Modern C++ and its Software Ecosystem

Attila Krasznahorkay
● Will try to convince you why C++ is great, and why we shouldn’t try to get rid of it all too eagerly
● After a bit of historical context I will go over aspects of the language that I think are important for writing good code in it
● C++ is not a “simple” language 😞
  ○ It is also tremendously rich by now. I do not believe that even every member of the standards committee each know every aspect of it.
Overview
A Brief History

● It came from the desire of using state of the art language concepts of the time with C’s close-to-hardware performance
  ○ Both considerations still dominate the language’s evolution
● 1985 - C++ is released outside of Bell Labs
● 1998 - C++(98) is blessed as an ISO standard
● 2011 - Start of the “modern era” with C++11
Language Standardization

- Is one of surprisingly few ("live") languages that have strict standards defined
  - Showing how much of the world depends on C++
- Does not have a single reference implementation, though it also does not have too many of them
  - As complex as the language is, every implementation has its own quirks. Testing as many compilers as you can is always a very good idea.
- All (modern) compilers allow you to specify which standard you want to compile your code with
  - A good reference for standard support is: https://en.cppreference.com/w/cpp/compiler_support
Language Basics

● Is based on C, but is not simply a superset of it
  ○ In reality though 99.99% of C code will work just fine with a C++ compiler

● Is composed of 2 main parts
  ○ The C++ compiler, implementing all of the “compiler features” of the standard
  ○ The C++ standard library, implementing all of the “library features” of the standard

● Has a lot of excellent free resources to start learning/using it
  ○ I myself started with it around 2000 by downloading a free book as a PDF…

```cpp
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << "Received arguments:\n";
    for (int i = 0; i < argc; ++i) {
        std::cout << "  - " << argv[i] << "\n";
    }
    std::cout << std::flush;
    return 0;
}
```
Its close ties to C have benefitted C++ a lot

- Linux is C, so any low-level hardware/OS access will always have a C interface
  - Which we can also directly use from C++ as well
  - Windows and macOS are a little different, but you can do a lot with just C++ on those platforms as well
- Since C (especially on Linux) is so important, basically every modern language can cooperate with it
  - Which makes it possible to cooperate with all those languages from C++ as well
    - The interoperability with Python became very successful, with many “new languages” trying to replicate that success
    - Allowing for 100% interoperability with Windows / macOS as well
Language Features to Know
(About)
Disclaimer

- This is of course not meant as a C++ course / tutorial
- I will be highlighting language aspects that I personally think are important for modern NHEP code
- Code examples were not checked verbatim, errors/typos are a real possibility…
Object Orientation

- **Is one of the main features of C++**
  - It is one of its strongest features as well. But as with anything else, it is very easy to over-use it.

- **We should not be afraid of it, if used correctly**
  - There is a lot of talk about functional programming lately. Which can also be very quickly over-used.
  - I personally believe that most applications benefit from using some objects “with states”, while trying to keep the “long term state” of long-lived objects to a minimum.

- **Another generally good design is to keep “data objects” strictly separate from “algorithmic code”**
  - Which on its face is contrary to object orientation, but greatly helps with code structuring

```cpp
struct Particle {  
    float m, px, py, pz;
};

class Electron : public Particle {  
public:
    float pt() const { std::sqrt(px*px + py*py); }
};

float invariantMass(const Particle& p1,  
                    const Particle& p2);
```
Templating has come a long way since its first introduction to C++

- It allows for very powerful generalisation in our code
  - But it also comes with significant costs. Over-use of it, as with anything, is a bad idea.

- For any functionality, always consider how you can provide a thin user-friendly templated interface over a “compiled”, possibly non-user-friendly low-level interface
  - It will not always be possible to do this, but in many cases it is

- Variadic templates can be amazing!
  - But do be mindful of code readability!

```cpp
void setZeroImpl(void* p, std::size_t l) {
    std::memset(p, 0, l);
}

template<typename T,
    std::enable_if_t<std::is_standard_layout_v<T>,
    bool> = true>
void setZero(T& obj) {
    setZeroImpl(&obj, sizeof(obj));
}
```
As powerful as templates can be in C++17 already, we will likely rewrite a lot of our core code in ATLAS once constraints and concepts become available.

- The new formalism should allow for much easier-to-read code.
Standard Containers

Sequence containers
Sequence containers implement data structures which can be accessed sequentially.

- **array (C++11)**: static contiguous array
- **vector**: dynamic contiguous array
- **deque**: double-ended queue
- **forward_list (C++11)**: singly-linked list
- **list**: doubly-linked list

**std::span** and **std::mdspan** will likely reform in the coming years how we interact with vector-type data in memory.

Associative containers
Associative containers implement sorted data structures that can be quickly searched (O(log n) complexity).

- **set**: collection of unique keys, sorted by keys
- **map**: collection of key-value pairs, sorted by keys, keys are unique
- **multiset**: collection of keys, sorted by keys
- **multimap**: collection of key-value pairs, sorted by keys

Unordered associative containers
Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched (O(1)) amortized, (O(n) worst-case complexity).

- **unordered_set (C++11)**: collection of unique keys, hashed by keys
- **unordered_map (C++11)**: collection of key-value pairs, hashed by keys, keys are unique
- **unordered_multiset (C++11)**: collection of keys, hashed by keys
- **unordered_multimap (C++11)**: collection of key-value pairs, hashed by keys

Container adaptors
Container adaptors provide a different interface for sequential containers.

- **stack**: adapts a container to provide stack (LIFO data structure)
- **queue**: adapts a container to provide queue (FIFO data structure)
- **priority_queue**: adapts a container to provide priority queue

**span**
A span is a non-owning view over a contiguous sequence of objects, the storage of which is owned by some other object.

- **span (C++20)**: a non-owning view over a contiguous sequence of objects
Standard Algorithms

- Making use of algorithms defined in the standard library is generally a good idea
  - The language is developed to make these algorithms as efficient to execute as possible
  - If your code is designed to perform these types of operations, it can likely be implemented efficiently with C++
- Generic support for non-CPU execution is first concerned about supporting these algorithms!
Modern(?) Data Structures

● In a lot of “LHC code” we went overboard with object orientation and polymorphism to describe detector data in memory
  ○ We have been un-doing most of it in the last decade…

● My personal views:
  ○ Use polymorphism very sparingly in your data types, never “own” anything through a base type pointer
    ■ In general only containers should own anything through pointers
  ○ Make generous use of standard library container types
    ■ There can be reasons for using a custom container type. But that should only come if you absolutely cannot do what you want with just standard containers.
    ■ Remember though that the internals of such types are not standardized. To be kept in mind with I/O code!
  ○ Be mindful of using AoS vs. SoA, consider making use of podio
    ■ But don’t put it above everything else! In ATLAS raw performance has so far proved secondary to a convenient user interface for “algorithm performance”.
Memory Management (1)

- Is one of the more difficult problems in any large project
- In HEP applications we usually manage data objects of different lifetimes in central “stores”
  - Making it easier for independent components to produce and use a given data object
- Smart pointers have made this type of code a lot more readable
  - In modern code you should only construct and pass around objects in heap memory using smart pointers!
  - In such code any “bare pointer” is known not to own the thing that it points to
Memory Management (2)

- One of the first issues in a heterogeneous application is the management of memory
- **C++17** introduced a very powerful new way of managing objects/containers in vendor specific memory types
  - At the current moment still requiring significant effort to use, but **C++23** should simplify things a little further
Multi-Threading

- One of the places where the standard library is just not enough for HEP
  - Even things like OpenMP don’t quite scale for our applications
- Many years ago the decision was made to use (one)TBB for general multi-threading in HEP
  - Which, I believe, proved as one of the most successful standardizations in our field
- For small applications of course feel free to just use std::thread
  - But for anything bigger, which may need to interact with ROOT / Geant4 / etc., TBB is the way to go
Heterogeneous Computing

- **Is becoming very important for HEP**
  - Lucky for us, C(++) has been the language of choice for writing “general applications” for GPGPUs since the start

- **We are in a very tumultuous time with “heterogeneous languages” right now**
  - One technical upside is that they are practically all based on LLVM, aiding in harmonisation efforts between them

- **I believe that in a few more years C++ will standardize a large portion of these languages**
  - Or at least make it much easier to make use of vendor specific libraries in standard C++ code

![Diagram showing ROCm/HIP, CUDA, and oneAPI/SYCL]
Emerging Possibilities / C++XY Features

- **Unified Executors** ([P2300R4](#))
  - This is what I have the highest hopes for, but am the most scared of at the same time
    - Some of the interfaces in the proposal do not look nearly as user-friendly as I would like 😞
  - Once in the C++ standard, should allow hardware manufacturers to build vendor-specific binaries out of standard C++ code with their own compilers for their own devices
  - Will allow for the declaration of execution graphs (DAGs) for inter-dependent algorithms, which may each run on different types of hardware
    - May make some/most of TBB obsolete as well

- **Coroutines**
  - I am even more skeptical about this one, as I still didn’t see any good explanation of what people want to use it for exactly
  - Seemingly it could come in handy for asynchronous execution on heterogeneous hardware
    - But as far as I’m aware, neither Intel or NVIDIA are considering it at the moment in their compilers
  - Still, has promise once improvements planned for C++23 will arrive
Code Build / Management
C++ does not define a standard build or packaging system
- Unlike many modern languages, which come with built-in source code management solutions

This is both a blessing and a curse
- It allowed for the development of many different build systems for different levels of project complexity
- It also presents a big hurdle in managing large projects that depend on many other projects

My personal suggestion
- Use GNU Make for anything trivial that also doesn’t need much portability
- Use CMake for building anything larger
- To manage many CMake projects together… 😞

```cpp
g++ -o helloWorld helloWorld.cpp
```
Summary

- Having a formal standard for a language that NHEP uses in the long term should not be taken lightly
  - With any new thing that we pick up, we must seriously consider how long it may live
- Developments are very active in making C++2x even more capable, allowing its usage on all current and future hardware
  - The “extensions” that we have to use currently for heterogeneous hardware may completely go away eventually
- C++ has lived, it lives, and it will live (for a long time to come…)

<table>
<thead>
<tr>
<th>Working Group</th>
<th>Working Area</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 1</td>
<td>PLIP (Programming Languages for Industrial Processes)</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 2</td>
<td>Pascal</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 3</td>
<td>APL</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 4</td>
<td>COBOL</td>
<td>Active</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 5</td>
<td>FORTRAN</td>
<td>Active</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 6</td>
<td>ALGOL</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 7</td>
<td>PI/I</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 8</td>
<td>BASIC</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 9</td>
<td>Ada</td>
<td>Active</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 10</td>
<td>Guidelines</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 11</td>
<td>Binding Techniques</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 12</td>
<td>Conformity</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 13</td>
<td>Modula-2</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 14</td>
<td>C</td>
<td>Active</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 15</td>
<td>POSIX</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 16</td>
<td>ISLisp</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 17</td>
<td>Prolog</td>
<td>Active</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 18</td>
<td>FIMS (Form Interface Management System)</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 19</td>
<td>Formal Specification Languages</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 20</td>
<td>Internationalization</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 21</td>
<td>C++</td>
<td>Active</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 22</td>
<td>PCTE (Portable Common Tool Environment)</td>
<td>Disbanded</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 23</td>
<td>Programming Language Vulnerabilities</td>
<td>Active</td>
</tr>
<tr>
<td>ISO/IEC JTC 1/SC 22/WG 24</td>
<td>Linux Standard Base (LSB)</td>
<td>Active</td>
</tr>
</tbody>
</table>