Constructors and destructors – details

Let's dive into the details...

Reminder

- Constructor is a special method which name is like the class. In a constructor we can write code which is responsible for initializing fields. When we declare an object then a constructor is called after its allocation.
- Constructor cannot add new fields
- Constructors and destructor are methods with the special feature that they are called automatically
- A constructor can be overloaded – this is a very common practice
- A constructor cannot return any value
- A constructor can be called for an object declared as **`const`** or **`volatile`** but itself cannot be defined as **`const`** and **`volatile`**
- Constructor cannot be defined as **`static`** – it has to work on non-static fields

How and when?

Similarly, to built-in, user defined types can be defined in several ways – this depends on how and where we declare them.

Local auto objects

Objects created on a stack during the run time – they are created when a program encounters their declaration and their lifespan is bounded by a block in which they are declared.

```
{   // beginning of the block

        device M; // object declaration
        // auto device M; - compiler puts auto for us so no need to
    add it

}   // end of the block
//... we have no more access to M
```

Such define objects are of type local auto.

How and when?

Local static objects

If we declare an object in a block with the word **`static`** then this object is defined as a local static. The lifespan of such an object is from the execution of a program to its termination but we can access it only from the block in which we declared it. In this case a constructor of a local static object is called before the **`main`** function. This gives us an opportunity to run our code before **`main`**.

How and when?

Global scope objects

If an object is defined outside any block and without the word **`static`** then it is of the global scope – we can use it in different compilation units while using the protected word **`extern`**.

```
device G;
main()
{ }
```

File scope objects

They are declared similarly to the global scope objects with the difference that their declarations contains the word **`static.`** Such defined objects can be accessed from any place of a compilation unit.

The lifespan of global and file scope objects: The whole run time of a program. Moreover, constructors are called before the **`main`** function.

How and when?

Objects created by the new operator

In C++ we can declare objects with the **new** operator. It is a dynamical allocation (in C we used **malloc** like functions to dynamically allocate memory).

```
void f()
{
    device *needle;  // first we declare a pointer
    needle = new device(1,1, "Weight", "kg");
//
}
```

The lifespan of dynamically allocated objects: from the place where we allocated it until a call of the **delete** operator.

```
delete needle;
```

We cannot use the C function **free** for memory allocated with **new**!

Explicit constructor calls

We can explicitly call a constructor as a consequence we will obtain an object which lifespan is bounded by the call

```
class_name(arguments)
```

Note that when we call a constructor then we do not use the below notation:

```
object.method(arguments)
```

Constructor is a method but a special one and its role is to create an object so above notation has no sense.

Destructors

Destructor itself does not release memory of an object. There is no obligation to define a destructor. We need a destructor when we need to “clean” while removing an object from memory. For example, if we allocated memory during the lifespan of an object we can clean it in a destructor by we have to do this explicitly.

Destructor is called automatically when an object reached the end of its lifespan.

Note that a destructor is not called when a pointer or a reference reached the end of its lifespan.

We can call from destructors different methods.

Note that a destructor cannot be declared with words **`const`** or **`volatile`**, but it can work on objects defined as **`const`** or **`volatile`**.

Explicit declarations of destructors

We can call a destructor explicitly.

```
object->~class_name();
```

```
object.~class_name();
```

Note that if we want to call a destroctor from another method then we cannot call it:

```
~class_name();
```

We need to explicitly write:

```
this->~class_name();
```

Call of non-existing destructor

If we call the destructor of a class which has no destructor then this call will be ignored. For example:

```
class C
{
  int a;
  public:
  void test() {};
}

C a;

a.~C();
```

Default constructor

The default constructor is a constructor which signature is void.

```
class Boss {
public:
    Boss(int);
    Boss (void); //<-- default
    Boss(float*);
};
class House {
public:
    House(int);
    House(float)     ;
    House(char *s = NULL, int a = 4, float pp = 6.66); //<-- default
};
```

Note that the last constructor can be called without any argument.

If we do not declare any constructor then the defulat one will be generated automatically in the `public` section.

Constructors with classes in their signatures

We can have a constructor which takes as its arguments objects of different classes. In such situations first constructors of these objects are called and then a constructor of the main object can be called. When we build a radio we have to build first transistors, etc and then we can think about a case, etc.

Copy constructor

We will call a constructor defined as

```
class_name::class_name( <const | volatile> class_name &)
```

a copy constructor. The argument has to be a reference.

With a copy constructor we create a copy of another object – we use this another object as an template.

This is also a copy constructr:

```
K::K(K&, float = 51.45,int* = NULL)
```

If we will not define any copy constructor then it will be generated automatically.

Explicit calls of copy constructors

For example:

```
K obj_template;  // we define our template
//...
K obj_new = K(obj_template);
K obj_new = obj_template;      //almost explicit calls
```

Implicit calls of copy constructors

a) **Passing arguments by value** – this is like for built-in types. When a function takes as its argument an object then this object is copied to it.

b) Returning an object as results of functions. As above.

Why copy construct uses references?

When we use references then we do not work on a copy. We define a copy constructor to control the process of copying. We know that passing an object by value involves copying – eg, calls to copy constructor. Therefore, we need to avoid passing by value and work directly on a given object passed into the copy constructor.

Automatically generated copy constructor

If we do not define the copy constructor in a class then compiler will generate it.

In automatically generated copy constructor copying is done field-by-field.

If we need only identical copies then automatically generated copy constructor is enough.