Standard Library

PV264 Advanced Programming in C++

Nikola Beneš Jan Mrázek Vladimír Štill

Faculty of Informatics, Masaryk University

Autumn 2020

Standard Library

What you have seen:

- C library
- algorithms, containers, iterators
- I/O
- some of the utilities
- `unique_ptr`
- `optional`, `variant`
- thread support
- `string`, `string_view`
- ...

What other things does the standard library offer?

What are we going to see today?

- user-defined literals
- smart pointers
- `any`
- dealing with time (`<chrono>`)
- (pseudo-)random numbers (`<random>`)
- regular expressions (`<regex>`)
- filesystem library (`<filesystem>`)

User-Defined Literals

C++ literal suffixes

- integer suffixes: `u`, `l`, `ll`, `ul`, `ull`
- floating-point suffixes: `f`, `l`

Since C++11: User-defined literals

- used by standard library
- can be defined in user code (must start with `_`)

```
long double operator""_km(long double km) {
    return km * 1000.0;
}
long double operator""_miles(long double miles) {
    return miles * 1609.344;
}
```

User-Defined Literals

- `1234_x` calls first available of:
  - `operator""_x(1234ull)`
  - `operator""_x("1234")`
  - `operator""_x<'1', '2', '3', '4'>()`
- similarly for floating-point literals (`long double`)
- `"abcd"_x` calls `operator""_x("abcd", std::size_t(4))`
- `'w'_x` calls `operator""_x('w')`

User-defined literals in standard library (C++14)

- `std::complex`: `i`, `if`, `il` for pure imaginary numbers
- `std::chrono`: `s` seconds, `m` minutes, ...
- `std::string`: `"Hello"s`
- `std::string_view`: `"Hello"sv`

See http://en.cppreference.com/w/cpp/language/user_literal.

Smart Pointers

std::unique_ptr

- unique owner of memory
- std::unique_ptr< T > owns one object
- std::unique_ptr< T[] > owns an array of objects
- non-copyable, movable
- constructor does not allocate memory, it simply takes ownership of the memory given by a pointer
- std::make_unique< T >(ctor, params), std::make_unique< T[] >(size) (C++14) allocate memory (using **new**) + call the unique_ptr constructor
- destructor calls **delete** (or **delete**[] for T[] version)
  - default behaviour
  - can be changed via second template parameter

Smart Pointers — std::unique_ptr

Custom deleter

- second template parameter — type of deleter
- may be any callable object
- example usage: wrapping C library functions that allocate memory using `malloc`

```
std::unique_ptr< char, decltype(&std::free) >
    ptr{ strdup("Hello"), &std::free };
// free is called at the end of ptr's lifetime
```

Smart Pointers — std::unique_ptr

Custom deleter example — SDL2 library

```
namespace MySDL {
struct Deleter {
    void operator()(SDL_Window *w) { SDL_DestroyWindow(w); }
    void operator()(SDL_Surface *s) { SDL_FreeSurface(s); }
    // ...
};

using Window = std::unique_ptr< SDL_Window, Deleter >;
using Surface = std::unique_ptr< SDL_Surface, Deleter >;
// ...
} // namespace MySDL

int main() {
    // ...
    MySDL::Window w{ SDL_CreateWindow( ... ) };
}
```

Smart Pointers — std::unique_ptr

Custom deleter example — useful template trick

```
template< auto fn >
struct FnDeleter {
        template< typename T >
        void operator()(T* ptr) {
                fn(ptr);
        }
};
```

- can be used as:

```
std::unique_ptr< char[], FnDeleter< std::free > >
    ptr{ strdup("Hello") };
```

Smart Pointers — Reference Counting

`std::shared_ptr`

- shared ownership, counts references (`shared_ptr` instances pointing to the memory)
- *note:* reference counter has to be allocated on the heap too!
- deallocates memory when the last `shared_ptr` instance is destroyed
  - data structures must not contain `shared_ptr` cycles (use `std::weak_ptr` to break cycles)
- copyable, copy increases reference count
- `std::make_shared< T >(ctor, params)`
  - allocates memory only once (both for the `T` object and for the counter)
- should almost always be taken by value
- thread safe – reference count increments/decrements are *atomic*

Smart Pointers — Reference Counting

std::weak_ptr

- to be used with `shared_ptr` to break cycles
- does not own the memory; can detect whether the memory is still valid
    - using the counter value

```
std::weak_ptr< A > wp;
{
    std::shared_ptr< A > sp{ new A() };
    wp = sp;

    if ( auto locked = wp.lock() ) {
        locked->foo();
    }
}
if (wp.expired()) {
    std::cout << "wp has expired\n";
}
```

Smart Pointers — `std::shared_ptr`

Custom deleter

- not part of the type, simply a second argument to the constructor
- `std::shared_ptr< A > sp{ new A(), MyOwnDeleter() };`
- why is this different from `unique_ptr`?

Smart Pointers — `std::shared_ptr`

Custom deleter

- not part of the type, simply a second argument to the constructor
- `std::shared_ptr< A > sp{ new A(), MyOwnDeleter() };`
- why is this different from `unique_ptr`?
  - greater flexibility, more overhead

Polymorphic deletion

```
struct A { /* ... */ };
struct B : A { /* ... */ };

int main() {
    std::shared_ptr< A > ptr{ new B() };
}   // which destructor gets called here?
```

- again, different from `unique_ptr`

Smart Pointers — std::shared_ptr

What is wrong?

```
struct X {
    std::shared_ptr< X > getPtr() {
        return std::shared_ptr< X >( this );
    }
};
```

Smart Pointers — `std::shared_ptr`

What is wrong?

```
struct X {
    std::shared_ptr< X > getPtr() {
        return std::shared_ptr< X >( this );
    }
};
```

- can create shared pointers that do not share ownership
- object gets possibly deallocated more than once

Smart Pointers — `std::shared_ptr`

What is wrong?

```
struct X {
    std::shared_ptr< X > getPtr() {
        return std::shared_ptr< X >( this );
    }
};
```

- can create shared pointers that do not share ownership
- object gets possibly deallocated more than once

Solution: `std::enable_shared_from_this` (CRTP class)

```
struct X : std::enable_shared_from_this< X > {
    std::shared_ptr< X > getPtr() {
        return shared_from_this();
    }
};
```

Smart Pointers — Shared Ownership, Casting

`std::static_pointer_cast`, `std::dynamic_pointer_cast`, ...

- special functions to cast shared pointers
- the result has a different type but *shares ownership with the original*

Shared Ownership

- *aliasing constructor*: `std::shared_ptr< X > q(r, p);`
- shares ownership information with `r`, but points to memory of `p`
- will call deleter for the original pointer of `r`
- calling `get()` will return `p`
- programmer's responsibility: ensure `p` is valid as long as `r` lives
- example usage: `p` points to a member of the object of `r`

Container for Any Type, `std::any`

- `std::variant` can store a single value from a given list of types
  - without heap allocation
- `std::any` can store a value of any type, but with larger overhead
  - uses run-time type support (RTTI) for this
  - allocates memory (at least for larger objects)

Container for Any Type, `std::any`

- `std::variant` can store a single value from a given list of types
  - without heap allocation
- `std::any` can store a value of any type, but with larger overhead
  - uses run-time type support (RTTI) for this
  - allocates memory (at least for larger objects)
- any can be empty (`has_value()`)
- the type can be queried (`type()`)
- `std::make_any`
- `std::any_cast` for access
- no `visit` (why?)

Date and Time in the Standard Library

- C-style date and time utilities `<ctime>`
- since C++11: `std::chrono` library `<chrono>`
- three main types:
  - clocks
  - durations
  - time points
- proposed for C++2z:
  - calendar, time zones

`std::chrono::duration`

A time interval in given units (number of ticks)

```
template< class Rep, class Period = std::ratio< 1 > >
class duration;
```

- `Rep` - type for tick count (can be floating point)
- `Period` - number of seconds per tick as `std::ratio`
- stores the number of ticks
  - can be obtained through `count()`
- durations can be added and subtracted
  - type of result is a common type of both operands
- durations can be multiplied or divided by a number

Helper classes

- from `chrono::nanoseconds` to `chrono::hours`
- literals for these types (`h`, `min`, `s`, `ms`, `us`, `ns`) – C++14
  - defined in the namespace `std::chrono_literals`
- conversion between durations: `chrono::duration_cast`

std::chrono::time_point

Basically a duration linked to a given clock

```
template< class Clock,
        class Duration = typename Clock::duration >
class time_point;
```

- can be subtracted, resulting in duration of common type
- a duration can be added or subtracted

Clocks

3 predefined clocks in STL:

- `std::chrono::system_clock`
  - wall-time clock
  - can be converted to C-style time (can be displayed as datetime)
  - may not be monotonic (time can decrease)
- `std::chrono::steady_clock`
  - must be monotonic
  - may not be related to wall-clock time
- `std::chrono::high_resolution_clock`
  - may not be monotonic
  - at least as precise as `steady_clock`

Clock Interface

- clocks are types, have no instances → static methods
- each clock defines the following types:
  - `rep` – type for number of ticks
  - `period` – number of seconds per tick as `std::ratio`
  - `duration` – usually `std::chrono::duration< rep, period >`
  - `time_point` – usually `std::chrono::time_point< clock >`
- `is_steady` – static member constant, true if clock monotonic
- `now()` – static method, returns `time_point` with current time
- `system_clock` also has following static methods for conversion:
  - `std::time_t to_time_t( const time_point& )`
  - `time_point from_time_t( std::time_t )`

Random Numbers in the Standard Library

- header `<random>`
- directly in namespace `std`
- two concepts:
  - UniformRandomBitGenerator
    - generate pseudo-random unsigned integers
    - uniform distribution
    - do not use directly – source of random bits
  - RandomNumberDistribution
    - take random bits as input
    - produce numbers of given type and distribution

Random Number Engines

- a source of random bits in form of unsigned integer
  - range between `min()` and `max()` (both methods)
  - `operator()` returns next random sequence of bits
- usually a pseudo-random generator
  - seeded in constructor
- a number of predefined engines:
  - `minstd_rand`
  - `mt19937`
  - `ranlux48`
  - `...`
  - `default_random_engine` – implementation defined, usually best option
- engine adaptors
  - can change characteristics of engines
  - `discard_block_engine` – discard some of the output
  - `shuffle_order_engine` – deliver output of engine in different order

How To Seed Engines

- seeding with timestamp is usually not a great idea
  - pseudo-random number generators are deterministic
  - part of the timestamp can be guessed → limited entropy
- `std::random_device`
  - provides access to OS entropy source
  - may be a true random number generator
  - `operator()` – return random number
  - not intended for direct usage
  - usually quite slow
  - use only for seeding, not for random number generation

Distributions

- take a block of random bits and produce a number from a given distribution
- `operator()( Engine )`
  - return new random number
  - use only amortized constant number of `Engine` invocations
- number of predefined distributions:
  - `uniform_int_distribution`, `uniform_real_distribution`
  - `normal_distribution`
  - `student_t_distribution`
  - `bernoulli_distribution`
  - `binomial_distribution`
  - ...

Regular Expressions in the Standard Library

- header `<regex>`
- directly in the `std` namespace
- concept relying on:
  - a class for regular expression
  - iterators
  - algorithms
- multiple syntax options for regexes
  - modified ECMAScript = JavaScript, this is the default
  - POSIX basic, POSIX extended, AWK, (e)grep

std::basic_regex

- class representing regular expression
- `std::basic_regex` = string + matching rules (flags)
- general template:
  - `std::regex` = `std::basic_regex< char >`
  - `std::wregex` = `std::basic_regex< wchar_t >`
- 2 parametric constructors – string + optional flags. String can be
  - `std::string`
  - C-style string
  - pointer + length
  - iterators
- for available flags, see http://en.cppreference.com/w/cpp/regex/basic_regex

std::sub_match

- basically a pair of iterators of input sequence
- identifies a match
- attribute `matched` of type `bool`
- method `str()` for converting to `string`
- `std::match_results` – a collection of `std::sub_match`

std::regex_iterator

- read-only ForwardIterator
- operates on top of string iterators:
  - `sregex_iterator` = `regex_iterator< string::const_iterator >`
  - `wsregex_iterator` = `regex_iterator< wstring::const_iterator >`
  - `cregex_iterator` = `regex_iterator< const char * >`
  - `wcregex_iterator` = `regex_iterator< const wchar_t * >`
- constructors:
  - input iterators + `std::regex`
  - non-parametric constructor – end iterator
- each increment searches for next match
- dereference is of type `std::sub_match`

Regex Algorithms

For all possible prototypes, see CppReference

- `std::regex_match`
  - returns true if the whole string matches regex
  - provides matched results as `std::match_results`
    - each bracket group of the regex as a single result
- `std::regex_search`
  - similar to `regex_match`, does not have to match the whole string
  - matches depends on flags
- `std::regex_replace`
  - replace regex matches with a format string
    - `$&` – whole match
    - `$n` – n-th bracket group
    - `$$` – dollar literal

Efficiency of `std::regex`

Note: current implementations of `std::regex` are unfortunately rather slow (both at compile-time and at run-time).

There are attempts to do compile-time (and faster) regexes:
`https://www.youtube.com/watch?v=QM3W36COnE4`
(CppCon 2018: Hana Dusíková “Compile Time Regular Expressions”)

Filesystem In STL

- since C++17, the filesystem library is a part of STL
- header `<filesystem>`
- namespace `filesystem`
- originally `boost::filesystem`
  - not fully compatible with the C++17 version
- covers most used functionality
  - is portable, but not all functions are supported on every filesystem
  - e.g. FAT misses hard- and sym-links
- compile with:
  - `-lstdc++fs` when using libstdc++ (GNU)
  - `-lc++fs` when using libc++ (LLVM)

`std::filesystem::path`

- path representation
- standard syntax:
  - *root-name* if FS has multiple roots (“C:”, “//servername”) – optional
  - *root-directory mark* – makes the path absolute (e.g. “/” in POSIX), otherwise the path is relative – optional
  - zero or more of the following:
    - *file name* (including “.” and “..” to mark current and parent directory)
    - *directory separator* (default “/”); if the separator is repeated, is treated as one
- `path` also accepts the native syntax of the host OS (e.g “\” on Windows)

`std::filesystem::path`

- many useful methods:
  - `root_name`
  - `filename`
  - `stem` (filename without extension)
  - `extension`
  - `replace_extension`
  - … see CppReference for full list
- iterators for accessing elements (`begin()`, `end()`) – can be used with range-based loop
- beware of concatenation:
  - `operator+` treats paths as strings (no separators included)
  - `operator/` inserts separator between paths

Filesystem Functions

- number of functions, for full list see CppReference
- useful functions:
  - `current_path` – working directory
  - `exists` – check if path corresponds to existing FS object
  - `equivalent` – check if two paths refer to the same FS object
  - `copy` – copy a file or a directory
  - `remove`, `remove_all`
  - `temp_directory_path` – returns directory suitable for temporary files
  - ...

Filesystem Iterators

- to explore directory content, directory iterators can be used
- two types:
  - `directory_iterator` – explore content
  - `recursive_directory_iterator` – recursively explore content
- number of constructor options (follow/do not follow symlinks, etc.)

```
int main() {
    namespace fs = std::filesystem;
    fs::create_directories("example/a/b");
    std::ofstream("example/f.txt");
    for (const auto& p :
            fs::recursive_directory_iterator("example")) {
        std::cout << p << "\n";
    }
    fs::remove_all("example");
```