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.

```cpp
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:

```cpp
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.