Modern C++ Primer

2024.03.14

SWPP Practice Session

Seunghyeon Nam

Modern C++

- `std::optional<T>`, `std::expected<T, E>`
- `std::function<T(P…)>`, lambda expressions, high-order fns
- `std::string_view, std::format`
- `decltype`, `auto`
- `std::move`, rvalue reference, `std::unique_ptr<T>`
- Concepts and requirements

`std::optional<T>`

- Concept: A container that may or may not contain an object
- The contained object is accessible via dereference operator (*)
- Trying to access an empty optional results in exception
- Supports monadic operation for concise exception handling
- More on [cppreference.com]

std::optional<T>

```cpp
std::optional<int> safe_division(const int dividend, const int divisor) {
    if (divisor == 0) {
        return std::nullopt;
    } else {
        return dividend / divisor;
    }
}
```

std::optional<T>

```cpp
std::optional<int> safe_division(const int dividend, const int divisor) {
    if (divisor == 0) {
        return std::nullopt;
    } else {
        return dividend / divisor;
    }
}
```

```cpp
SomeType construct_obj() {
    if (some_cond()) {
        auto tmp_obj = construct_true_obj();
        tmp_obj.modify();
        return tmp_obj;
    } else {
        auto tmp_obj = construct_false_obj();
        tmp_obj.modify();
        return tmp_obj;
    }
}
```

```cpp
SomeType construct_obj_with_optional() {
    std::optional<SomeType> tmp_obj;
    if (some_cond()) {
        tmp_obj = construct_true_obj();
    } else {
        tmp_obj = construct_false_obj();
    }

    tmp_obj->modify();
    return *tmp_obj;
}
```

Monadic Operations

```
std::optional<int> some_opt = 5;
std::optional<int> none_opt; // None
std::optional<int> sum_opt =
    some_opt.value_or(0) + none_opt.value_or(0); // Some(5 + 0)
auto transform_opt =
    sum_opt.transform([](const int i) { return i * 2; }); // Some(10)
auto and_then_opt = sum_opt.and_then(
    [](const int i) { return std::optional(i + 2); }); // Some(12)
auto or_else_opt =
    sum_opt.or_else([]() { return std::optional(2); }); // Some(12)
```

- Reduces redundant if-else checks
- Code is more focused on the actual computation

Monadic Operations

Monadic operations	
and_then (C++23)	returns the result of the given function on the contained value if it exists, or an empty optional otherwise (public member function)
transform (C++23)	returns an optional containing the transformed contained value if it exists, or an empty optional otherwise (public member function)
or_else (C++23)	returns the optional itself if it contains a value, or the result of the given function otherwise (public member function)

- and_then: apply T -> optional<U> if contains value
- or_else: apply () -> optional<U> if empty
- transform: apply T -> U in-place if contains value

`std::expected<T, E>`

- Concept: A container that contain an object or an error
- The contained object is accessible via dereference operator (*)
- Accessing expected as a wrong kind results in exception
- Supports monadic operation as well
- More on [cppreference.com]()

Monadic Operations

Monadic operations	
`and_then`	returns the result of the given function on the expected value if it exists; otherwise, returns the expected itself (public member function)
`transform`	returns an expected containing the transformed expected value if it exists; otherwise, returns the expected itself (public member function)
`or_else`	returns the expected itself if it contains an expected value; otherwise, returns the result of the given function on the unexpected value (public member function)
`transform_error`	returns the expected itself if it contains an expected value; otherwise, returns an expected containing the transformed unexpected value (public member function)

- `and_then`: apply `T -> expected<U, E>` if contains value
- `or_else`: apply `() -> expected<T, F>` if error

Monadic Operations

Monadic operations	
`and_then`	returns the result of the given function on the expected value if it exists; otherwise, returns the expected itself (public member function)
`transform`	returns an expected containing the transformed expected value if it exists; otherwise, returns the expected itself (public member function)
`or_else`	returns the expected itself if it contains an expected value; otherwise, returns the result of the given function on the unexpected value (public member function)
`transform_error`	returns the expected itself if it contains an expected value; otherwise, returns an expected containing the transformed unexpected value (public member function)

- `transform`: apply `T -> U` in-place if contains value
- `transform_error`: apply `E -> F` in-place if error

`std::function<T(Ts...)>`

- Concept: A function object
  - Functions can be tossed around like ordinary objects!
  - Includes lambda expressions
- Often used in higher-order functions (next slides)
- More on [cppreference.com]()

`std::transform()`

- Concept: Apply a function to every element in the iteration
- Accepts a transformer function and input/output iterator
  - Function type should be `InputT -> OutputT`
  - Using constant input iterator is a good practice
  - Input/output iterators should not overlap
- More on [cppreference.com]

`std::accumulate()`

- Concept: Apply a function to every element in the iteration
- Accepts an accumulator function, init value and input iterator
    - Accumulator function type should be `(outputT, inputT) -> outputT`
    - Using constant input iterator is a good practice
- More on cppreference.com

Lambda Expression

- Concept: Defining a function to use in a very narrow scope
  - A function that will be used once and never don't really need a name
- Weird syntax!
  - `[captures](args) { definition }`
- Context can be 'captured' when creating a lambda.
- More on cppreference.com

`std::string_view`

- Concept: A read-only reference to part of the string
- Make a substring without copying the contents
- `string` must not be modified or deleted while view is alive
  - Dangling reference!
- More on cppreference.com

Keyword `decltype`

- Concept: Type of an object
- Useful when hiding a very long typename
  - In large codebase, typenames can get extremely long…
- More on cppreference.com

Keyword `auto`

- Concept: Deduce the type from the RHS expression
  - You can’t use auto when there’s no RHS to deduce type from!
  - Notable exception is lambda’s arguments
- Useful when hiding a very long typename
  - Using auto is enough for most of the cases
- More on cppreference.com

Thinking of Pointers…

- Pointers serve many purposes
  - Simply points to an object or an array
  - Owns a dynamically allocated object or array
  - ... Which are possibly nonexistent (nullable type)

Thinking of Pointers…

- Pointers serve too many purposes!
  - One cannot tell if a pointer is an array or a single object
  - One cannot tell if a pointer can be (or should be) freed
  - Tedious null-check codes must be added everywhere

Thinking of Pointers…

- Solution: Offer alternatives for each use case of pointers
  - Raw pointers should only be used to point to a non-owning object
  - Use spans to refer to a non-owning array
  - Use smart pointers to manage an owning object or array
  - Use optional to make types nullable

`std::unique_ptr<T>`

- Concept: Exclusive ownership of an object
- Copying is forbidden
  - You have to `std::move()` the `unique_ptr` to transfer the ownership
  - Or you can only take the reference of the contained object
- Usually created using `std::make_unique()`

`std::unique_ptr<T>`

- Concept: Automated resource management
- When `unique_ptr` gets out of scope, the contained object is automatically deleted
  - No more leaking memories you forgot to `delete`!
- More on cppreference.com
- See also: RAII, shared_ptr⟨T⟩

`std::unique_ptr<T>`

- Smart pointers allow nullptr
    - Always check for null when using smart pointers
- Nothing stops you from manually deleting the managed pointer
    - As enforcing it may break the legacy C++ codes...

Problems with Copying

- Assigning from one variable to another is done via copying
- But copying can be extremely costly
  - A string of 1M+ characters
  - A vector of a very large struct with more than 100 member variables
- Don’t copy unless you really need a separate copy!

Implicit Copy

- Detecting the copy operations in the code is very hard
  - At least one assignment, construction, or function call in every LOC
- Missing a single copy can open up a ‘copy hell’
  - Overhead due to repetitive or recursive copying
- Can be a potential performance bottleneck

Deleted Operations

- You can forbid copying the objects
  - To enforce explicit ownership, prevent implicit copy, etc
  - Construction: `T(const T&) = delete;`
  - Assignment: `T& operator=(const T&) = delete;`
- Most LLVM API types forbid copying
  - You should rely on reference or pointers for most of the times

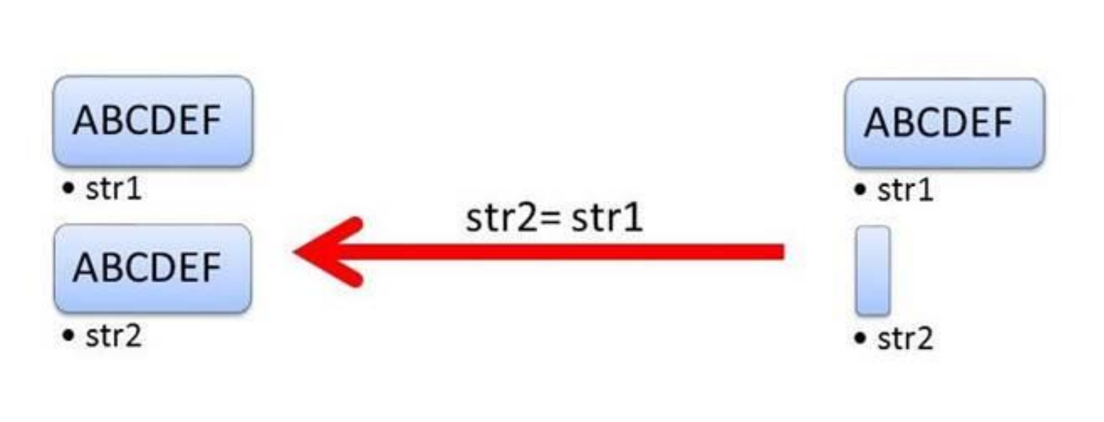
Deleted Operations

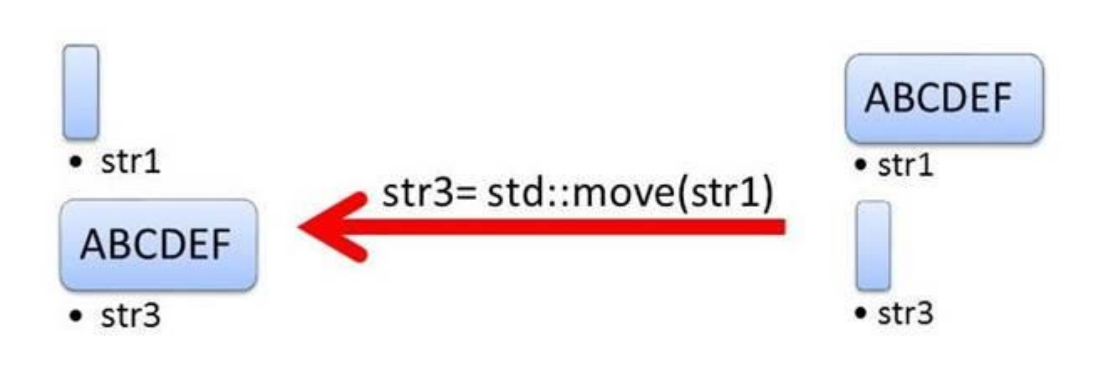
- Programming without assignment is virtually impossible
  - You cannot store the return value of a function
  - You cannot alias the variables for readability
- What if there's a way to assign a value without copying?

Move Semantics

- Concept: Transfer the ownership of data instead of copying
- Moving is cheaper than copying in most of the cases
  - Move operations should be implemented for actual benefit
  - If not, the behavior defaults to copying

Move Semantics





Move Semantics

- There are many C++ idioms you should know in order to implement your own move operations
- Following slides will be hard to understand at the first glance
- Understanding only the colored sentences should be sufficient for most of the cases
  - You can always look more into the reference page though

rvalue Reference (&&)

- Concept: An object that might be moved instead of copying
- Despite the syntax, it is not reference of reference
- Used to distinguish when to copy and when to move
  - Move operations are specialization for rvalue operand(s)
  - If operations are not specialized, your object will be copied
- More on cppreference.com (warning: very technical)

`std::move()`

- Concept: Cast an object into an rvalue reference
- The name is terribly misleading
  - It does not actually move your object
  - Object won't be 'moved' unless move operations are defined
- More on cppreference.com

Implementing Move Operations

- Implement move constructor
  - `T(T&& other)`
- Implement move assignment operator
  - `T& operator=(T&& other)`
- Copy constructor and copy AOp will be automatically deleted
  - You can manually re-implement them if you want to

Implementing Move Operations

- Simply copy pointer or integral types
  - Copying such small types have negligible overhead
- Use `std::move()` for standard library types
  - Most of them have well-implemented move operations
- Heap-allocate large structs or classes using `unique_ptr`
  - Moving only the `unique_ptr` can be cheaper

Implementing Move Operations

- Or just use the default implementation!
  - `T(T&& other) = default;`
  - `T& operator=(T&& other) = default;`
- Default implementation will ‘try to’ move every member
  - If you have a lot of member variables, use the `unique_ptr` trick

Implementing Move Operations

- The state of 'moved from' object is unspecified
  - Unspecified means the behavior depends on implementation
  - Each type may show different behavior or state
  - Behaviors of standard library types are specified in the reference
- It is your responsibility to correctly implement your type's behavior after the move

Thinking of Templates…

- Templates serve many purposes
  - Allows reusing same code for various types
  - Allows additional optimization for selected types
- But which specialization should we use for some type?
  - This is not an easy choice, due to the nature of C++

Thinking of Templates…

- Template copy–pastes the entire code but types
  - Technically not much better than macros
  - Template substitution error results in enormous amount of message
  - “I tried copy–pasting every possible type but it won’t compile”

```
template <typename T> void print(const T &obj) noexcept {
  std::cout << obj << "\n";
}

int main() {
  print(std::set<int>{1, 3, 5});
  return 0;
}
```

Concepts to Rescue!

- A named constraint that limits the template type candidates
  - Compiler first checks if the candidate type satisfies the concept
  - Improves compile time and reduces error message

Concepts to Rescue!

```cpp
#include <iostream>
#include <set>

template<typename T>
concept Printable = requires (T a){
    std::cout << a; // is cout defined?
};

template<typename T> requires Printable<T>
T print(const T& obj) noexcept {
    std::cout << obj << "\n";
}
```

```cpp
int main() {
  print(std::set<int>{1, 3, 5});
  return 0;
}
```

Thinking of Headers…

- Headers have been traditionally used as library interface
- Headers are `#include` d
  - This is just a fancy way of copy–pasting!
  - Improper copy–pasting may cause problems

Thinking of Headers…

- ‘Copied’ code may affect codes included or written below
  - using namespace
  - Multiple definition of function
- Copying cause code bloat & slows down compilation
  - Header guards are a must for each header
  - And yet ‘Hello, world’ becomes 700+kB just by including `<iostream>`

Modules to Rescue!

- A pre-compiled interface for library
  - Compiler creates a binary interface for each module
  - Compiler can quickly locate the declaration & definition in this binary
  - Programmer can control visibility of each library component
  - This improves compile time and reduces potential errors

Modules to Rescue!

```cpp
#ifndef SWPP_TEST_H
#define SWPP_TEST_H

#include <print>
#include <span>

namespace test {
void print_value(const int value);
void print_span(const std::span<int> arr);
} // namespace test

#endif // SWPP_TEST_H
```

```cpp
module;
#include <print>
#include <span>

export module test;

namespace test {
void print_value(const int value);
export void print_span(const std::span<int> arr) {
  for (const auto i : arr) {
    print_value(i);
  }
}
} // namespace test
```

Writing Module Interface

- We write module interface instead of headers
- `export <module_name>;` declares a new module
- Each function and type can be individually exported
  - Only exported entities are visible to importers
- Writing definition in module interface do not break ODR

Writing Module Implementation Unit

- We can hide the actual definition behind the interface
- `<module_name>;` denotes 'module implementation unit'
- The actual definition of module entities can come here
  - Just like hiding definitions inside the cpp file with headers!

Using Module

- `import <module_name>` is all it takes
- Every exported entities in the module will now be available
- However, entities are thrown into global namespace by default
    - It is a good practice to always contain module inside namespaces

Using Headers with Modules

- You still need headers to use legacy functions
- `module;` at the beginning denotes ‘global module fragment’
- Headers can only reside inside global module fragment
  - Only exported entities are visible to importers
- Writing definition in module interface do not break ODR

Use the Latest clangd

- `clangd` is a C++ linter & helper (aka. language server)
- You need the latest clangd to use the latest language features
  - install–llvm.sh installs it clangd as well
  - Specify the path to your clangd extension to use it on vscode