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 standard C++ language uses the Stroustrup solution:
  - ***Constructors are named by the class name*** where they belong,
  - ***Destructors are named by the class name preceded by the character '~'***.

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

- Constructors and destructors ***do not return values***, not even of the `void` type, which made them ***special functions*** by comparing with the others functions.

A. Constructors

- The creation of an object has two distinct parts:
  - ***Allocation by the compiler of an uninitialized memory block*** having an appropriate size (operation transparent to the programmer),
  - ***Calling of a constructor*** of the respective class.

Example. The class *time* allows the determination of a time interval passed from an initial date of the form *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;
}
```

- In this example there ***are two implicit calls*** of the constructor of the class *time*. The compiler ***inserted*** in the place of the two definitions a sequence similar to:

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

- 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 fact there are 4 parameters passed to the constructor):

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

- The moment when ***constructors*** and ***destructors*** are called depends on the type of the memory allocation, and on the places on the program where objects are defined:

  a) ***for allocation in the static data zone***:

  - for the ***external objects*** defined outside any function of a program (the life cycle of the object is the same as the life cycle of the program):

- ***the constructor*** is called ***before*** the execution of the function **`main`**,
- ***the destructor*** is called ***after*** the finish of the function **`main`**;

- for ***static local objects***:
  - ***the constructor*** is called at ***the first declaration*** of the object,
  - ***the destructor*** is called ***after the finish of the function*** **`main`**;

b) ***for allocation in the stack zone*** of the program, in the case of ***local objects*** defined inside the blocks (the life cycle of such an object represents the time when the block is active on the stack):

- ***the constructor*** is called when the ***program execution reaches*** the respective ***object definition***,
- ***the destructor*** is called ***after leaving*** the current block;

c) ***for allocation in the heap zone*** of the program, in the case of ***dynamic objects***, ***created*** by using the **`new`** operator, and ***deleted*** by using the **`delete`** 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):

- ***the constructor*** is called when the operator **`new`** is called,
- ***the destructor*** called when the operator **`delete`** is called.

- In the ***first two cases*** the constructors and destructors are ***automatically*** called by the compiler, while in the ***last case*** they are ***implicitly*** called with the help of operators **`new`** and **`delete`**.

- Types of constructors:
  - ***general constructors***,
  - ***default constructors***,
  - ***copy constructors***,
  - ***conversion constructors***.

- Usually a class may have ***more than one*** different constructor, which allows the creation the ***state*** of objects.

General constructors

- Are constructors that have ***at least one argument***, which is ***not a reference*** at the respective class type (the argument values are used for initialization of the data members of the created object)

- 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, /*...*/);
```

- These constructors can have parameters with ***default*** values. In this case the default values must be specified in the ***class definition*** and not in the implementation part.

- 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;
```

Default constructors

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

```
X(void);
```

Example:

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

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

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

- ***Default constructors*** are the ***only constructors*** that can be ***automatically*** generated by the compiler in the case when a class ***does not have any constructor***

Remarks:

1. The class constructors can be ***overloaded***.
2. A ***general constructor*** with ***all arguments having default values* it *is not an 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.

Example. 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 constructors*** (not even one), because the default generated constructor by the compiler do ***not perform*** any ***member initialization***.

Example. 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
  // memory access error!!
  cout << s1.get() << endl;
  // ...
}
```

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***. 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. 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
```

Copy-constructors

- An object can be ***initialized*** with the ***values*** of anoter ***created object***.

Example. Adding a copy-constructor to the class *time*:

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

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

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. If a copy-constructor has in addition ***other parameters***, ***all these parameters*** must have ***default values***; ***otherwise*** we have a ***general constructor***. This restriction is due to the syntax of the call of a copy-constructor:

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

Example.

```
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);
```

In the case of the following definition:

```
X(X& x, int k);
```

the above constructor is no longer a copy-constructor, and the following expression is incorrect:

```
X x2 = x1;
```

- In the case when into a class declaration it ***is not specified*** any copy-constructor, the compiler will ***automatically generate*** such a constructor (not as ***default constructors***).
  - A copy-constructor generated by a compiler, usually, will do *a* ***member by member copy*** of the data members of the object.

- 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 the member data are ***pointers***, or ***objects*** of other classes.

Example: 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; }
  // adds a node after the current node
  void Add (int);
  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;
  // adds an element at the end of the list
  void Add(int);
  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;
}
// ...
```

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

Example: 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(l1);
  // ...
}
```

- 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.

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***.
- A constructor is considered as ***general***, either it has all parameters with default values, or it has 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***.

Example. A conversion constructor for the class *String*:

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

```
class String {
  char s[MaxString + 1];
public:
  String() { s[0] = '\0'; }
  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
  // ...
}
```

- In ***Python***, the special function **`__init__`** can be overloaded for each class
  - It has the same meaning as a *constructor* from C++
  - There is a single **`__init__`** function for each class
  - If a class does not contain a **`__init__`** function, it is inherited from the root **`object`** class
  - The goal of the **`__init__`** function is to initialize the instance variables
  - The first parameter related to the instance reference (ususally **`self`**) is mandatory

B. 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 ***de-allocation 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.

Example:

```
#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 output:

```
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
```

- In the case when there are several elements to be destroyed, the destructors are called in ***reverse order*** as for constructors.
- In the next example one can observe the ***call of constructors and destructors*** in the case of ***pass-by-value*** of the objects as arguments in the function call.

Example: 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 f with return value" << endl;
  Contor o2 = f(o1);
```

```
    cout << "After f" << endl;
    cout << "Before f without return value" << endl;
    f(o1);
    cout << "After f 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 f with return value
Before f 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 f without return value
Destructor : contor 1 object
Destructor : contor 0 object
```

Remarks:

1. The ***initialization*** of the parameter of the function *f* is made by the ***copy-constructor***. The parameter *x* becomes a ***temporary object*** which is local into the function *f*, and it will be ***destroyed*** when *f* ***finishes*** and it returns to the function *main*.
2. When the expression from the statement **`return`** is evaluated, the ***second temporary object*** is generated by using also the *copy-constructor*.
3. In the case when the ***function returns a value***, this object ***is not destroyed***, because it represents the value of the variable *o2* from the function main.
4. In the case when the ***function does not return a value***, this ***object is destroyed after the finishing of the function*** *f*, and before the returning to the function *main* (at the second call of *f*, two destructors acre 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 ***explicitly 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 called by default).
2. If at ***the end of the program execution*** there are objects allocated in the ***heap*** zone, the compiler ***forces*** the destructor call for these objects after the exit from the **`main`** function.

Example: 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;
  // ...
}
```

- In ***Python***, ***destructors*** are needed much less than in C++
  - Python has a *garbage collector* that handles memory management
- However, memory is not the only *resource* used by class instances:
  - There are also *sockets*, *database connections*, *files*, *buffers*, etc.
  - These resources need to be *released* when an object is destructed
- In ***Python***, the special function **`__del__`** can be overloaded for each class
  - It has the same meaning as a *destructor* from C++
  - There is a single **`__del__`** function for each class
  - If a class does not contain a **`__del__`** function, it is inherited from the root **`object`** class
  - The goal of the **`__del__`** function is to *release* resources used by an object (other than memory allocation)
  - The function **`__del__`** is called when the counter of the references to an object becomes zero

- A simple example:

```
class C(object):
    def __init__(self, x_):
        self.x = x_
        print (self.x, 'born')
    def __del__(self):
        print (self.x, 'died')
ob = C(5)
```

prints:

```
5 born
5 died
```

- However, there is a problem with the *garbage-collector*, called *circular references*:
    - Python does not know the *order* in which to destroy objects that hold circular references to each other
    - As a consequence, it *does not call* the destructors for such methods

- An example of circular references:

```
class A:
   def __init__(self, x_, b_):
      self.x = x_
      self.b = b_
      print('A', self.x, 'born')
   def __del__(self):
      print ('A', self.x, 'died')
class B:
   def __init__(self, y_):
      self.y = y_
      self.a = A(y_, self)
      print('B', self.y, 'born')
   def __del__(self):
      print('B', self.y, 'died')
ob = B(5)
```

prints:

```
A 5 born
B 5 born
```

- Between the objects of the classes **`A`** and **`B`** there are *circular references*
  - Destructors **`__del__`** are not called
- ***Python*** provides the **`weakref`** module that can solve this problem: ***week references***
- From the **`weakref`** documentation:
  - A *weak reference* to an object *is not enough to keep the object alive*:
    - when the only *remaining references* to a referent are *weak references*
      - garbage collection *is free to destroy the referent* and *reuse its memory* for something else
- The previous example written using week references:

```
import weakref
class A:
   def __init__(self, x_, b_):
      self.x = x_
      self.b = weakref.ref(b_)
      print('A', self.x, 'born')
   def __del__(self):
      print ('A', self.x, 'died')
class B:
   def __init__(self, y_):
      self.y = y_
      self.a = A(y_, self)
      print('B', self.y, 'born')
   def __del__(self):
      print('B', self.y, 'died')
ob = B(5)
```

The above sequence will print:

```
A 5 born
B 5 born
B 5 died
A 5 died
```

- Some examples related to the reference counter:

```
class C:
   pass
a = C()           # Creates an object (refcount = 1)
b = a             # Increases refcount on the object (2)
c = []
c.append(b)       # Increases refcount on the object (3)
del a             # Decrease refcount on the object (2)
b = 7             # Decrease refcount on the object (1)
c[0] = 5          # Decrease refcount on the object (0)
                  # Object will be destroyed
```

C. Modern features in C++ related to constructors and destructors

C1. Non static data members initialization

- Starting to ***C++11***, ***non static data members*** of a class ***can be initialized inside of the class***, as in the case of **`static const`** members
- In this case, the ***constructors*** of the class can:
  - ***Inherit*** the initialized values, or
  - ***Override*** these values
- Advantages:
  - Easier to write
  - Can perform a uniform initialization of objects
  - Are useful when a class has several constructors

Example:

```
class X {
   int min{5};
   int max{10};
public:
   X(int a, int b) : min(a), max(b) {}
   X() {}
   void print() const {cout << min << " " << max << endl;}
};
int main() {
   X x1, x2(4, 9);
   x1.print();      // 5 10
   x2.print();      // 4 9
   return 0;
}
```

C2. Move semantics and rvalue reference

- In the traditional C++, a ***lvalue reference*** is bind to another ***lvalue***:

```
int n = 5;          // OK: initialzation
int &m = n;         // OK: binding an lvalue reference
// ERROR! An lvalue reference cannot be bound to a rvalue
int &k = 10;
```

- However, a ***const lvalue reference*** can be bound to a ***rvalue***:

```
const int &k = 10;     // OK
```

- ***C++11*** introduces ***rvalue references*** which bind only to rvalues:

```
int&& v = 99;       // OK: v is a rvalue reference
```

- If **`T`** is a type, **`T&&`** represents the ***rvalue references*** to the values of **`T`**

- **Example** of two overladed functions:

```
void print (int& n) { cout << n << endl; }
void print (int&& n) { cout << n << endl; }
int value () {
   int tmp = 77;
   return tmp;
}
int main() {
   int i = 7;
   f(i);          // 7: lvalue reference is called
   f(value());    // 77: rvalue reference is called
   return 0;
}
```

- However, the standard library contains a function, **`move()`**, which takes an ***lvalue*** and converts it into an ***rvalue***:

```
f(move(i));   // OK: rvalue reference is called
```

- **Remark.** **`T&&`** represents in fact ***temporary objects*** that are permitted to be ***modified after they are initialized***:
  - The ***rvalue reference*** allows bind a ***mutable reference*** to an ***rvalue***, but not an ***lvalue***
    - ***rvalue references*** can ***detect*** if a value is a ***temporary*** object or ***not***
- The above remark represents the ***main concept*** of ***move semantics***
- In the classical C++, in a program, a lot of deep object copies can be created when objects are passed by value
  - This ***degradation of performance*** can be ***avoided*** by using a ***rvalue reference***
- The main usage of ***rvalue references*** is to create ***move constructors*** and ***move assignment operators***
- A ***move constructor*** is similar to a ***copy constructor***:
  - It takes an instance of an object as its argument and creates a new instance from original object.

- However, the move constructor will ***avoid memory reallocation*** because it knows that a ***temporary object is provided***:
    - Instead of ***copy*** the fields of the ***original object***, it will ***move*** them to the ***new instance***
    - The ***rvalue references*** and ***move semantics*** allow to avoid ***unnecessary copies*** when working with ***temporary objects***
- **Example**:

```
#include <iostream>
#include <algorithm>
#include <vector>

class A {
   int len;
   int* data;
public:
   A(int l) : len(l), data(new int[l]) {
        cout << "A: length = " << len << endl;
   }
```

```
    ~A() {
        cout << "~A(): length = " << len << endl;
        if (data != nullptr) {
            cout << " Deleting resource\n";
            delete[] data;
        }
    }
    // Copy constructor.
    A(const A& o) : len(o.len), data(new int[o.len]) {
        cout << "A(const A&): length = " << o.len << endl;
        copy(o.data, o.data + len, data);
    }
    // Move constructor.
    A(A&& o) : data(nullptr), len(0) {
        cout << "A(A&&): length = " << o.len << endl;
        data = o.data;
        len = o.len;
        // Release the data pointer from the source object
        // the destructor does not free the memory multiple times
        other.data = nullptr;
        other.len = 0;
    }
};
```

```
int main() {
    vector<A> v;
    v.push_back(A(25));   // move constructor
    A a(55);              // conversion constructor
    A b = a;              // copy constructor
    return 0;
}
```

- Since C++11, STL functions such as **`push_back()`** now define ***two overloaded versions***: one that takes const **`T&`** for ***lvalue arguments*** as before, and a new one that takes a parameter of type **`T&&`** for ***rvalue arguments***

- The ***move constructor***:
  - does ***not allocate any new resources***
  - the ***content is moved not copied***

- The ***move constructor*** is much ***faster*** than a ***copy constructor*** because it does ***not allocate memory*** nor does it ***copy memory blocks***

- **Remark**: as a result of ***moving resources*** from the ***initial object*** to the ***new object***, the ***initial object will disappear***

C3. Explicitly defaulted and deleted functions

- In C++, the compiler ***automatically generates*** the ***default constructor***, ***copy constructor***, ***copy-assignment operator***, and ***destructor*** for a user-defined class if they ***are not explicitly declared***
- However, ***not*** all these special functions are ***all time automaticaly generated***:
  - If ***any constructor is explicitly declared***, then ***no default constructor is automatically generated***
  - If a ***move constructor or move-assignment operator is explicitly declared***, then:
    - ***No copy constructor is automatically generated***
    - ***No copy-assignment operator is automatically generated***
  - If a ***copy constructor, copy-assignment operator, move constructor, move-assignment operator, or destructor is explicitly declared***, then:
    - ***No move constructor is automatically generated***
    - ***No move-assignment operator is automatically generated***
  - If a ***virtual destructor is explicitly*** declared, then ***no default destructor is automatically generated***

- As a consequence, if these special functions are not properly declared, objects from a class hierachy cannot be properly constructed
- For example:
  - If a ***base*** class **`A`** does not have a ***public or protected default constructor***, then a class **`B`** ***derived*** from **`A`** ***cannot automatically generate its own default constructor***
- In ***C++11***, ***explicitly defaulted functions*** ***make the compiler to generate*** these special functions, ***even if the above rules are accomplished***
- The ***syntax*** for a defaulted special function has only the declarator of the function followed by the construction **`=default`**

Example.

```
class A {
   int n;
public:
   A(int a): n(a) {
      cout << "A: Conversion constructor\n";
   }
```

```
    // the compiler will create the default constructor
    A() = default;
    void print() { cout << "n = " << n << endl; }
};
class B {
    int m;
public:
    B(int a): m(a) {
        cout << "B: Conversion constructor\n";
    }
    // user-defined default constructor
    B() {}
    void print() { cout << "m = " << m << endl; }
};
int main() {
    // call the defaulted constructor
    A* p = new A();
    p->print();   // n = 0
    // call the conversion constructor
    A a(1);
```

```
    a.print();    // n = 1
    // call the user-defined default constructor
    B* q = new B();
    q->print();   // m = -84631749
    // call the conversion constructor
    B b(2);
    b.print();    // m = 2
    return 0;
}
```

- **Remark**. In the case of ***non-user-defined default constructor***, a ***special kind of initialization*** will take place, and for ***built-in types*** this will result in ***zero-initialization***

- Except to the case when a ***virtual destructor*** is defined in a class (and a ***default destructor will be not created*** by the compiler), another case when a ***defaulted destructor*** is useful is related to the ***move semantics***

Example

```
class X {
public:
    ~X() { /* do something */ }
    // ...
};
```

- The above class will ***loose its move operations***, because the ***move constructor*** and ***the move asignment operator*** will ***not be generated*** by the compiler
  - The code will continue to compile, but will ***silently*** it will call ***copy operations*** instead of ***move operations***

- In order to ***not inhibit*** the generation of ***default move constructors and move asignment operators***, a ***defaulted destructor*** can be used:

```
class X {
public:
    ~X() = default
    // ...
};
```

- ***C++11*** introduced another use of the operator **`delete`**:
  - To ***disable*** the ***usage of a function***
    - This is done by appending the **`=delete;`** specifier to the end of the function declaration

- ***Special functions***, as well as as ***normal member functions*** and ***non-member functions*** can be ***deleted*** to ***prevent them from being defined or called***

- ***Deleting*** of ***special member functions*** provides a ***cleaner way*** of preventing the compiler ***to not generate these special member functions*** if not desired

Example:

```
class X {
   int n;
public:
   A(int k): n(k) {}
   // Delete (disable) the copy constructor
   A(const A&) = delete;
};
```

```
int main() {
   A a1(1);   // OK
   // Error! The usage of the
   // copy constructor is disabled
   A a2 = a1;
   return 0;
}
```

- **Remark**. A ***deleted special member function*** is implicitly ***inline***
- ***Deleting*** of ***normal member function*** or ***non-member functions prevents problematic type promotions*** from causing an ***unintended function to be called***
- **Remark**. **`=delete`** is a ***function definition*** (it does ***not remove or hide the declaration***)
  - As a consequence, ***deleted functions*** still ***participate in overload resolution*** any other function
  - Attempts to use a ***deleted function*** result is a ***compile time error***

Example. ***Deleted a overloaded function prevents its call*** through ***type promotion*** of ***int*** to ***double***

```
#include <cmath>
#inclide <iostream>
using namespace std;
void f(int) =delete;
void f(double x) {
   cout << sqrt(x) << endl;
}
int main() {
   f(4);   // compiler error
   f(4.0); // OK
}
```

- However, if we add the following code, the result is OK because of ***promotion*** from ***float*** to ***double***:

```
float x = 4.0;
f(x);   // OK
```

- To ensure that ***no promotion will be performed***, one can define a ***template function*** that is ***deleted***:

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

template <typename T>
void f(T) =delete;

void f(double x) {
   cout << sqrt(x) << endl;
}

int main() {
   int n = 4;
   float x = 4.0;
   double y = 4.0;
   f(n);    // compiler error
   f(x);    // compiler error
   f(y);    // OK
}
```

C4. Overlapping and delegating constructors

- Because a class can have several constructor, in many cases some constructors have overlapping functionality

Example.

```
class X {
   int n, m, p;
public:
   X() {
      n = m = 0;    // redundant actions
      p = 0;
   }
   X(int a) {
      n = m = 0;    // redundant actions
      p = 10;
   }
   // ...
};
```

- A variant to reduce the redundant actions is to write a distinct initialization function and to call it in the constructors.

```
class X {
   int n, m, p;
public:
   void init() {
      n = m = 0;
   }
   X() {
      init();
      p = 0;
   }
   X(int a) {
      init();
      p = 10;
   }
   // ...

};
```

- With the ***delegating constructors*** feature, ***common initializations*** can be concentared in one constructor named ***target constructor***
- ***Delegating constructors*** can call the ***target constructor*** to do the ***initialization***

Example:

```
class X {
    int n, m, p;
public:
    // target constructor
    X() {
        n = m = 0;
        p = 0;
    }
    // delegating constructor
    X(int a) : X() {
        p = 10;
    }
    // ...
};
```

- A ***delegating constructor*** can also be used as the ***target constructor*** of ***one or more delegating constructors***
  - This feature can be used to make programs more readable and maintainable

```
class Complex {
    int re, im;
public:
   Complex() : Complex(0) {}
    Complex (int x) : Complex (x, 0) {}
    Complex (int x, int y) : re(x), im(y) {}
};
```

C5. `constexpr` constructors

- Constructors can also be qualified as ***constexpr*** to indicate that ***object construction*** can be performed at ***compile time***, provided that ***all arguments*** to constructor are ***constant expressions***
- In addition, ***constexpr constructors*** are implicitly ***inline***
- If an ***object*** of a class has to be constructed at ***compile time***, its ***constructors*** have to be ***constexpr functions*** (and eventually its member functions)

Example

```
class Circle {
    int x, y, r;
    public:
    constexpr Circle (int a, int b, int c) :
            x(a), y(b), r(c) {}
    constexpr double getArea () {
            return r * r * 3.1415926;
    }
};
```

```
int () {
   constexpr Circle c(0, 0, 10);
   constexpr double area = c.getArea();
   cout << area << endl;
   return 0;
}
```