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
  - Formatting 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}); // f(x) = 3x^2 + 2x + 1
std::function<double(double)> f1(poly);
std::cout << f1(0) << '\n'; // f(0)

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

Recap: callable

A callable object in C++ might be a pointer-to-function, a fucntion object (an object of class type that overloads `operator()`), or a lambda expression.

A function has an address! When the program is executed, the function's instructions 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<double>()},
  {'-', std::minus<double>()},
  {'*', std::multiplies<double>()},
  {'/', std::divides<double>()}
}; // Operator character => The wrapper of the corresponding function object

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`:

```cpp
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<double>()}, // An object of type `std::minus<>`
    {'*', [](double a, double b) { return a * b; }}, // A lambda
    {'/', Divides()} // An object of type `Divides`
  }; // Operator character => The wrapper of the corresponding callable object

  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**.

- 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.

```
// Solve the quadratic equation: ax^2 + bx + c = 0.
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 for testing 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;
}
```

Other member functions of `optional`

Some common ones:

- `*o` : return 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)` : return the stored object, or `x` if it does not contain one.
- `o1.swap(o2)` : swap the stored objects of `o1` and `o2` .
- `o.reset()` : destroy any stored object.
- `o.emplace(args...)` : construct the stored 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 a `char[N]`, `"hello"`, a `std::string`, `new char[N]{...}` have in common?

`string_view`

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

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

Define a class `StringView` to represent all of them:

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

public:
  // Initialize from a string without copying it.
  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 the header `<string_view>`.

`std::string_view`: Provide **read-only** access to an existing string **without copying it**. "Read-only" means that we can only *view* the string without modifying it.

- A `std:string` owns a copy of an existing string.

```
// A `std::string_view` is usually passed by value directly,
// since it is lightweight.
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!
}
```

Avoid dangling `string_view`!

```cpp
class Student {
  std::string_view name;
  // ...
public:
  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`!

```cpp
class Student {
  std::string_view name;
  // ...
public:
  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:

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

[Best practice] Prefer `std::string_view` over `std:string` for a read-only string.

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

- The use of a `std::string_view` as a function 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 `std::string_view`s!

`pair` and `tuple`

`std::pair` and `std::tuple` are handy data structures.

- `std::pair<T, U>` (defined in `<utility>`): hold a pair of elements, whose types are `T` and `U`, respectively.
- `std::tuple<T1, T2, ..., Tn>` (defined in `<tuple>`): hold multiple elements, whose types are `T1`, `T2`, ..., `Tn`, respectively.

`std::pair` is a special case of `std::tuple` with only two elements.

`pair` and `tuple`

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

```cpp
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 `std::tuple`.

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

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

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

A user-defined type can also be used conveniently:

```cpp
template <typename T>
struct Set {
  struct InsertResult { // An aggregate class with two public data members
    bool success;       // for packing the two returned values of `insert`.
    Iterator position;
  };
  InsertResult insert(const T &);
};
// Structured binding for unpacking the two returned values.
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 `std::pair` and `std::tuple`.

Others

Other things in the C++17 standard library we have not touched:

- `<regex>`: Standard library support for **regular expressions**.
- `<filesystem>`: Standard library support for **file system operations**.
- Concurrency support: `<thread>`, `<atomic>`, `<mutex>`, ...

Going into C++20

C++20 is historic!

CppCon2021 Talk by Bjarne Stroustrup: C++20: Reaching the aims of C++

C++20 is the first C++ standard that delivers on virtually all the features that Bjarne Stroustrup dreamed of in *The Design and Evolution of C++* in 1994.

- Coroutines (Talk)
- Concepts and requirements (`concept`, `requires`) (Talk)
- Modules (Talk) (Talk on the implementation by MSVC)
- Ranges library
- Formatting library
- Three-way comparison (`operator<=>`, `std::partial_ordering`, ...)

Ranges library: The next generation of STL.

An extension and generalization of the algorithms and iterator libraries that makes them more easy to use and powerful.

A range is represented by one object, instead of `(begin, end)` or `(begin, n)`.

Ranges library: C++20 constrained algorithms

Given `Student` defined as

```
struct Student { std::string name; int id; };
```

C++17:

```
void sortStudentsByID(std::vector<Student> &students) {
  std::sort(students.begin(), students.end(),
            [](const Student &a, const Student &b) { return a.id < b.id; });
}
```

C++20:

```
void sortStudentsByID(std::vector<Student> &students) {
  // The container itself is a range. `&Student::id` is a pointer-to-member.
  std::ranges::sort(students, {}, &Student::id);
}
```

Ranges library: Operations are composable

Enumerate the first 10 even numbers in a vector in reverse order:

```
using namespace std::views;
for (auto x : vec | filter([](auto x) { return x % 2 == 0; }) | take(10)
              | reverse)
  do_something(x);
```

It looks very much like the Linux pipes: The following Linux shell command will list all the installed packages related with LaTeX, sort them, and display the first five lines.

```
apt list --installed | grep latex | sort | head --lines 5
```

Formatting library

Some may think that

```cpp
printf("%d + %d == %d\n", a, b, a + b);
```

is better than

```cpp
std::cout << a << " + " << b << " == " << a + b << '\n';
```

However, the existing `printf` family of functions have some drawbacks:

- Not type-safe. Specifying type information in the format string is error-prone. Mismatch bewteen the specified types and the acutal types of arguments to output results in undefined behavior.
- Not extensible. We cannot use `printf` to print objects of our self-defined types.

Formatting library

The text formatting library offers a safe and extensible alternative to the `printf` family of functions. It is intended to complement the existing C++ I/O streams library.

```
std::cout << std::format("{} + {} == {}.\n", a, b, a + b);
```

The C++23 `print` library

With the C++23 `print`, we can print a formatted string directly:

```
std::print("{} + {} == {}.\n", a, b, a + b);
```

Furthermore, the C++23 `print` is able to handle Unicode! The following should never produce garbled characters.

```
std::print("你好，世界！");
```

Future

- The graph library (Talk) (P1709R3), which may be in C++26?
- Standard library support for linear algebra algorithms: `<linalg>` in C++26.
- Debugging support: `<debugging>` in C++26.
- ……

Summary

- `function`: A function wrapper that stores a callable object.
- `optional`: Either an object or nothing.
- `string_view`: Offer read-only access to a string of any form without copying, which is often used as a function parameter.
- `pair` and `tuple`: Data structures, which should be seldomly used in modern C++.