

Modern C++ Primer

2023.03.16

SWPP Practice Session

Seunghyeon Nam

Modern C++

- `std::optional<T>`, `std::variant<T, Ts...>`, `std::string_view`
- `std::function<T(P...)>`, lambda expressions
- `std::transform`, `std::accumulate`
- `decltype`, `auto`
- `std::move`, rvalue reference, `std::unique_ptr<T>`

`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
- More on cppreference.com

`std::variant<T, Ts...>`

- Concept: A container that contains **one of the specified types**
- Can specify any number of types without duplicates
 - Except zero, of course
- Helper functions such as `std::get<T>` or `std::visit()`
- Trying to access as a different type results in exception
- 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

`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 `outputT(inputT)`
 - 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(outputT, inputT)`
 - 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

Keyword `decltype`

- Concept: Type of an object
- Useful when hiding a very long typename
 - In large codebase, typenamees 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

`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>](#)

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

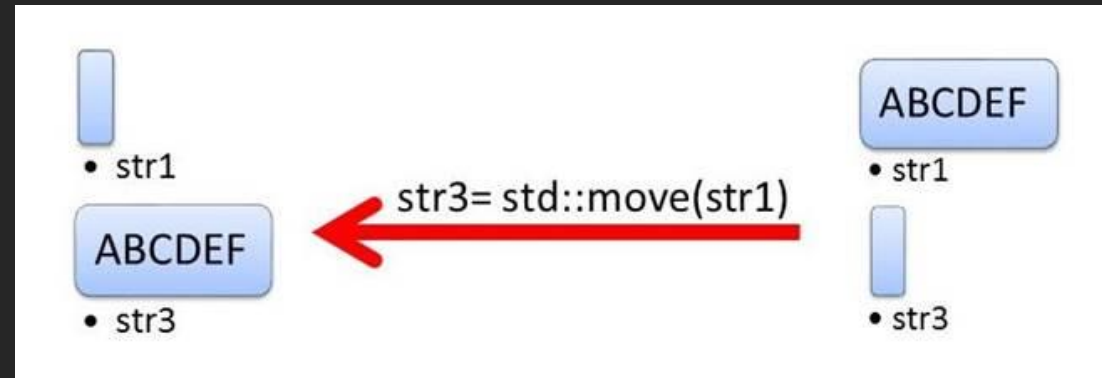
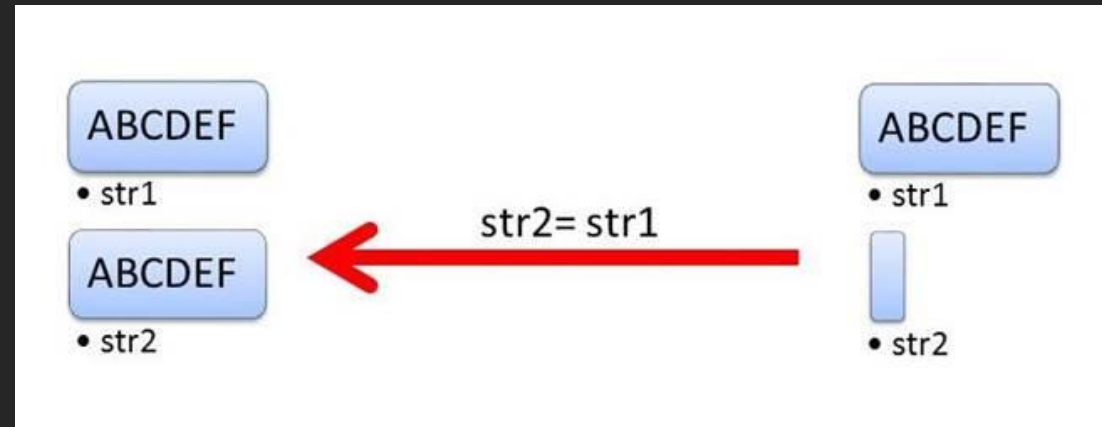
Deleted Operations

- 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