : contor 1 object
Destructor : contor 0 object
```

It is observed that the initialization of the argument *x* of the function *f* is done with the help of the copy-constructor. The *x* parameter becomes a temporary object local to the function and it will be destroyed after the function execution finishes and it returns to the *main* function.

The evaluation of the expression related the statement `return` generates a second temporary object (the value which must be returned) which is created with the help of copy-constructor. In the case when the function returns a value, this object is not destroyed (it representing in fact the *o2* object from *main* function). In the case when the function does not return any value, this object is destroyed after the function call and before the returning to the *main* function (at the second call there exist two destructors successively called, one for the temporary object and another for the returned value).

In the case of using pointers, the constructors and destructors must be explicit called with the help of `new` and *`delete`* operators.

Remarks:

1. Even if a pointer exits from his scope, if the `delete` operator is not called, the associated object to the pointer will not be destroyed (the destructor is not implicit called).
2. In the case when at the end of the program execution there are remained objects allocated in the heap zone, the compiler forces the destructor call for these objects after the exit from the *main* function.

Example 4.12. The destructor for list class from the previous example:

```
struct list {
  node* first;
  void Copy (list& l);
  void Delete();
  list() { first = 0; }
  list(list&);
  ~list();
  // ...
};

void list::Delete() {
  for(node* p=first; p ;) {
    node*q = p->next;
    delete p;
    p = q;
  }
}

list::~list() {
  Delete();
  first = 0;
}

void Processing() {
  list* l1 = new list;
  l1->Add(3);
  l1->Add(7);
  l1->Print() ;
  // ...
  delete l1;
  // ...
}
```