Introduction to C++ for C programmers

high-level meets low-level

Mikhail Katliar

Max Planck Institute for Biological Cybernetics
& Systems Control and Optimization Laboratory,
University of Freiburg

October 29, 2018

What is C++?

C++ is a general-purpose programming language with a bias towards systems programming that [**?**]

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming

Where did the name "C++" come from?

"The name C++ was coined by Rick Mascitti in the summer of 1983. The name signifies the evolutionary nature of the changes from C. [...] For yet another interpretation of the name C++, see the appendix of [Orwell,1949]." [**?**]

Is it true that . . . ?

- **C++ is low-level (?)** No. C++ offers both low-level and high-level features.

Is it true that . . . ?

- **C++ is low-level (?)** No. C++ offers both low-level and high-level features.
- **C++ is too slow for low-level work (?)** No. If you can afford to use C, you can afford to use C++, even the higher-level facilities of C++ where you need their functionality. See [**?**, **?**].

Is it true that . . . ?

- **C++ is low-level (?)** No. C++ offers both low-level and high-level features.
- **C++ is too slow for low-level work (?)** No. If you can afford to use C, you can afford to use C++, even the higher-level facilities of C++ where you need their functionality. See [**?**, **?**].
- **C++ is useful only if you write truly object-oriented code (?)** No. C++ provides support for a wide variety of needs, not just for one style or for one kind of application. In fact, compared to C, C++ provides more support for very simple programming tasks. [**?**]

Did he really say that?

B. Stroustrup:

- "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off".

Did he really say that?

B. Stroustrup:

- "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off".
- "There are only two kinds of languages: the ones people complain about and the ones nobody uses".

Did he really say that?

B. Stroustrup:

- "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off".
- "There are only two kinds of languages: the ones people complain about and the ones nobody uses".
- "C++ Is my favorite garbage collected language because it generates so little garbage".

Did he really say that?

B. Stroustrup:

- "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off".
- "There are only two kinds of languages: the ones people complain about and the ones nobody uses".
- "C++ Is my favorite garbage collected language because it generates so little garbage".
- "If you give people the choice of writing good code or fast code, there's something wrong. Good code should be fast".

Did he really say that?

B. Stroustrup:

- "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off".
- "There are only two kinds of languages: the ones people complain about and the ones nobody uses".
- "C++ Is my favorite garbage collected language because it generates so little garbage".
- "If you give people the choice of writing good code or fast code, there's something wrong. Good code should be fast".
- "I hate to choose between elegance and efficiency". [**?**]

What is the difference between C and C++?

- C++ is a direct descendant of C that retains almost all of C as a subset.

What is the difference between C and C++?

- C++ is a direct descendant of C that retains almost all of C as a subset.
- C++ provides stronger type checking than C and directly supports a wider range of programming styles than C.

What is the difference between C and C++?

- C++ is a direct descendant of C that retains almost all of C as a subset.
- C++ provides stronger type checking than C and directly supports a wider range of programming styles than C.
- C++ supports data abstraction, object-oriented programming, and generic programming.

What is the difference between C and C++?

- C++ is a direct descendant of C that retains almost all of C as a subset.
- C++ provides stronger type checking than C and directly supports a wider range of programming styles than C.
- C++ supports data abstraction, object-oriented programming, and generic programming.
- "I have never seen a program that could be expressed better in C than in C++ (and I don't think such a program could exist - every construct in C has an obvious C++ equivalent)." [**?**]

C++ language features

- Access modifiers + member functions = classes
- References
- Function overloading
- Constructors and deterministic destructors
- Nicer dynamic memory management
- Inheritance
- Virtual functions
- Run-time Type Information (RTTI)
- Exceptions
- Templates
- Standard library

C++ Standard Library

- I/O streams
- Containers
- Strings
- Smart pointers
- Type checking
- Threads
- Random numbers
- Time measurement
- Localization
- ...

What follows

In the rest of the talk, we consider 2 mechanisms which make life easier, and how they work in C++:

1. Encapsulation
2. Constructors, destructors and RAII

Example 1

Before we start talking about encapsulation, let's consider an example (Example 1).

Encapsulation

Definition

Encapsulation (aka *data abstraction*) means means separation of implementation details (internal representation of an object) from the interface (how an object is used).

Encapsulation

Definition

Encapsulation (aka *data abstraction*) means means separation of implementation details (internal representation of an object) from the interface (how an object is used).

- Encapsulation allows the internal representation of an object to be changed without affecting the user code – good!

Encapsulation

Definition

Encapsulation (aka *data abstraction*) means means separation of implementation details (internal representation of an object) from the interface (how an object is used).

- Encapsulation allows the internal representation of an object to be changed without affecting the user code – good!
- If the user code can access object's internal representation, it will break when the implementation details change – bad! This breaks incapsulation.

Encapsulation

Definition

Encapsulation (aka *data abstraction*) means means separation of implementation details (internal representation of an object) from the interface (how an object is used).

- Encapsulation allows the internal representation of an object to be changed without affecting the user code – good!
- If the user code can access object's internal representation, it will break when the implementation details change – bad! This breaks incapsulation.
- $\Rightarrow$ to achive proper encapsulation, internal representation should be inaccessible. But it should be accessible, in order to do anything useful.

Encapsulation

Definition

Encapsulation (aka *data abstraction*) means means separation of implementation details (internal representation of an object) from the interface (how an object is used).

- Encapsulation allows the internal representation of an object to be changed without affecting the user code – good!
- If the user code can access object's internal representation, it will break when the implementation details change – bad! This breaks incapsulation.
- $\Rightarrow$ to achive proper encapsulation, internal representation should be inaccessible. But it should be accessible, in order to do anything useful.
- How this contradiction can be resolved?

Member access

Access modifiers

- **private** members are accessible to the member functions of the class only.
- **public** members are accessible to everyone
- As a rule, all data should be **private**, otherwise encapsulation is broken.

Member access

Access modifiers

- **private** members are accessible to the member functions of the class only.
- **public** members are accessible to everyone
- As a rule, all data should be **private**, otherwise encapsulation is broken.

class vs **struct**

The only difference between **class** and **struct** is that

- Within **struct**, the default access is **public**.
- Within **class**, the default access is **private**.

Crtitical distinction

There is a critical distinction between accessing object's data directly vs via functions:

- Functions have separate interface (signature) and implementation.

Crtitical distinction

There is a critical distinction between accessing object's data directly vs via functions:

- Functions have separate interface (signature) and implementation.
- When directly accessing data members, interface $\equiv$ implementation.

Crtitical distinction

There is a critical distinction between accessing object's data directly vs via functions:

- Functions have separate interface (signature) and implementation.
- When directly accessing data members, interface $\equiv$ implementation.
- Directly manipulating data members can put and object into an inconsistent state!

Crtitical distinction

There is a critical distinction between accessing object's data directly vs via functions:

- Functions have separate interface (signature) and implementation.
- When directly accessing data members, interface $\equiv$ implementation.
- Directly manipulating data members can put and object into an inconsistent state!

Crtitical distinction

There is a critical distinction between accessing object's data directly vs via functions:

- Functions have separate interface (signature) and implementation.
- When directly accessing data members, interface $\equiv$ implementation.
- Directly manipulating data members can put and object into an inconsistent state!

$\Rightarrow$ data must always be **private**.

Example 2

Before going into ctror and dtor topic, let's go back to an example (Example 1).
How can we still mess up with this code?

Constructors and destructors

- **Constructor** is a function which is called at the beginning of object's lifecycle. It initializes the internal state and acquires necessary resources.

Constructors and destructors

- **Constructor** is a function which is called at the beginning of object's lifecycle. It initializes the internal state and acquires necessary resources.
- **Destructor** is a function which is called at the end of object's lifecycle. It releases the resources acquired in the constructor.

Constructors and destructors

- **Constructor** is a function which is called at the beginning of object's lifecycle. It initializes the internal state and acquires necessary resources.
- **Destructor** is a function which is called at the end of object's lifecycle. It releases the resources acquired in the constructor.
- For automatic variables, the ctor is called when the variable is **declared**, and dtor is called when it **goes out of scope** (automatically!).

Constructors and destructors

- **Constructor** is a function which is called at the beginning of object's lifecycle. It initializes the internal state and acquires necessary resources.
- **Destructor** is a function which is called at the end of object's lifecycle. It releases the resources acquired in the constructor.
- For automatic variables, the ctor is called when the variable is **declared**, and dtor is called when it **goes out of scope** (automatically!).
- For global static variables, the ctor is called **before entering** `main()` and the dtor is called **after exiting** `main()` (automatically!).

Constructors and destructors

- **Constructor** is a function which is called at the beginning of object's lifecycle. It initializes the internal state and acquires necessary resources.
- **Destructor** is a function which is called at the end of object's lifecycle. It releases the resources acquired in the constructor.
- For automatic variables, the ctor is called when the variable is **declared**, and dtor is called when it **goes out of scope** (automatically!).
- For global static variables, the ctor is called **before entering** `main()` and the dtor is called **after exiting** `main()` (automatically!).
- For automatic and global static variables, it is **guaranteed** that dtors are called in **reverse order** w.r.t. ctors.

Constructors and destructors

- **Constructor** is a function which is called at the beginning of object's lifecycle. It initializes the internal state and acquires necessary resources.
- **Destructor** is a function which is called at the end of object's lifecycle. It releases the resources acquired in the constructor.
- For automatic variables, the ctor is called when the variable is **declared**, and dtor is called when it **goes out of scope** (automatically!).
- For global static variables, the ctor is called **before entering** `main()` and the dtor is called **after exiting** `main()` (automatically!).
- For automatic and global static variables, it is **guaranteed** that dtors are called in **reverse order** w.r.t. ctors.
- It is **guaranteed** that the dtors for fully-constructed objects **will be called** regardless of the program flow.

RAII

Resources

. . . are anything that you first acquire and then release (memory, files, mutexes, device driver contexts, etc.)

RAII

Resources

. . . are anything that you first acquire and then release (memory, files, mutexes, device driver contexts, etc.)

RAII

- **Acquire** the resource in **ctor**.

RAII

Resources

. . . are anything that you first acquire and then release (memory, files, mutexes, device driver contexts, etc.)

RAII

- **Acquire** the resource in **ctor**.
- **Release** the resource in **dtor**.

RAII

Resources

. . . are anything that you first acquire and then release (memory, files, mutexes, device driver contexts, etc.)

RAII

- **Acquire** the resource in **ctor**.
- **Release** the resource in **dtor**.
- The compiler **guarantees** that dtor will be called $\Rightarrow$ the resource will always be **released** $\Rightarrow$ no resource leaks!

RAII

Resources

. . . are anything that you first acquire and then release (memory, files, mutexes, device driver contexts, etc.)

RAII

- **Acquire** the resource in **ctor**.
- **Release** the resource in **dtor**.
- The compiler **guarantees** that dtor will be called $\Rightarrow$ the resource will always be **released** $\Rightarrow$ no resource leaks!

RAII

Resources

. . . are anything that you first acquire and then release (memory, files, mutexes, device driver contexts, etc.)

RAII

- **Acquire** the resource in **ctor**.
- **Release** the resource in **dtor**.
- The compiler **guarantees** that dtor will be called $\Rightarrow$ the resource will always be **released** $\Rightarrow$ no resource leaks!

This is called "resource allocation is initialization" (**RAII**) idiom.

RAII

Resources

. . . are anything that you first acquire and then release (memory, files, mutexes, device driver contexts, etc.)

RAII

- **Acquire** the resource in **ctor**.
- **Release** the resource in **dtor**.
- The compiler **guarantees** that dtor will be called $\Rightarrow$ the resource will always be **released** $\Rightarrow$ no resource leaks!

This is called "resource allocation is initialization" (**RAII**) idiom.

Remember

One object – one resource! (Why?)