NHR FAU

Friedrich-Alexander-Universität Erlangen-Nürnberg

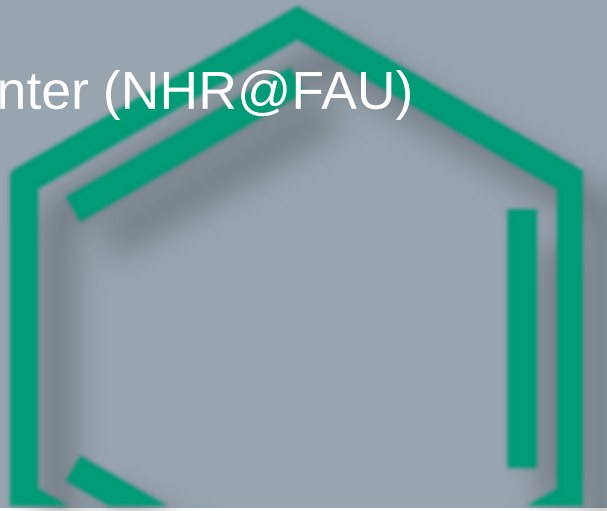
C++ on tranquilizers –

writing simpler and more reliable code for HPC

Michael Panzlaff

Erlangen National High Performance Computing Center (NHR@FAU)

HPC Café, Sep 22, 2026



Survey

Who in this room uses C++?

- Who (somewhat) regularly writes C++?
- Who has written C++ before?
- Who (occasionally) reads C++?
- Who doesn’t do any of those?

Which programming language for HPC?

A decades old problem:

- Fortran (probably oldest language still in use)
- C
- C++ (~~C with classes~~)
- ~~Matlab~~
- Python (via numpy, scipy, pytorch, etc.)
- Julia
- DSLs
- Rust (?)

Which programming language for HPC?

Language	Difficulty	Abstraction / Productivity	Complexity at Scale	Speed
Fortran	Medium	Low	?	High
C	Medium	Low	Hard	High
C++	High	Low to medium	Medium	High
Python	Easy	High	Hard	High*

*with native libraries only, slow otherwise

Why choose C++ for HPC?

- Low level features for writing kernels
- Higher level features for everything else
- Large library catalog
- Static type system
- Advantages over plain C
  - Automatic/easier resource management
  - Less boiler plate code
  - Less prone to bugs
  - Object Oriented Programming
  - Templates
  - Much more!

Why **not** choose C++ for HPC?

- There is no (good) C++ compiler for your architecture
- One of
  - You don‘t write kernels
  - You only write kernels
- You want to prototype very quickly
- You need ABI compatibility (libraries)

Why dedicate an HPC-Café to C++?

- It is commonly used in HPC Code:
  - All CUDA code is theoretically C++ code
  - Same for HIP/SyCL
- It is a complex language
  - Many ways to do the same thing
  - New standard (thus features) every three years
- Often treated as “C with classes”
  - Not untrue, but (in my opinion) not its key feature
- Its steep learning curve pays off

Why bother with a complex language?

- Low level control vs. high level tradeoff
- We are working on software with complex requirements
  - Maximum speed/efficiency
  - Doesn‘t crash (→ e.g. no memory errors)
  - Correct code (→ no bugs)
  - “Easy to read“, good abstractions
- “No free lunch“

What I want to teach you

- Yes, C++ is complicated, but there are reasons
  - C backwards compatibility
  - Burden of C++' language history ("pre c++11 era")
- How to write better code with less effort
  - Resource acquisition is initialization (RAII)
  - Zero cost abstractions
  - Understand what your compiler can (and cannot) do

C++ Hello World

A “hello world” is simple enough:

```cpp
#include <cstdio>

int main() {
    auto foo = 3.1415926;
    std::printf("Hello World: %10.7f\n", foo);
}
```

In C++, you can also do:

```cpp
#include <iostream>
#include <iomanip>
int main() {
    auto foo = 3.1415926;
    std::cout << "Hello World: " << std::setw(10) <<
        std::setprecision(7) << foo << std::endl;
}
```

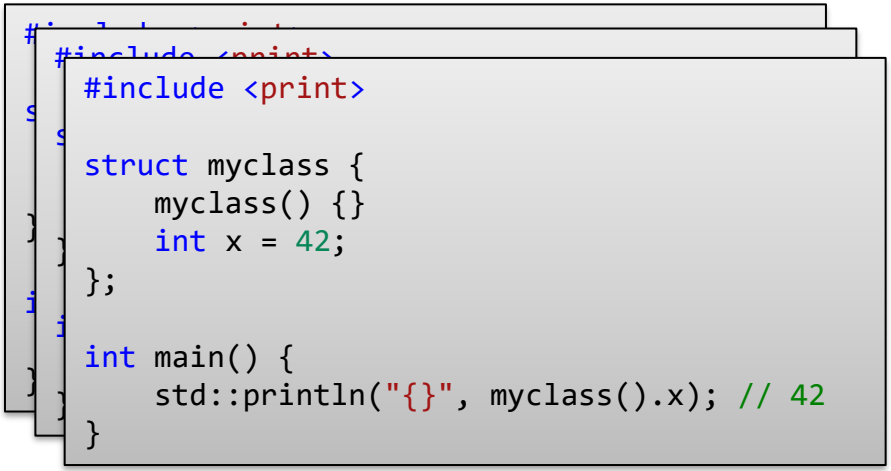
But perhaps you should do this:
(C++23 only)

```cpp
#include <print> // C++23

int main() {
    auto foo = 3.1415926;
    std::println("Hello World: {:10.7}", foo);
}
```

C++ Class Hello World (primitive member)

How do we initialize x?



```cpp
#include <print>

struct myclass {
    myclass() {}
    int x = 42;
};

int main() {
    std::println("{}", myclass().x); // 42
}
```

C++ References

A reference behaves as [illegible] pointer in compiled code.

Who needs this?
Aren't pointers

```cpp
#include <print>

struct myvec {
    int *data = new int[64];
    ~myvec() { delete[] data; }

    int &operator[](unsigned index) {
        return data[index];
    }
};

int main() {
    myvec v;
    v[42] = 1337;
    std::println("{}", v[42]); // 1337
}
```

They are necessary to implement e.g. containers

C++ Class Hello World (reference member)

How do we initialize x?

```cpp
#include <print>

struct myclass {
    myclass(int &a) {}
    int &x = a; // error
};

int main() {
    int foo = 42;
    std::println("{}", myclass(foo).x);
}
```

The (in my opinion) most important feature of C++

So why does C++ have all this weird and complicated initialization behavior?

RAII

Resource Acquisition Is Initialization (RAII)

Programming technique based on C++ rules of object lifetime:

- Avoids the use of uninitialized variables*
- Avoids resources leaks (memory, threads, open files, locked mutexes, etc.)
- …in a way that is exception-safe (via stack unwinding)
- Avoids double frees

*does not apply (since not mandatory) to primitive types and raw pointer types!

Resource Acquisition Is Initialization (RAII)

Whenever a (class) variable is allocated it is also initialized (constructed)

…

Whenever a (class) variable is deallocated is also deinitialized (destructed)

Example “myvec” revised

What’s wrong with “myvec”?

```
#include <print>
#include <memory>

struct myvec {
    std::unique_ptr<int[]> data{r

    int &operator[](unsigned index) {
        return data[index];
    }
};

int main() {
    myvec v;
    myvec v2 = v; // <-- compile error
    v[42] = 1337;
    std::println("{}", v2[42]); // 1337
}
```

```
<source>: In function 'int main()':
<source>:11:16: error: use of deleted function 'myvec::myvec(const myvec&)'
   11 |     myvec v2 = v; // <-- compile error
      |                ^
<source>:3:8: note: 'myvec::myvec(const myvec&)' is implicitly deleted because
    3 | struct myvec {
      |        ^~~~~
<source>: At global scope:
<source>:3:8: error: use of deleted function 'std::unique_ptr<_Tp [], _Dp>::uni
```

How convenient, we get a compile error ☺

Example “myvec” revised

But I want a copyable “myvec”:

```
#include <print>
#include <vector>

struct myvec {
    myvec() : data(64) {}
    std::vector<int> data;
    //std::vector<int> data{64};
    // ^ unfortunately does something else
    int &operator[](unsigned index) {
        return data[index];
    }
};

int main() {
    myvec v;
    myvec v2 = v;
    v[42] = 1337;
    std::println("{}", v2[42]); // 0
}
```

In most cases std::vector is probably the right choice. Copies are handled as deep copies.

Stop using new/delete (and start to use RAII wrappers)

"But I know what I am doing. No way this is wrong":

```
#include <vector>

void dothings(int *data, unsigned count);

void foobar() {
    std::vector<int> vec(64);
    dothings(vec.data(), vec.size());
}
```

On exception, stack is unwound and the destructor of "vec" is called. No memory leaks ☺.
…oh, and less code!

Interactive Example (Compiler Explorer)

std::span, std::mdspan, std::linalg

Handy helpers for avoiding pointer and element count pairs:

- std::span – Pointer and element count
- std::mdspan – Pointer with multidimensional extents and strides
- std::linalg – BLAS functions

Interactive Example (Compiler Explorer)

Why C++ is handy

Interactive Example (Compiler Explorer)

C++ and CUDA

Interactive Example (Compiler Explorer)

Is this relevant for HPC?

- Yes:
  - Writing code that is better readable
  - Writing code with less (resource allocation) bugs
  - You want to publish a library (and not just a paper)
- No:
  - “One time” smaller projects, proof of concepts
  - You just writing kernels (and nothing besides)
  - You are already using Rust

Final Learnings

- Stop wasting your time with resource allocation and start to use RAII
  - NUMA insensitive:
    - std::vector (unless NUMA sensitive)
  - NUMA sensitive (and other non-array pointers)
    - std::unique_ptr (preferred, single owner)
    - std::shared_ptr (use when ownership is unclear)
  - Do not use new/delete
    (new is ok for smart pointer init, consider std::make_unique, std::make_shared)
- Consider using thrust for CUDA code
- If in doubt, check en.cppreference.com
- Avoid pre C++11 projects
- C++ is still “messy” at times (is Rust going to be the rescue?)