CS100 Lecture 27

Other Facilities in the Standard Library

Contents

- C++17 library facilities
  - `function`
  - `optional`
  - `string_view`
  - `pair` and `tuple`
- Going into C++20:
  - Ranges library
  - Format and print library
  - Future

C++17 library facilities

`function`

Defined in `<functional>`

`std::function<Ret(Args...)>` is a general-purpose function wrapper that stores any callable object that can be called with arguments of types `Args...` and returns `Ret`.

```
Polynomial poly({3, 2, 1}); // `Polynomial` in homework 5
std::function<double(double)> f1(poly);
std::cout << f1(0) << '\n';

std::function<void()> f2 = []() { std::cout << 42 << '\n'; };
f2(); // prints 42
```

Recap: callable

A callable object in C++ might be a function, a pointer-to-function, or an object of class type that has an overloaded `operator()` [1].

- Lambdas belong to the last category, whose type is compiler-generated.

A function has an address! When the program is executed, the program instructions (machine code) are loaded into the memory.

```
int add(int a, int b) { return a + b; }
int main() {
  auto *padd = &add;
  std::cout << (*padd)(3, 4) << '\n';
  std::cout << padd(3, 4) << '\n'; // Also correct.
}
```

A pointer-to-function itself is also callable. `pfunc(...)` is the same as `(*pfunc)(...)`.

Example: Calculator

A more fancy way of implementing a calculator:

```
std::map<char, std::function<double(double, double)>> funcMap{
  {'+', std::plus<>{}},
  {'-', std::minus<>{}},
  {'*', std::multiplies<>{}},
  {'/', std::divides<>{}}
};
double lhs, rhs; char op;
std::cin >> lhs >> op >> rhs;
std::cout << funcMap[op](lhs, rhs) << '\n';
```

`std::plus`, `std::minus`, etc. are defined in the standard library header `<functional>`.

Example: Calculator

Combining different ways of using `std::function`:

```
double add(double a, double b) { return a + b; }
struct Divides {
  double operator/(double a, double b) const { return a / b; }
};
int main() {
  std::map<char, std::function<double(double, double)>> funcMap{
    {'+', add}, // A function (in fact, a pointer-to-function)
    {'-', std::minus<>{}}, // An object of type `std::minus<>`
    {'*', [](double a, double b) { return a * b; }}, // A lambda
    {'/', Divides{}} // An object of type `Divides`
  };
  double lhs, rhs; char op;
  std::cin >> lhs >> op >> rhs;
  std::cout << funcMap[op](lhs, rhs) << '\n';
}
```

`optional`

Defined in the header `<optional>`.

`std::optional<T>` manages **either an object of type `T`, or nothing**.

- Algebraically: Let $\mathcal{T}$ be the value set of `T`, and let $\mathcal{O}$ be the value set of `std::optional<T>`. We have

$$\mathcal{O} = \mathcal{T} \cup \{\text{std::nullopt}\},$$

  where `std::nullopt` is a special object that represents the state of ***nothing***.

Example: Solving quadratic equation in $\mathbb{R}$.

A typical example: Use `std::optional<Solution>` when there may be no solutions.

```
std::optional<std::pair<double, double>> solve(double a, double b, double c) {
  auto delta = b * b - 4 * a * c;
  if (delta < 0)
    return std::nullopt; // No solution.
  auto sqrtDelta = std::sqrt(delta);
  // An `std::optional<T>` can be initialized directly from `T`.
  return std::pair{(-b - sqrtDelta) / (2 * a), (-b + sqrtDelta) / (2 * a)};
}
```

Example: Solving quadratic equation in $\mathbb{R}$.

```cpp
void printSolution(const std::optional<std::pair<double, double>> &sln) {
  if (sln) { // conversion to bool tests whether it contains an object
    auto [x1, x2] = sln.value(); // .value() returns the contained object.
    std::cout << "The solutions are " << x1 << " and " << x2 << '.'
              << std::endl;
  } else
    std::cout << "No solutions." << std::endl;
}
int main() {
  auto sln1 = solve(1, -2, -3);
  printSolution(sln1);
  auto sln2 = solve(1, 0, 1);
  printSolution(sln2);
  return 0;
}
```

How will you implement `std::optional<T>`?

Is this good?

```cpp
template <typename T>
struct Optional {
  T object;
  bool hasObject;
  // ...
};
```

How will you implement `std::optional<T>`?

Is this good?

```
template <typename T>
struct Optional {
  T object;
  bool hasObject;
  // ...
};
```

NO! It models $\mathcal{O} = \mathcal{T} \times \{\text{true}, \text{false}\}$. The `object` is alive even when `hasObject` is `false`!

- This also requires the "nothing" state to be represented by default-initializing `object`, but the default-initialization of `T` may be expensive or disabled!

How will you implement `std::optional<T>`?

Is this good?

```
template <typename T>
struct Optional {
  std::unique_ptr<T> pObject; // "Nothing" is represented by nullptr.
  // ...
};
```

How will you implement `std::optional<T>`?

Is this good?

```cpp
template <typename T>
struct Optional {
  std::unique_ptr<T> pObject; // "Nothing" is represented by nullptr.
  // ...
};
```

It does model $\mathcal{O} = \mathcal{T} \cup \{\text{std::nullopt}\}$, but it requires dynamic memory allocation.

If I just need something to represent "no solution", why would I have to store the solution on dynamic memory?

- Such overhead is not acceptable!

An `std::optional` models an **object**, not a pointer!

How will you implement `std::optional<T>`?

The implementation is not trivial. See this page if you are interested.

- It requires careful treatment of memory, possibly using a `union`.

Other member functions of `std::optional`

Some common ones:

- `*o`: returns the stored object. The behavior is undefined if it does not contain one.
- `o->mem`: equivalent to `(*o).mem`.

  `std::optional<T>` **does not model a pointer**, although it provides `*` and `->`.
- `o.value_or(x)`: returns the stored object, or `x` if it does not contain one.
- `o1.swap(o2)`
- `o.reset()`: destroys any contained object
- `o.emplace(args...)`: constructs the contained object in-place.

Refer to cppreference for a full list.

`string_view`

The old question: How do you pass a string?

```cpp
void some_operation(const std::string &str) {
  // ...
}
```

Pass-by-reference-to-`const` seems to be quite good: It accepts both lvalues and rvalues, whether `const`-qualified or not, and avoids copy.

- Wait ... Does it really avoid copy?

`string_view`

The old question: How do you pass a string?

```
void some_operation(const std::string &str) {
  // ...
}
```

```
std::string s = something();
some_operation(s); // Copy is avoided, of course.
some_operation("The quick red fox jumps over the slow red turtle."); // Ooops!
```

- When we pass a string literal, a temporary `std::string` is created first, during which the content of the string is still copied!

`string_view`

What do `char[N]`, `"hello"`, `std::string`, `str = new char[N]{...}` have in common?

`string_view`

What do `char[N]`, `"hello"`, `std::string`, `str = new char[N]{...}` have in common?

- A pointer to the first position, and a length!

```
struct StringView {
  const char *start;
  std::size_t length;

  StringView(const char *cstr) : start{cstr}, length{std::strlen(cstr)} {}
  StringView(const std::string &str) : start{str.data()}, length{str.size()} {}

  std::size_t size() const { return length; }
  const char &operator[](std::size_t n) const { return start[n]; }

  // ...
};
```

`string_view`

Defined in header <string_view>.

`std::string_view`: An **non-owning** *reference* to a string. It is often used to refer to a string that we don't modify.

```
// `std::string_view` is usually passed by value directly,
// since it is light-weighted and models a "pointer".
void some_operation(std::string_view str);
int main() {
  std::string s1 = something(), s2 = something_else();
  some_operation(s1);
  some_operation(s1 + s2);
  some_operation("hello");
}
```

- No copy is performed, even for `"hello"`.

Avoid dangling `string_view`!

Let's use `std::string_view` everywhere, shall we?

```cpp
struct Student {
  std::string_view name;
  // ...

  Student(std::string_view name_) : name{name_} {}
};

int main() {
  std::string s1 = something(), s2 = something_else();
  Student stu(s1 + s2);
  std::cout << stu.name << '\n'; // Undefined behavior!
}
```

Avoid dangling `string_view`!

Let's use `std::string_view` everywhere, shall we?

```cpp
struct Student {
  std::string_view name;
  // ...

  Student(std::string_view name_) : name{name_} {}
};

int main() {
  std::string s1 = something(), s2 = something_else();
  Student stu(s1 + s2); // `s1 + s2` is a temporary!
  std::cout << stu.name << '\n'; // Undefined behavior! `stu.name` is dangling!
}
```

`stu.name` refers to a **temporary** created by `s1 + s2`! It is destroyed immediately when the initialization of `stu` ends.

Avoid dangling `string_view`!

The same thing happens if you try to use reference-to-`const` as a member:

```
struct Student {
  const std::string &name;
  // ...

  Student(const std::string &name_) : name{name_} {}
};

int main() {
  std::string s1 = something(), s2 = something_else();
  Student stu(s1 + s2); // `s1 + s2` is a temporary!
  std::cout << stu.name << '\n'; // Undefined behavior! `stu.name` is dangling!
}
```

`string_view`

Using a `string_view` parameter can accept strings of any form, and avoid copy.

- The use of `string_view` as a parameter is often safe, because the lifetime of the argument should be longer than the execution of the function.
- In other cases, be extremely careful to avoid dangling `string_view`s!

`pair` and `tuple`

`pair` and `tuple` can be thought of as a "quick and dirty" data structure.

- `std::pair<T, U>`: defined in `<utility>`. It models

$$\mathcal{T} \times \mathcal{U} = \{(t, u) \mid t \in \mathcal{T}, u \in \mathcal{U}\}.$$

- `std::tuple<T1, T2, ...>`: defined in `<tuple>`. It models

$$\mathcal{T}_1 \times \mathcal{T}_2 \times \cdots \times \mathcal{T}_n,$$

  where n is compile-time known non-negative constant integer.

`pair` and `tuple`

`std::pair<T, U>` is defined almost just like this:

```
template <typename T, typename U>
struct pair {
  T first;
  U second;
};
```

It comes from C++98. At that time, there was no **variadic templates** which is necessary for building a `tuple`.

`std::tuple<Types...>` is an extension of `std::pair<T1, T2>`, which can contain an arbitrary number of things.

`pair` and `tuple` in modern C++

With the increasing support for **aggregates** and **structured binding** in modern C++, `pair` and `tuple` are seldom needed now.

A user-defined type can also be used conveniently:

```
template <typename T> struct Set {
  struct InsertResult {
    bool success;
    Iterator position;
  };
  InsertResult insert(const T &);
};

// structured binding
auto [ok, pos] = mySet.insert(something);
if (ok)
  do_something(pos);
```

`pair` and `tuple` in modern C++

Which one do you prefer?

```cpp
template <typename T> struct Set {
  struct InsertResult {
    bool success;
    Iterator position;
  };
  InsertResult insert(const T &);
};

auto result = mySet.insert(x);
if (result.success)
  do_something(result.position);
```

```cpp
template <typename T> struct Set {
  std::pair<bool, Iterator>
  insert(const T &);
};

auto result = mySet.insert(x);
if (result.first)
  do_something(result.second);
```

[Best practice] Prefer a self-defined type with meaningfully named members to `pair` and `tuple`.

Notes

[1] A **pointer-to-member** is also a callable.