Modern C++ and its Software Ecosystem

Attila Krasznahorkay

BROOKHAVEN NATIONAL LABORATORY
Jefferson Lab
HSF

SOFTWARE & COMPUTING round table

Introduction

- 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…

```
#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;
}
```

OS / Language Ties

- 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

```
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);
```

- 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

Templating

- 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!

```
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));
}
```

Constraints / Concepts

- 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

Constraints and concepts (since C++20)

This page describes the core language feature adopted for C++20. For named type requirements used in the specification of the standard library, see named requirements. For the Concepts TS version of this feature, see here.

Class templates, function templates, and non-template functions (typically members of class templates) may be associated with a *constraint*, which specifies the requirements on template arguments, which can be used to select the most appropriate function overloads and template specializations.

Named sets of such requirements are called *concepts*. Each concept is a predicate, evaluated at compile time, and becomes a part of the interface of a template where it is used as a constraint:

Run this code

```cpp
#include <string>
#include <cstddef>
#include <concepts>

// Declaration of the concept "Hashable", which is satisfied by any type 'T'
// such that for values 'a' of type 'T', the expression std::hash<T>{}(a)
// compiles and its result is convertible to std::size_t
template<typename T>
concept Hashable = requires(T a)
{
    { std::hash<T>{}(a) } -> std::convertible_to<std::size_t>;
};

struct meow {};

// Constrained C++20 function template:
template<Hashable T>
void f(T) {}
//
// Alternative ways to apply the same constraint:
// template<typename T>
//     requires Hashable<T>
// void f(T) {}
//
// template<typename T>
// void f(T) requires Hashable<T> {}

int main()
{
    using std::operator""s;

    f("abc"s);    // OK, std::string satisfies Hashable
    // f(meow{}); // Error: meow does not satisfy Hashable
}
```

Standard Containers

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

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

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 (class template)
`map`	collection of key-value pairs, sorted by keys, keys are unique (class template)
`multiset`	collection of keys, sorted by keys (class template)
`multimap`	collection of key-value pairs, sorted by keys (class template)

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 (class template)
`unordered_map` (C++11)	collection of key-value pairs, hashed by keys, keys are unique (class template)
`unordered_multiset` (C++11)	collection of keys, hashed by keys (class template)
`unordered_multimap` (C++11)	collection of key-value pairs, hashed by keys (class template)

Container adaptors

Container adaptors provide a different interface for sequential containers.

`stack`	adapts a container to provide stack (LIFO data structure) (class template)
`queue`	adapts a container to provide queue (FIFO data structure) (class template)
`priority_queue`	adapts a container to provide priority queue (class template)

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 (class template)

- Containers in the standard library are pretty smart, make use of them!
- std::span and std::mdspan will likely reform in the coming years how we interact with vector-type data in memory

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!

Execution policies

Most algorithms have overloads that accept execution policies. The standard library algorithms support several execution policies, and the library provides corresponding execution policy types and objects. Users may select an execution policy statically by invoking a parallel algorithm with an execution policy object of the corresponding type.

Standard library implementations (but not the users) may define additional execution policies as an extension. The semantics of parallel algorithms invoked with an execution policy object of implementation-defined type is implementation-defined.

Parallel version of algorithms (except for std::for_each and std::for_each_n) are allowed to make arbitrary copies of elements from ranges, as long as both std::is_trivially_copy_constructible_v<T> and std::is_trivially_destructible_v<T> are true, where T is the type of elements.

Defined in header <execution>
Defined in namespace std::execution (since C++17)

sequenced_policy (C++17) parallel_policy (C++17) parallel_unsequenced_policy (C++17) unsequenced_policy (C++20)	execution policy types (class)
seq (C++17) par (C++17) par_unseq (C++17) unseq (C++20)	global execution policy objects (constant)

Defined in namespace std

is_execution_policy (C++17)	test whether a class represents an execution policy (class template)

Feature testing macro: __cpp_lib_parallel_algorithm (for parallel version of algorithms).
Feature testing macro: __cpp_lib_execution (for execution policies).

Non-modifying sequence operations

Defined in header <algorithm>

all_of (C++11) any_of (C++11) none_of (C++11)	checks if a predicate is true for all, any or none of the elements in a range (function template)
ranges::all_of (C++20) ranges::any_of (C++20) ranges::none_of (C++20)	checks if a predicate is true for all, any or none of the elements in a range (niebloid)
for_each	applies a function to a range of elements (function template)
ranges::for_each (C++20)	applies a function to a range of elements (niebloid)
for_each_n (C++17)	applies a function object to the first n elements of a sequence (function template)
ranges::for_each_n (C++20)	applies a function object to the first n elements of a sequence (niebloid)
count count_if	returns the number of elements satisfying specific criteria (function template)
ranges::count (C++20) ranges::count_if (C++20)	returns the number of elements satisfying specific criteria (niebloid)
mismatch	finds the first position where two ranges differ (function template)
ranges::mismatch (C++20)	finds the first position where two ranges differ (niebloid)
find find_if find_if_not (C++11)	finds the first element satisfying specific criteria (function template)

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)

Smart pointers

Smart pointers enable automatic, exception-safe, object lifetime management.

Defined in header `<memory>`

Pointer categories

`unique_ptr` (C++11)	smart pointer with unique object ownership semantics (class template)
`shared_ptr` (C++11)	smart pointer with shared object ownership semantics (class template)
`weak_ptr` (C++11)	weak reference to an object managed by `std::shared_ptr` (class template)
`auto_ptr` (deprecated in C++11) (removed in C++17)	smart pointer with strict object ownership semantics (class template)

Helper classes

`owner_less` (C++11)	provides mixed-type owner-based ordering of shared and weak pointers (class template)
`enable_shared_from_this` (C++11)	allows an object to create a `shared_ptr` referring to itself (class template)
`bad_weak_ptr` (C++11)	exception thrown when accessing a `weak_ptr` which refers to already destroyed object (class)
`default_delete` (C++11)	default deleter for `unique_ptr` (class template)

Smart pointer adaptors

`out_ptr_t` (C++23)	interoperates with foreign pointer setters and resets a smart pointer on destruction (class template)
`out_ptr` (C++23)	creates an `out_ptr_t` with an associated smart pointer and resetting arguments (function template)
`inout_ptr_t` (C++23)	interoperates with foreign pointer setters, obtains the initial pointer value from a smart pointer, and resets it on destruction (class template)
`inout_ptr` (C++23)	creates an `inout_ptr_t` with an associated smart pointer and resetting arguments (function template)

- 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

Allocators

Allocators are class templates encapsulating memory allocation strategy. This allows generic containers to decouple memory management from the data itself.

Defined in header <memory>	
allocator	the default allocator (class template)
allocator_traits (C++11)	provides information about allocator types (class template)
allocation_result (C++23)	records the address and the actual size of storage allocated by allocate_at_least (class template)
allocate_at_least (C++23)	allocates storage at least as large as the requested size via an allocator (function template)
allocator_arg_t (C++11)	tag type used to select allocator-aware constructor overloads (class)
allocator_arg (C++11)	an object of type std::allocator_arg_t used to select allocator-aware constructors (constant)
uses_allocator (C++11)	checks if the specified type supports uses-allocator construction (class template)
uses_allocator_construction_args (C++20)	prepares the argument list matching the flavor of uses-allocator construction required by the given type (function template)
make_obj_using_allocator (C++20)	creates an object of the given type by means of uses-allocator construction (function template)
uninitialized_construct_using_allocator (C++20)	creates an object of the given type at specified memory location by means of uses-allocator construction (function template)
Defined in header <scoped_allocator>	
scoped_allocator_adaptor (C++11)	implements multi-level allocator for multi-level containers (class template)
Defined in header <memory_resource> Defined in namespace std::pmr	
polymorphic_allocator (C++17)	an allocator that supports run-time polymorphism based on the std::memory_resource it is constructed with (class template)

Memory resources

Memory resources implement memory allocation strategies that can be used by std::pmr::polymorphic_allocator

Defined in header <memory_resource> Defined in namespace std::pmr	
memory_resource (C++17)	an abstract interface for classes that encapsulate memory resources (class)
new_delete_resource (C++17)	returns a static program-wide std::pmr::memory_resource that uses the global operator new and operator delete to allocate and deallocate memory (function)
null_memory_resource (C++17)	returns a static std::pmr::memory_resource that performs no allocation (function)
get_default_resource (C++17)	gets the default std::pmr::memory_resource (function)
set_default_resource (C++17)	sets the default std::pmr::memory_resource (function)
pool_options (C++17)	a set of constructor options for pool resources (class)
synchronized_pool_resource (C++17)	a thread-safe std::pmr::memory_resource for managing allocations in pools of different block sizes (class)
unsynchronized_pool_resource (C++17)	a thread-unsafe std::pmr::memory_resource for managing allocations in pools of different block sizes (class)
monotonic_buffer_resource (C++17)	a special-purpose std::pmr::memory_resource that releases the allocated memory only when the resource is destroyed (class)

Multi-Threading

Thread support library

C++ includes built-in support for threads, mutual exclusion, condition variables, and futures.

Threads

Threads enable programs to execute across several processor cores.

Defined in header <thread>

`thread` (C++11) manages a separate thread

`jthread` (C++20)

Functions managing

Defined in namespace th

`yield` (C++11)

`get_id` (C++11)

`sleep_for` (C++11)

`sleep_until` (C++11)

oneAPI Threading Building Blocks (oneTBB)

This document contains information about oneTBB. It is a flexible performance library that let you break computation into parallel running tasks.

The following are some important topics for the `novice user`:

- Get Started with oneTBB gives you a brief explanation of what oneTBB is.
- oneTBB Benefits describes how oneTBB differs from typical threading packages.
- Package Contents describes dynamic library files and header files for Windows*, Linux*, and macOS* operating systems used in oneTBB.

The following is an important topic for the `experienced user`:

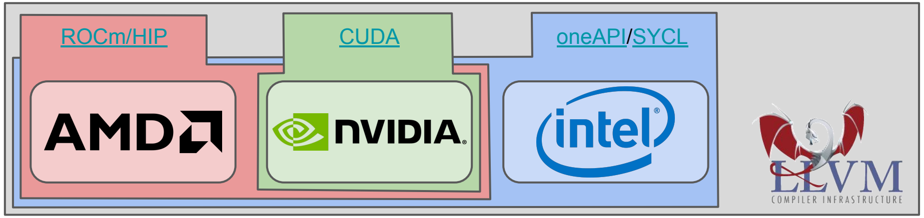
Migrating from Threading Building Blocks (TBB) describes how to migrate from TBB to oneTBB.

- Getting Help and Support
- Notational Conventions
- Introduction
- oneTBB Benefits
- Get Started with oneTBB
  - System Requirements
  - Before You Begin
  - Example
  - Find more
- oneTBB Developer Guide
  - Package Contents

- 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

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

From Simple to Complex

- 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… 😵

Spack ? vcpkg ? rpm

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

```
myExecutable: source1.o source2.o
    g++ -o $@ $^

.SUFFIXES: .cpp .o
.cpp.o:
    g++ -c -o $@ $<
```

```
cmake_minimum_required(VERSION 3.17)
project(MySuperProject VERSION 1.0.0 LANGUAGES CXX)

find_package(Boost REQUIRED)

add_executable(myBoostedExecutable
   source1.cpp source2.cpp)
target_link_libraries(myBoostedExecutable
   PRIVATE Boost::boost)
```

Summary

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

- 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…) 😃

CERN

http://home.cern