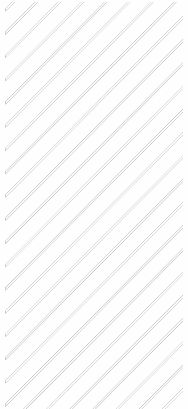
Lecture 10

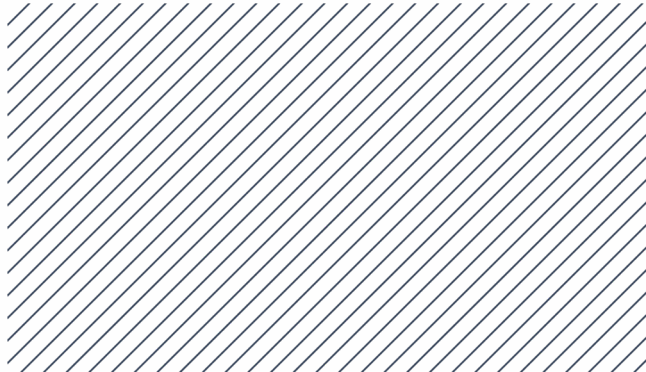
Classes and structures, pt.2

Konstantin Leladze

C++ Basics

DIHT MIPT 2021





Constructor

Constructor is a special non-static member function (aka method) of a class that is used to initialize objects (aka instances) of its class type.

Defining custom constructor:

Defining your own constructor for a class is very similar to defining a method, however, instead of the method name, we write the name of the class itself, and do not specify the return value (constructors do not return anything!)

As you can see, in the main function, we passed values to the constructor, and now an object named instance has 23 and -42 in the _x and _y fields, respectively.

```
struct coordinates {
    coordinates (int x, int y) {
        _x = x;
        _y = y;
    }

private:
    int _x;
    int _y;
};


int main () {
    coordinates instance(23, -42);
}
```

Default constructor

A default constructor is a constructor which can be called with no arguments (either defined with an empty parameter list, or with default arguments provided for every parameter). A type with a public default constructor is called **DefaultConstructible**.

You should not write parentheses when creating an instance to call the default constructor.

A constructor, all arguments of which **have default values**, is also a default constructor (because formally, it can be called without passing values from the outside!):

```
/// Also a default constructor
coordinates (int x = 0, int y = 0) {
    _x = x;
    _y = y;
}
```

```
struct coordinates {
    /// Default constructor
    coordinates () {
        _x = 0;
        _y = 0;
    }

    /// Custom non-default constructor
    coordinates (int x, int y) {
        _x = x;
        _y = y;
    }

private:
    int _x;
    int _y;
};


int main () {
    coordinates default_constructed_instance;
    coordinates non_default_constructed_instance(23, -42);
}
```

Keyword this

The keyword this stores an address of the object (aka instance) on which the non-static member function is being called.

```
struct awesome_dude {
    void say_hi () {
        std::cout << "hi from " << this << std::endl;
    }
};


int main () {
    awesome_dude first_dude;
    awesome_dude second_dude;
    first_dude.say_hi();
    second_dude.say_hi();
}
```

```
hi from 0x7ffeeb657ac8
hi from 0x7ffeeb657ac0
```

Another example with keyword this: class logger

The this keyword is also available in the constructor, so we can write something like this:

```
struct logger {
   logger () {
       std::cout << "constructor of logger called at " << this << std::endl;
   }
};


int main () {
    logger first_logger;
    logger second_logger;
}
```

```
constructor of logger called at 0x7ffee64dbac8
constructor of logger called at 0x7ffee64dbac0
```

Instance as a field

A class or structure can contain an instance as a field inside itself. But in this case, if the instance does not have a default constructor, then it is not clear how to call the constructor of this instance.

```
struct logger {
    logger (int number) {
        std::cout << "Logger #" << number << " has been constructed at" << this << std::endl;
    }
};

struct awesome_struct {
    logger the_logger;
    awesome_struct () {}
};

int main () {
    awesome_struct a;
}
```

Constructor for 'awesome_struct' must explicitly initialize the member 'the_logger' which does not have a default constructor

Destructor

A destructor is a special member function that is called when the lifetime of an object ends. The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.

```
struct logger {
    logger (int number) {
        _number = number;
        std::cout << "Logger #" << number << " has been constructed at" << this << std::endl;
    }

    ~logger () {
        std::cout << "Logger #" << _number << " has been destructed at" << this << std::endl;
    }

private:
    int _number;
};

int main () {
    logger a(0);
}
```

```
Logger #0 has been constructed at0x7ffee063bac8
Logger #0 has been destructed at0x7ffee063bac8
```

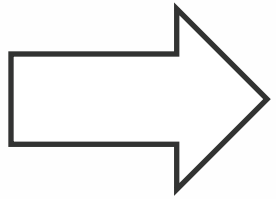
Member initializer lists

Member initializer list allows you to initialize members in the order of their declarations in the class **before** executing the constructor body. Member initializer list is part of the **definition** (not declaration).

```
struct coordinates {
    coordinates (int x, int y) {
        _x = x;
        _y = y;
    }

private:
    int _x;
    int _y;
};


int main () {
    coordinates instance(23, -42);
}
```



```
struct coordinates {
    coordinates (const int x, const int y): _x(x), _y(y) {}

private:
    int _x;
    int _y;
};
```

In addition, member initializer lists help us solve the problem with member-instance initialization.
An example of a solution to the problem is shown on the next slide.

Member initializer lists: order of initialization

The order of member initializers in the list is irrelevant: the actual order of initialization is as follows:
1) Non-static data member are initialized in order of declaration in the class definition.
2) The body of the constructor is executed

```
struct logger {
    logger (int number) {
        _number = number;
        std::cout << "Logger #" << number << " has been constructed at" << this << std::endl;
    }

    ~logger () {
        std::cout << "Logger #" << _number << " has been destructed at" << this << std::endl;
    }

private:
    int _number;
};

struct awesome_struct {
    logger a;
    logger b;
    awesome_struct (): a(0), b(1) { std::cout << "Awesome struct has been constructed at " << this << std::endl; }
};

int main () {
    awesome_struct instance;
}
```

```
Logger #0 has been constructed at0x7ffee78a6ac8
Logger #1 has been constructed at0x7ffee78a6acc
Awesome struct has been constructed at 0x7ffee78a6ac8
Logger #1 has been destructed at0x7ffee78a6acc
Logger #0 has been destructed at0x7ffee78a6ac8
```

Defaulted default constructor and destructor

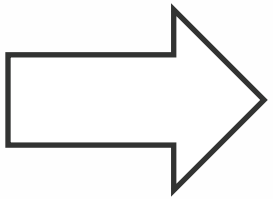
Explicitly defaulted function declaration is a new form of function declaration that is introduced into the C++11 standard. You can append the = default; specifier to the end of a function declaration to declare that function as an explicitly defaulted function. The compiler generates the default implementations for explicitly defaulted functions, which are more efficient than manually programmed function implementations. A function that is explicitly defaulted must be a special member function and has no default arguments. Explicitly defaulted functions can save your effort of defining those functions manually.

```
struct awesome_struct {
    awesome_struct () {

    }

    ~awesome_struct () {

    }
};
```



```
struct awesome_struct {
    awesome_struct () = default;
    ~awesome_struct () = default;
};
```

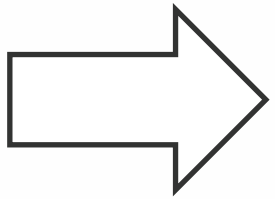
Defaulted default constructor and destructor

In the last lecture, I said that in some cases the compiler will generate default constructos on its own. In fact, this does not always happen. For example, if you declare a custom constructor for your class, the default constructor will no longer be generated for it. We'll talk about this in detail on the next slides, but now let's see an example of the code.

```
struct coordinates {
    coordinates (const int x, const int y): _x(x), _y(y) {}

private:
    int _x;
    int _y;
};

int main () {
    coordinates instance;
}
```



```
struct coordinates {
    coordinates () = default;
    coordinates (const int x, const int y): _x(x), _y(y) {}

private:
    int _x;
    int _y;
};

int main () {
    coordinates instance;
}
```

This is how we can train the C ++ compiler to generate the defaulted default constructor even if it was not automatically generated.

Deleted default constructor and destructor

The default constructor and destructor can also be removed. To do this, use the syntax = delete;
This can be useful in some exotic cases, such as when you want to create a class that should not be instantiated.

```
struct exotic {
    exotic () = delete;
    ~exotic () = delete;
};

int main () {
    exotic instance; /// CE
}
```

Implicitly declared and defined default constructor

Implicitly declared default constructor:
If no user-declared constructors of any kind are provided for a class type (struct or a class), the compiler will always declare a default constructor as a public member of its class.

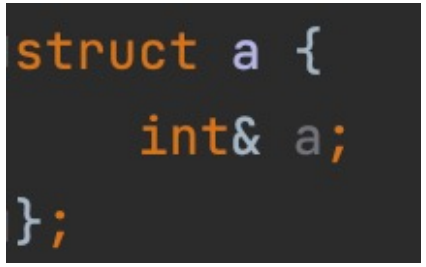
Implicitly defined default constructor:
If the implicitly-declared default constructor is not defined as deleted, it is defined (that is, a function body is generated and compiled) by the compiler if, and it has the same effect as a user-defined constructor with empty body and empty initializer list. That is, it calls the default constructors of the non-static members of this class.

The implicitly-declared or defaulted default constructor for class **T** is defined as **deleted** if any of the following is true:

- **T** has a member of reference type without a default initializer.
- **T** has a non-const-default-constructible const-member without a default member initializer.
- **T** has a member which has a deleted default constructor, or its default constructor is ambiguous or inaccessible from this constructor.

Implicitly declared and defined default constructor

- **T** has a member of reference type without a default initializer.



```
struct a {
    int& a;
};
```

- **T** has a non-const-default-constructible const-member without a default member initializer.

```
struct a {
    const int k;
};
```

- **T** has a member which has a deleted default constructor, or its default constructor is ambiguous or inaccessible from this constructor.

```
struct a {
    a () = delete;
};

struct b {
    a instance;
};
```

Member initializer lists again

Let’s solve the problem with initializing const-members and reference members using member initializer lists

```
struct logger {
    const int const_member;
    int& reference_member;
    int member;

    logger (): const_member(3321), reference_member(member), member(44) {}
}
```

Implicitly declared and defined destructor

Implicitly declared destructor:
If no user-declared prospective destructor is provided for a class type (struct, class), the compiler will always declare a destructor as an inline public member of its class.

Deleted implicitly declared destructor:
The implicitly-declared or defaulted destructor for class T is undefined defined as deleted if any of the following is true:

- T has a non-static data member that cannot be destructed (has deleted or inaccessible destructor)

```
struct a {
    ~a () = delete;
};

struct b {
    a instance_a;
};

int main () {
    b instance_b;
}
```

```
Call to implicitly-deleted default constructor of 'b'
default constructor of 'b' is implicitly deleted because field 'instance_a' has a deleted
destructor
```

Implicitly defined destructor:
If an implicitly-declared destructor is not deleted, it is implicitly defined (that is, a function body is generated and compiled) by the compiler. This implicitly-defined destructor has an empty body.

Delegating constructors

Many classes have multiple constructors that do similar things—for example, validate parameters:

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c() {}
    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};
```

Delegating constructors

You could reduce the repetitive code by adding a function that does all of the validation, but the code for class_c would be easier to understand and maintain if one constructor could delegate some of the work to another one.

To add delegating constructors, use the constructor (. . .) : constructor (. . .) syntax:

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) : class_c(my_max) {
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) : class_c (my_max, my_min){
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};

int main() {
    class_c c1{ 1, 3, 2 };
}
```

- Notice that the constructor class_c(int, int, int) first calls the constructor class_c(int, int), which in turn calls class_c(int).
- Each of the constructors performs only the work that is not performed by the other constructors.
- The first constructor that's called initializes the object so that all of its members are initialized at that point.
- You can’t do member initialization in a constructor that delegates to another constructor

Explicit keyword

Let's take another look at the logger class:

```
struct logger {
    logger (const int number): _number(number) {
        std::cout << "logger " << number << " constructed at " << this << std::endl;
    }

    ~logger () {
        std::cout << "logger " << _number << " destructed at " << this << std::endl;
    }

private:
    int _number;
};
```

As you can see, it has a constructor from one int argument.
The following code is allowed to compile, which is a problem (it can be confusing in some cases!)

```
int main () {
    logger instance = 3;
}
```

Explicit keyword

But if we declare a constructor as explicit, it won't work anymore.

```
struct logger {
    explicit logger (const int number): _number(number) {
        std::cout << "logger " << number << " constructed at " << this << std::endl;
    }

    ~logger () {
        std::cout << "logger " << _number << " destructed at " << this << std::endl;
    }

private:
    int _number;
};

int main () {
    logger instance = 3;
}
```

Thus, the explicit construct cannot be used for implicit conversions.

Copy constructor

A copy constructor of class **T** is a non-template constructor whose first parameter is **T&**, **const T&**, and either there are no other parameters, or the rest of the parameters all have default values.

The purpose of the copy constructor is to construct a new object based on the passed one. At its core, copy-constructor should not modify the object passed to it (although the C++ language standard does not prohibit it). Therefore, we will not use the version in which we receive the object through a non-const reference.

Two options remain: **T** and **const T&**. But what's the difference? In the second case, no extra copy is created!

```
struct coordinates {
    coordinates (const coordinates& other): _x(other._x), _y(other._y) {}

private:
    int _x;
    int _y;
};
```

Implicitly declared and defined copy constructor

Implicitly declared copy constructor:
If no user-defined copy constructors are provided for a class type (struct, class, or union), the compiler will always declare a copy constructor as a non-explicit public member of its class.

Deleted implicitly declared default constructor:
The implicitly-declared or defaulted copy constructor for class T is defined as *deleted* if any of the following conditions are true:

- **T** has non-static data members that cannot be copied (have deleted, inaccessible, or ambiguous copy constructors);

Implicitly defined copy constructor
If the implicitly-declared copy constructor is not deleted, it is defined (that is, a function body is generated and compiled) by the compiler. The constructor performs full member-wise copy of non-static members, in their initialization order.

```
struct coordinates {
    coordinates (const coordinates& other) = default;

private:
    int _x;
    int _y;
};
```

Rule of three

As you can see, there are a lot of rules for implicit generation of default constructor, copy constructor and destructor (and I didn't even show you half of that!). Therefore, I propose to remember the following rule, following which you will never make the mistake when thinking about whether to define a certain constructor or destructor for you or not.

> **Rule of 3**: If a class requires a user-defined **destructor**, a user-defined **copy constructor**, or a user-defined **copy assignment operator**, it almost certainly requires all three.

We haven't discussed the copy assignment operator yet, but don't worry, we will discuss it in the next lecture, in which we will talk about redefining operators in classes.

RAII

Resource acquisition is initialization (**RAII**) is a programming idiom used in several object-oriented, statically-typed programming languages to describe a particular language behaviour. In RAII, holding a resource is a class invariant, and is tied to object lifetime: resource allocation (or acquisition) is done during object creation (specifically initialization), by the constructor, while resource deallocation (release) is done during object destruction (specifically finalization), by the destructor. In other words, resource acquisition must succeed for initialization to succeed. Thus the resource is guaranteed to be held between when initialization finishes and finalization starts (holding the resources is a class invariant), and to be held only when the object is alive. Thus if there are no object leaks, there are no resource leaks.

RAII is associated most prominently with C++ where it originated, but also D, Ada, Vala, and Rust. Other names for this idiom include *Constructor Acquires, Destructor Releases* (CADRe). RAII ties resources to object *lifetime,* which may not coincide with entry and exit of a scope.

RAII, example

Let's create a polygon class that will take coordinates in the form of two arrays x and y, and store them inside itself

```
class polygon {
public:
    polygon (int n, const double* const x, const double* const y) {
        _x = new double[n];
        _y = new double[n];

        for (int i = 0; i < n; i ++) {
            _x[i] = x[i];
            _y[i] = y[i];
        }
    }

    ~polygon () {
        delete[] _x;
        delete[] _y;
    }

private:
    double* _x;
    double* _y;
};
```

RAII, example

Let's add a couple of useful methods

```
int vertices_count () const { return _n; }

double perimeter () const {
    double p = 0.;

    for (int i = 0; i < _n; i ++) {
        int j = (i + 1) % _n;
        p += std::sqrt(std::pow(_x[j] - _x[i], 2.) + std::pow(_y[j] - _y[i], 2.));
    }

    return p;
}
```

RAII, example

Using defaulted copy constructor will not work in this case, because we need a "deep copy" of the object (instance).

But anyway, in order not to rewrite the main constructor code, let's use the delegate-constructor

```
polygon (const polygon& other): polygon(other._n, other._x, other._y) {}
```

Allocating class instances on heap

Class instances can be allocated to the heap, as is the case with primitive types such as int:

```
struct coordinates {
    int x;
    int y;

    coordinates (const int x, const int y): x(x), y(y) {}
};

int main () {
    coordinates* instance = new coordinates(3.421, 1.4442);
}
```

But in this case, if we refer to a pointer on an instance, we have to use the **operator ->** instead of the operator .

```
int main () {
    coordinates* instance = new coordinates(3.421, 1.4442);
    std::cout << instance->x << " " << instance->y << std::endl;
}
```

Allocating class instances on heap

Let’s rewrite our entire polygon class so that it works with the coordinates class.

Full code can be found here.

Lecture 10

Classes and structures, pt.2

Konstantin Leladze

C++ Basics

DIHT MIPT 2021