The Skeleton of Modern C++

Matteo Cicuttin

CERMICS
École Nationale des Ponts et Chaussées
Marne-la-Vallée

Nicola Gigante

Università di Udine

INFN-CNAF, Bologna, July 7th, 2017

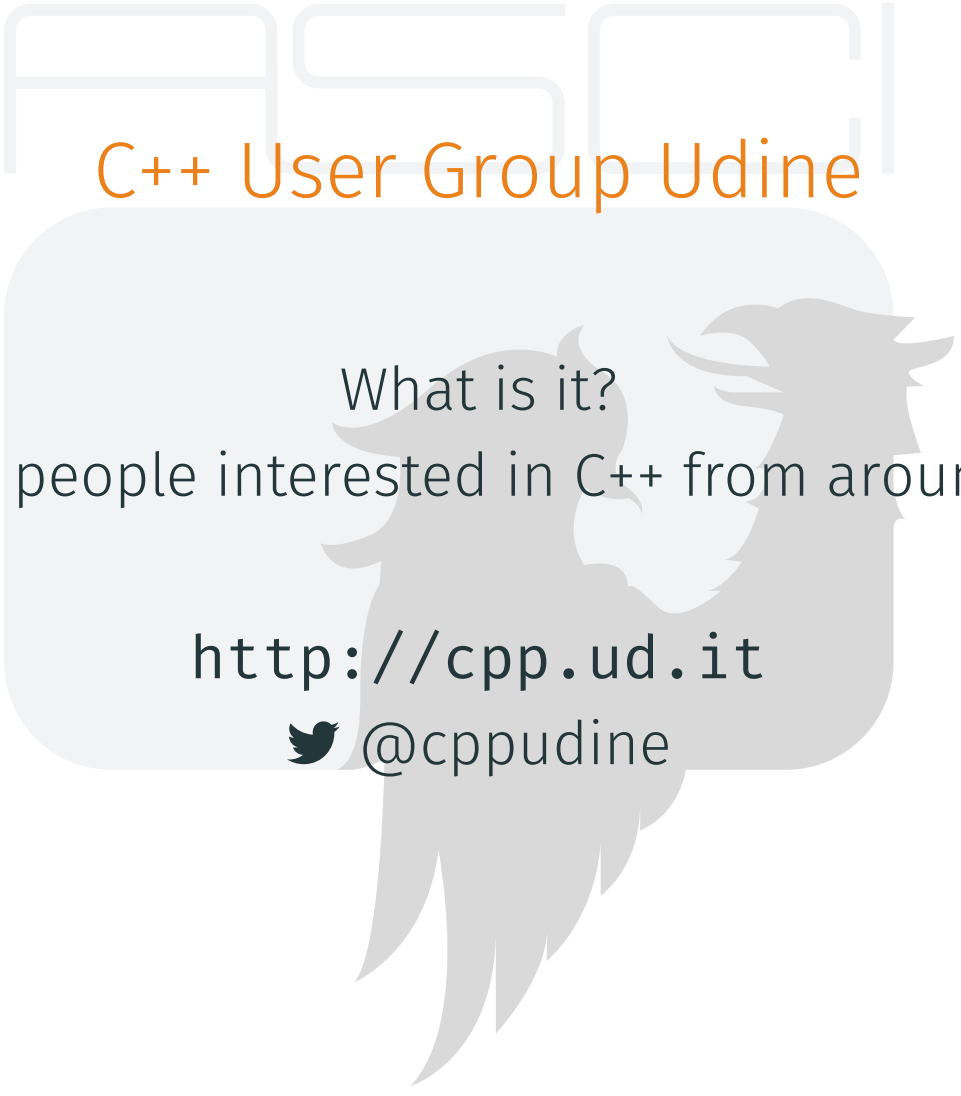
Who we are

Matteo Cicuttin

- Post-doc researcher at CERMICS, École des Ponts ParisTech
- PhD at University of Udine in 2015
- Research area: Discontinuous Skeletal (HHO) methods
- I'm on the CS side, not on the math side!

Nicola Gigante

- Ph.D. student at University of Udine
- Research Area: automated temporal reasoning in AI



C++ User Group Udine

What is it?

Group of people interested in C++ from around Udine

`http://cpp.ud.it`

@cppudine

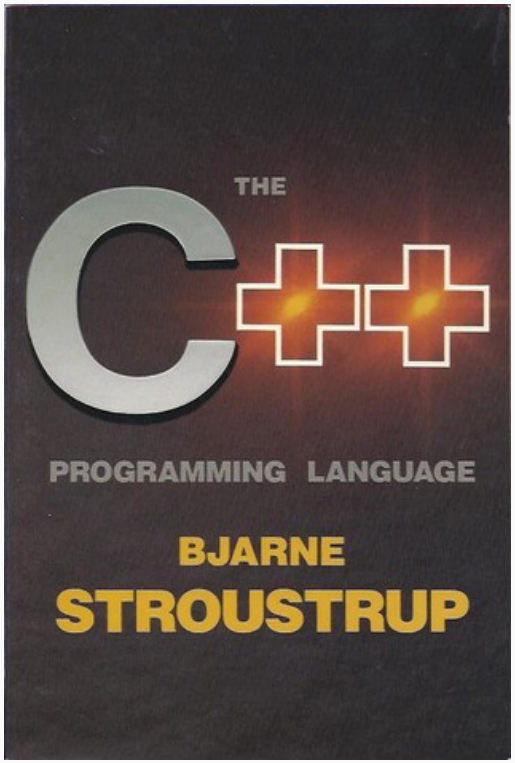
Schedule

Brief summary of what we'll talk about:

1. First part (Nicola):
   - Why still talking about C++?
   - What is Modern C++?
   - Two examples:
     - Zero overhead abstraction
     - Generic programming
2. Second part (Matteo):
   - A concrete example: generic programming for numerical applications
   - The context: discontinuous skeletal methods

Modern C++

Why still talking about C++?



We are in 2017, why still talk about C++?

- C++ has been around for more than 30 years now!
- First book on C++ published in 1986.
- Hasn't everything been settled yet?

Why still talking about C++? (2)

Turns out there's still a lot to say.

- The language evolved dramatically since 1986.
- Generic programming introduced in mid-'90s.
- First ISO standard published in 1998.
- Alexandrescu's "Modern C++ Design" published in 2000.
- Major evolution of the ISO standard published in 2011.
- C++11 enabled the diffusion of Modern C++ techniques.
- Subsequent revisions published in 2014 and 2017.
- Next planned for 2020.

Why still talking about C++? (3)

C++ is still one of the most suitable languages for:

- Really complex software
- Scarce resources environments

Why still talking about C++? (4)

On the other hand:

- C++ is used, but is usually not taught.
- People usually come from low level languages like C, or high level languages like Java,
- then they learn C++ while using it.
- But C++ is complex, and different from other languages.

Why still talking about C++? (5)

This often results into suboptimal usage patterns.

- This is wasteful, at best,
- and dangerous at worst, if cost of failures is high.
- Software matters. Always use your tools at best.

C++11 feels like a new language.

Bjarne Stroustrup

The C++ Programming Language, 4th edition

Modern C++ (2)

What is Modern C++?

- Not a new language, but a new way to use the language
- Encouraged by recent language features.
- Main advantages:
  - Code is high level, readable, easy to maintain, but
  - you still have complete control over the performance.

Zero-cost abstractions

Every studied person knows that...

High-level = Slow

Low-level = Fast

Debunking a myth

High-level = Slow

Low-level = Fast

Debunking a myth

~~High-level = Slow~~

~~Low-level = Fast~~

- Compilers have advanced dramatically in the last decades!
- We have strong optimization techniques.
- Higher-level code can be even more efficient because the compiler knows more about the intent of your code!
- Modern C++ is all about taking advantage of this observation.

Zero-cost abstraction

C++ supports the use of so-called zero-cost abstractions.

- We can write very high-level code
- which does not cause runtime overhead
  w.r.t. equivalent low level code.

You don’t pay what you don’t use

A key C++ design principle: you don’t pay what you don’t use.

- Structs and classes contain only fields, no boilerplate, unless you have virtual methods.
- Exceptions do not slow down the non-throwing code path.
- Method calls are always direct, unless you call a virtual method.
- Lambda functions are just normal functions, unless you need to carry state.
- Arrays and vectors are contiguous.
- …

You don’t pay what you don’t use (2)

Some examples

https://godbolt.org/g/U9EcgB
https://godbolt.org/g/SbPqmw
https://godbolt.org/g/BgDDLQ

Type-driven development

A key point of Modern C++ is the extensive use of types.

- Types in C++ are more than just annoying annotations
- The C++ type-system is really expressive:
  - Types can encode safety properties,
  - invariants,
  - preconditions
- Given the low overhead introduced by abstractions, types can be pervasive!

Generic Programming

Generic Programming

Consider some code:

```
int min(int a, int b) {
  return a < b ? a : b;
}
```

Generic Programming

Consider some code:

```
int min(int a, int b) {
  return a < b ? a : b;
}
```

Then, suppose you need the same function for another type:

```
float min(float a, float b) {
  return a < b ? a : b;
}
```

Generic Programming

Consider some code:

```
int min(int a, int b) {
  return a < b ? a : b;
}
```

Then another:

```
double min(double a, double b) {
  return a < b ? a : b;
}
```

Generic Programming

Consider some code:

```
int min(int a, int b) {
  return a < b ? a : b;
}
```

And another:

```
short min(short a, short b) {
  return a < b ? a : b;
}
```

Generic Programming (2)

How to solve the code duplication?

```
int min(int a, int b) {
  return a < b ? a : b;
}
```

Generic Programming (2)

How to solve the code duplication?

```
template<typename T>
T min(T a, T b) {
  return a < b ? a : b;
}
```

Abstract over types.

Templates

Templates are the mechanism used by C++ to support generic programming.

```
template<typename T>
T min(T a, T b) {
  return a < b ? a : b;
}

int a = min(3, 4);        // 3
double b = min(3.4, 3.5); // 3.4
```

Templates (3)

We can also write class templates.

```
template<typename T>
class vector {
  T *data = ...

  ...
};
```

Templates (2)

The template mechanism is very simple:

- At the use site, the template code is instantiated:
  - a new entity (class or function) is declared,
  - template arguments are substituted as-is.
- The instantiation is a stand-alone entity:
  - You write the generic code.
  - The compiler compiles the specific code.
  - No trace of the generic code is kept at runtime.

Generic Programming (3)

The Standard Template Library is a great example of generic programming.

```
std::vector<int> v = { 4, 3, 2, 1 };


int ten = std::accumulate(begin(v), end(v));

int one = *std::min_element(begin(v), end(v));

bool warning = std::any_of(begin(v), end(v), [](int i) {
  return i > 42;
});
```

Remember: all of these functions are at least as fast as low-level hand-written code.

Generic Programming (3)

The Standard Template Library is a great example of generic programming.

```
std::list<int> v = { 4, 3, 2, 1 };


int ten = std::accumulate(begin(v), end(v));

int one = *std::min_element(begin(v), end(v));

bool warning = std::any_of(begin(v), end(v), [](int i) {
  return i > 42;
});
```

Remember: all of these functions are at least as fast as low-level hand-written code.

Generic Programming (3)

The Standard Template Library is a great example of generic programming.

```
std::deque<int> v = { 4, 3, 2, 1 };


int ten = std::accumulate(begin(v), end(v));

int one = *std::min_element(begin(v), end(v));

bool warning = std::any_of(begin(v), end(v), [](int i) {
  return i > 42;
});
```

Remember: all of these functions are at least as fast as low-level hand-written code.

Standard algorithms

The Standard Library provides a lot of generic algorithms to be used on containers.

- The key concept of iterator allows for a generic interface.
- Pointers are iterators for C-style arrays,
  so you can use them on old code!

```
int func(int *values, int size) {
  std::sort(values, values + size);
}
```

- The generic implementation can some times be even faster than equivalent C code (see later).

Templates and type-driven development

Templates are the core part of the expressive C++ typesystem.

- Use types to form a domain-specific vocabulary.
- Enforce safety constraints by making invalid state unrepresentable.
- The low-overhead of C++ abstractions allows you to use this techniques even in the most cpu-intensive code!

Templates and type-driven development (2)

Some examples

https://godbolt.org/g/Mp7WhK
https://gist.github.com/f0677f6f4f97b9f08c5e

Final thoughts

Wrap up

So what is Modern C++?

- A way to think about and organize code,
- writing high-level programs,
- without overhead w.r.t. equivalent low-level code,
- while still keeping precise control over performance.

How to adopt modern techniques?

Adopting new techniques is difficult.
But it doesn't have to happen all at once.

- During day-to-day writing, think about types.
- Use standard algorithms whenever possible.
- Use smart pointers to handle memory management.
- Use modern C++ libraries like Boost, Type-Safe, ecc...
    - and learn from their APIs.

Resources

Excellent resources exist.

- Books
  - Invest time to read a recent book. It's worth it!
  - List of C++ books:
    `https://stackoverflow.com/questions/388242`
- C++ Standard FAQs
  - `https://isocpp.org/faq`
- StackOverflow FAQ questions.
  - Look for `c++-faq` tag.
- Conference talks. C++Now, Meeting C++, CppCon.

Questions?