



Advanced C++ Programming

Libraries



Preliminaries

Overview & Goals

- *The best code is the code you don't have to write*
- This chapter provides an overview of **available high-quality C++ libraries**
- We won't go into as much detail as for the language-specific chapters
 - Too many libraries, too little time
 - It's more important to get an overview of what is out there – if you decide to use some of these libraries, you'll have to study them in more detail

Libraries Already Covered

We already mentioned/used some parts of the standard library:

- **Containers and Iterators** `<iterator>`, `<vector>`, `<unordered_map>`, ...
- **Algorithms** `<algorithm>`
- **Utilities** (e.g. tuple, move) `<limits>`, `<typeinfo>`, `<tuple>`, ...
- **Memory** `<memory>`, `<new>`, ...
- **Strings** `<string>`
- **I/O Streams** `<iostream>`, `<fstream>`, ...



Standard Library

Regular Expressions

- Part of the standard library since C++11
 - **Note:** compiler language compliance is sometimes achieved more quickly than *full* standard library support for a given language version
- Supports well-known regex operations – e.g. match, search, replace
- Regex syntax defaults to ECMAScript grammar

<http://en.cppreference.com/w/cpp/regex/ecmascript>

<http://en.cppreference.com/w/cpp/regex>

Example in `07_01_regex.cpp`

Filesystem

- Allows you to operate on **paths** and **navigate/iterate** in the filesystem
- Also has operations to query and **modify meta-information** on files (e.g. permissions)
- Developed as a boost library, only recently standardized in **C++17**
- ➡ You should be familiar with this from earlier in the lab, we will use it later in another example

<http://en.cppreference.com/w/cpp/filesystem>

Thread Support Library

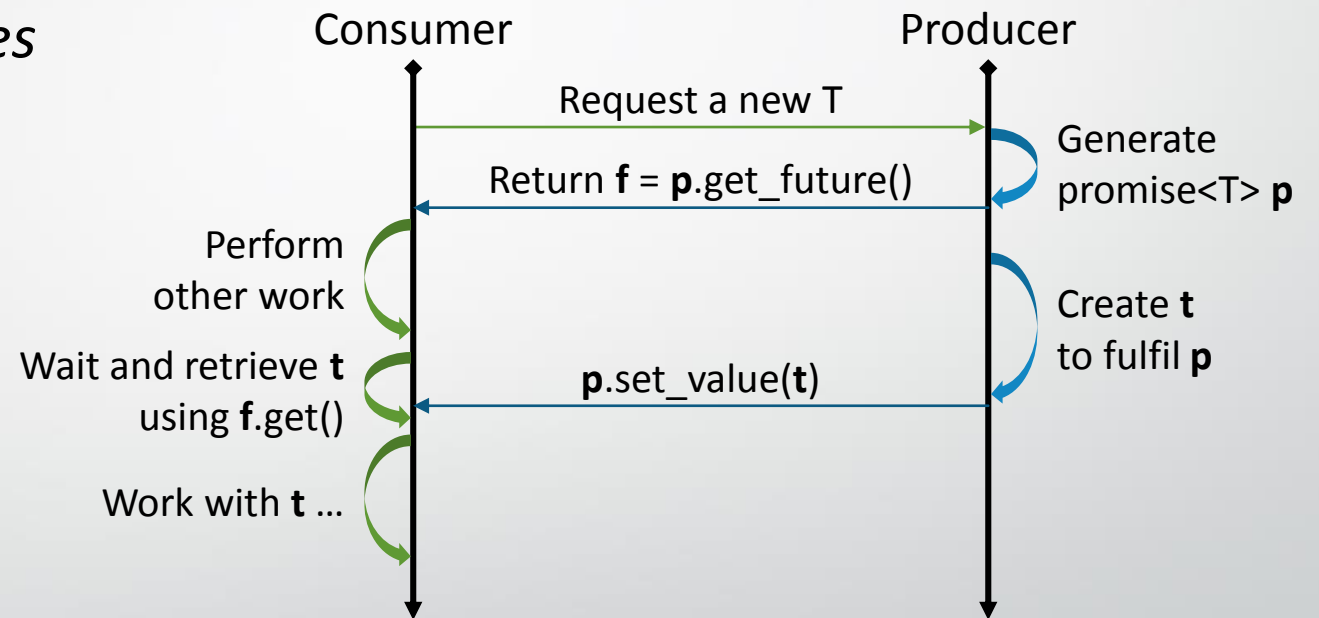
- Provides various features to deal with **concurrency**
 - Threads, mutexes, condition variables, locks, ...
 - How to actually use these to build a *correct* parallel program is the subject of several other lectures
 - Simple example in **07_02_threads.cpp**
- Also a few non-functional values provided:
 - `std::thread::hardware_concurrency()`
 - `std::hardware_destructive_interference_size`
`std::hardware_constructive_interference_size`

<http://en.cppreference.com/w/cpp/thread>

Futures and Promises

Mechanism for *returning values from asynchronous tasks*

- `std::future<T>`
proxy for a value of type `T` that *will become available*
- `std::promise<T>`
means of setting the value associated with a future



Futures are also returned from `std::async`

➡ Promise handling automated

Example in `07_03_future_fs.cpp`

More Concurrency/Parallelism in C++

- Parallel Algorithms (*see chapter 3*)
- Concurrency Technical Specification
 - Extends future, promise etc. <http://en.cppreference.com/w/cpp/experimental/concurrency>
 - e.g. `future.then(...)`, `when_all(...)`
- Atomics <http://en.cppreference.com/w/cpp/atomic>
- ➔ Important to note when implementing parallel code: **C++ memory model**
Especially when writing low-level primitives

http://en.cppreference.com/w/cpp/language/memory_model



Boost Libraries



Overview

- Set of **peer-reviewed C++ libraries** with some common design/build/distribution standards
- **Note:** you don't use "boost", you use a specific set of boost libraries
- Many libraries are **header-only**, some require compilation
- Usually aim for wide compiler and C++ version compatibility

<http://www.boost.org/>

Standardization

- Boost libraries commonly get picked up for standardization (often with minor changes)
- Examples:
 - `boost::regex` → `std::regex`
 - `boost::ref` → `std::ref`
 - Type Traits
 - Unordered Containers
 - `boost::filesystem` → `std::filesystem`

Nice side effect: if you are forced to use an older compiler/standard library implementation, you can use the “precursor” boost library to get a similar interface until you can upgrade.

Categories

- String and text processing
- Containers
- Iterators
- Algorithms
- Higher-order programming
- Image processing
- Input/Output
- Memory
- Patterns and Idioms
- System
- Generic Programming
- Template Metaprogramming
- Concurrent Programming
- Math and numeric
- Correctness and testing
- Data structures
- Domain Specific
- Parsing
- State Machines
- Miscellaneous

Examples

- **String and text processing**
- Containers
- Iterators
- Algorithms
- Higher-order
- Image proces
- Input/Output
- Memory
- Patterns and Idioms
- System
- Generic Programming
- Template Metaprogramming
- Programming
- eric
- d testing
- s
- c
- State Machines

E.g. **boost::format** for string formatting

- Supports formatting options similar to C-style printf
- **Type safe** and supports user types (!)
- Also supports reordering and additional format options

Example in **07_04_boost_format.cpp**

Examples

- String and text processing
- **Containers**
- Iterators
- Algorithms
- Higher-order
- Image processing
- Input/Output
- Memory
- Patterns and Idioms
- System
- Generic Programming
- Template Metaprogramming
- Programming
- Generic
- d testing
- s
- c
- State Machines

E.g. **boost::bimap** for bidirectional maps

- Works like having 2 maps which are automatically kept in sync
- Other useful container libs:
 - Circular buffer
 - Intrusive
 - ICL (interval sets)

Example in **07_05_boost_bimap.cpp**

Examples

- String and text processing
- Containers
- Iterators
- Algorithms
- Higher-order functions
- Image processing
- Input/Output
- Memory
- Patterns and Idioms
- System
- **Generic Programming**
- Template Metaprogramming
- Compiler Programming
- Generic
- and testing
- \$
- C
- State Machines

E.g. **boost::operators**

- Allows you to define some derived operators without lots of boilerplate code
- Provides fine-grained interface to define either a small set of operations or larger clusters

Example in **07_06_boost_operators.cpp**

Examples

- String and text processing
- Containers

E.g. `boost::hana`

- A template metaprogramming library
- Can work on values and types transparently
- Utility functions as well as compile-time algorithms and containers

- Patterns and Idioms
- System

- Generic Programming
- **Template Metaprogramming**
- Concurrent Programming
- Math and numeric
- Correctness and testing
- Data structures
- Domain Specific
- Parsing
- State Machines

Example in `07_07_boost_hana.cpp`

Examples

- String and text processing
 - Containers
 - Iterators
 - Algorithms
 - Higher-order programming
 - Image processing
 - **Input/Output**
 - Memory
 - Patterns and Idioms
 - System
 - Generic Programming
- E.g. **boost::program_options**
- Provides convenient interface for parsing and storing command line options
 - Many features, e.g. automatic vector aggregation
 - Good error handling, automatic help/description generation
- Parsing
 - State Machines

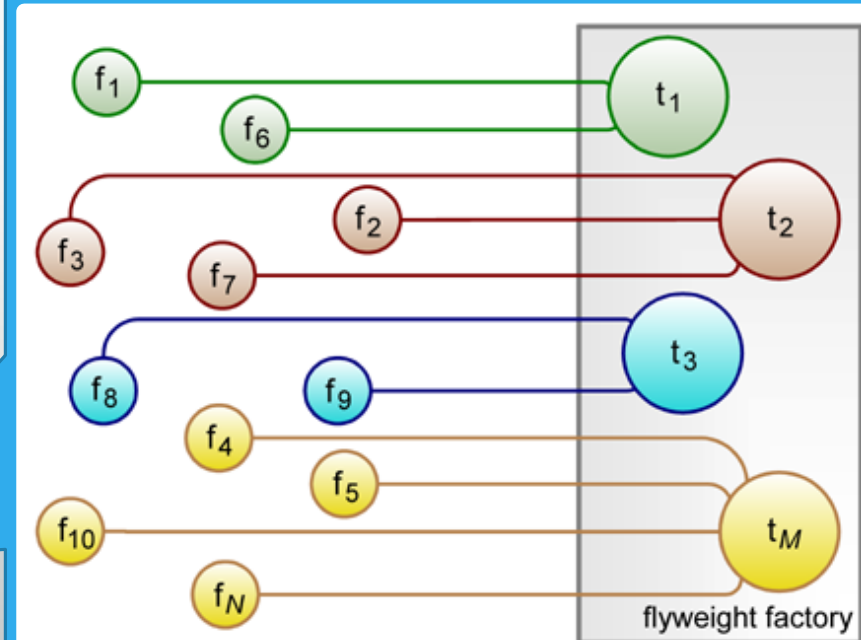
Example in **07_08_boost_program_options.cpp**

Examples

- String and text processing
- Containers
- Iterators
- Algorithms
- Higher-order programming
- Image processing
- Input/Output
- Memory
- **Patterns and Idioms**
- System

- Generic Programming

E.g. `boost::flyweight`



Examples

- String and text processing
- Containers
- Iterators

- A
- H
- In
- In
- M
- Pa
- Sy

E.g. **boost::log**

- Provides logging facilities
- Very configurable, from simple logging to console to file-based logging with custom scoped attributes
- **Note:** not header-only, needs to be built and linked

Example in **07_09_boost_log.cpp**

- Concurrent Programming
- Math and numeric
- Correctness and testing
- Data structures
- Domain Specific
- Parsing
- State Machines
- **Miscellaneous**



Other Libraries

Eigen

- *“Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms”*
- Versatile and elegant
- Easy to integrate (header only)
- Other options: Blaze, Armadillo, ...

Example in **07_10_eigen.cpp**

GUI

- **First:** think if you actually want to implement your UI in C++
- If so, the most classical choice is **Qt**
 - **Well supported and documented**, feature-rich
 - Somewhat outdated design by modern C++ standards
- *However*, if you have a good reason to write your UI in C++ you might also have a good reason to use an immediate mode GUI design



Immediate Mode UI

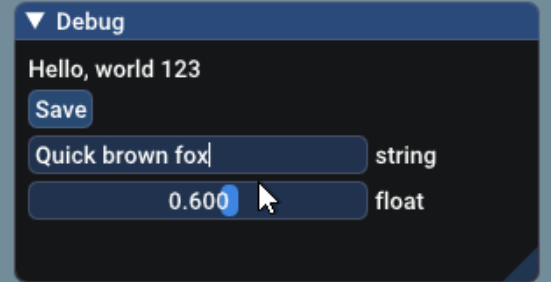
- Don't store another copy of data in UI toolkit
- Widgets are built by functions calls rather than objects
- Advantages and disadvantages compared to traditional retained mode UI
- Very suitable for custom data visualization of changing data sets
- Or for integration in existing real-time applications

→ Examples: <https://github.com/ocornut/imgui/issues/973>

C++ code

```
ImGui::Text("Hello, world %d", 123);  
  
if (ImGui::Button("OK"))  
{  
    // do stuff  
}  
  
ImGui::InputText("string", buf, 256);  
ImGui::SliderFloat("float", &f, 0.0f, 1.0f);
```

Output



<https://github.com/ocornut/imgui>



Conclusion

Summary

- C++, as a language, is designed to allow the implementation of fast, elegant and versatile libraries
- There are a large number of those out there, of varying quality and support
- Study the available technology before making an implementation decision
 - The more impactful / long-lasting the decision, the more effort you should spend on this search and selection process