CS100 Recitation 9

Contents

`const` Members

The Reason Why We need `const` Members

Please consider the following code:

```
class A {
public:
    int get() {return aaa_;}
private:
    int aaa_ = 33;

};

int main() {
    const A a;
    std::cout << a.get() << std::endl;
    return 0;
}
```

`const` Members

1. On a `const` object (including when using reference-to-`const`), only `const` member functions can be called.
2. **The `const` qualifier propagates during member access**:
   - If the type of `ptr` is `const T *`, and `T` has a data member `member` of type `U`, then the type of `ptr->member` is `const U`.
3. Inside a `const` member function:
   - Modifying its own data members is not allowed.
   - Calling its own non-`const` member functions is not allowed.
4. In the `const` member function of class `X`, the type of `this` is `const X *`.
   - In non-`const` member functions, the type of `this` is `X *`.

Overloaded Functions with `const`

```cpp
class Dynarray {
public:
  const int &at(std::size_t n) const {
    return m_storage[n];
  }
  int &at(std::size_t n) {
    return m_storage[n];
  }
};
arr.at(i) = 42;
```

Codes on the left would be "equivalent" to:

```cpp
const int &at(const Dynarray *this,
              std::size_t n) {
  return this->m_storage[n];
}
int &at(Dynarray *this, std::size_t n){
  return this->m_storage[n];
}
at(&arr, i) = 42;
```

Overloaded Functions with `const`

- For a `const` object:
  - It can only call the `const` version of a function.
  - The result is a reference-to-`const`, so modification is not allowed.
- For a non-`const` object:
  - Both `const` and non-`const` versions can be called.
  - Calling the non-`const` version is a perfect match.
  - Calling the `const` version involves adding **low-level `const`**, making it a less preferred match.

Avoiding Duplication in `const` vs non-`const` Overloads

- Writing identical code twice is cumbersome and may case bugs easily. Is there a way to avoid repetition?
  - C++23 deducing-this can help.
- If we don't have additional syntax features, can one function call the other?

`const` Invokes non-`const`

- Use `static_cast<const Dynarray *>(this)` to add low-level `const` to `this`.
- Then call `->at(n)`, which will match the `const` version of `at`.
- Remove the low-level `const` of the returned `const int &` using `const_cast`.

```cpp
class Dynarray {
public:
  const int &at(std::size_t n) const {
    return m_storage[n];
  }
  int &at(std::size_t n) {
    return const_cast<int &>
    (static_cast<const Dynarray *>(this)->at(n));
  }
};
```

non-`const` Invokes `const`?

Can we reverse it, letting the `const` version call the non-`const` version?

```
class Dynarray {
public:
  int &at(std::size_t n) {
    return m_storage[n];
  }
  const int &at(std::size_t n) const {
    return const_cast<Dynarray *>(this)->at(n);
  }
};
```

non-`const` Invokes `const`?

- **No**! A `const` member function is guaranteed **not to modify** the object's state, but a non-`const` member function makes no such promise!
- If the non-`const` version unintentionally modifies the object's state, calling it from the `const` version could lead to disaster.
- Compare the use of the "risky" `const_cast` in the two methods:
  - "Add first, then remove" (先添加，再去除): Safe.
  - "Remove first, then add" (先去除，再添加): Risky.

Type Alias Members

Type Alias Members

A class can have **type alias members**.

```
class Dynarray {
 public:
  using size_type = std::size_t;
  size_type size() const { return m_length; }
};
```

Usage: `ClassName::TypeAliasName`

```
for (Dynarray::size_type i = 0; i != a.size(); ++i)
  // ...
```

Note: Here we use `ClassName::` instead of `object`, because such members belong to the **class**, not one single object.

Type Alias Members In the Standard Library

All standard library containers (and `std::string`) define the type alias member `size_type` as the return type of `.size()`:

```
std::string::size_type i = s.size();
std::vector<int>::size_type j = v.size();
std::list<int>::size_type k = l.size();
```

- This type is **container-dependent**: Different containers may choose different types suitable for representing sizes.
- Define `Container::size_type` to achieve good **consistency** and **generality**.

Type Alias Members In Practice

It is common in large projects that we need to have self-designed data types.

```cpp
// These are very common:

using Age = unsigned int;
using Name = std::string;
using ID = int;
```

`static` Members

`static` Data Members

```cpp
class A {
public:
  static int something;
  // other members ...
};

A a1, a2;
// a1.something;
// a2.something;
```

- There is **only one** `A::something`: it does not belong to any object of `A`. It belongs to the class `A`.
- `a1.something` and `a2.something` refer to the same variable.
- This is widely seen in Python.

`static` Member Functions

A `static` member function:

```
class A {
 public:
  static void fun(int x, int y);
};
```

- `A::fun` (its name is in the class scope: `A::fun(x, y)`) does not belong to any object of `A`. It belongs to the **class** `A`. And there is no `this` pointer inside `fun`.
- It can also be called by `a.fun(x, y)` (where `a` is an object of type `A`), but here `a` will not be bound to a `this` pointer, and `fun` has no way of accessing any non-`static` member of `a`.

Move Operations

Please refer to the course slides, they are good.

Review: Rvalue References

A kind of reference that is bound to **rvalues**:

```
int &r = 42;                // Error: Lvalue reference cannot be bound to rvalue.
int &&rr = 42;              // Correct: `rr` is an rvalue reference.
const int &cr = 42;         // Also correct:
                            // Lvalue reference-to-const can be bound to rvalue.
const int &&crr = 42;       // Correct, but useless:
                            // Rvalue reference-to-const is seldom used.

int i = 42;
int &&rr2 = i;              // Error: Rvalue reference cannot be bound to lvalue.
int &r2 = i * 42;           // Error: Lvalue reference cannot be bound to rvalue.
const int &cr2 = i * 42; // Correct
int &&rr3 = i * 42;         // Correct
```

- Lvalue references (to non-`const`) can only be bound to lvalues.
- Rvalue references can only be bound to rvalues.

Lvalues/Rvalues and Copy/Move

Suppose `x` owns certain resources. There are two ways to access its resources:

- Copy: Duplicate its resources.
- Move: Take ownership of its resources.

Lvalues are persistent, Rvalues are temporary:

- **Lvalues** usually represent objects with a long lifespan; you can't casually move their resources.
- **Rvalues** often represent "soon-to-die" objects (or "suicidal" objects), so we can directly take their resources without needing to copy.

If It Has a Name, It's an Lvalue

```
struct X {
  std::string s;
  // /* bad */ X(X &&other) noexcept : s(other.s) {}
  /*   good */ X(X &&other) noexcept : s(std::move(other.s)) {}
};
```

- `other` is an Lvalue. Directly accessing `other.s` produces an Lvalue, leading to a **copy** of the string.
- Rvalue references extend the lifetime of Rvalues, and using them feels like using an ordinary variable.
- To move `other.s`, you must wrap it in `std::move`.

Pass-by-reference-to-`const` is Insufficient!

We are used to declaring unmodified parameters as reference-to-`const`:

```
class Message {
  std::string m_contents;

public:
  explicit Message(const std::string &contents) : m_contents{contents} {}
};
```

- However, if the input string is an Rvalue, it will still be copied. How can we move this Rvalue?

1. Overloaded Functions

In general, we aim to **copy Lvalues and move Rvalues.**

```cpp
class Message {
  std::string m_contents;

public:
  explicit Message(const std::string &contents)
          : m_contents{contents} {}
  explicit Message(std::string &&contents)
          : m_contents{std::move(contents)} {}
};

std::string s1 = foo(), s2 = bar();
Message m1(s1);      // s1 is copied into m_contents
Message m2(s1 + s2); // Temporary result of s1 + s2 is moved into m_contents
```

2. Pass-by-Value, then Move

Pass by value directly.

```
class Message {
  std::string m_contents;
public:
  explicit Message(std::string contents) : m_contents{std::move(contents)} {}
};
```

Copying/moving is automatically decided during the initialization of contents, and we only need to move contents into `m_contents`.

```
std::string s1 = foo(), s2 = bar();
Message m1(s1);
// `contents` is initialized with Lvalue s1, this is a copy.
Message m2(s1 + s2);
// `contents` is initialized with Rvalue s1 + s2, this is a move
```

A Good Practice on `std::move`

```
void foo(T &);  // (1)
void foo(T &&); // (2)
```

For a variable `x` of type `T`, `foo(x)` will choose (1), while `foo(std::move(x))` will choose (2).

Example: `std::vector<T>::push_back` provides overloads to copy Lvalues and move Rvalues.

```
std::vector<std::string> wordList;
for (int i = 0; i != n; ++i) {
  std::string word; std::cin >> word;
  wordList.push_back(std::move(word)); // word 被移动进 wordList ，而非被拷贝进去。
  // word 的作用到此为止，所以我们应该将它移入 wordList ，不应该拷贝它。
}
```

The `emplace` Function in Standard Library Containers

```
std::vector<Student> students;
students.emplace_back("Alice", "2020123123");
std::vector<std::string> words;
words.emplace_back(10, 'c');
```

- The `emplace` series of functions directly construct the object in place **using the provided arguments**, instead of copying/moving a pre-constructed object.
  - Improves efficiency.
  - Lowers requirements for the stored data type. For instance, `std::list<T>` (linked list) has not required `T` to be copyable/movable since C++11, as long as it can be `constructed and destructed`.
  - `std::vector` caveat: Since vectors need to reallocate memory when growing, they cannot store non-copyable, non-movable elements unless resizing is unnecessary.

The Rule of Five

The *copy control members* in modern C++:

- copy constructor
- copy assignment operator
- move constructor
- move assignment operator
- destructor

The Rule of Five: Define zero or five of them.

Copy and Swap

https://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom

Motivation

To build up **copy assignment operator** from the **copy constructor** and the **destructor** with the idea of *swap*.

- Copy assignment operator is rather more difficult.
- We want our code to be concise.

Swap `Dynarray` Objects

To swap two Dynarray objects.

```
class Dynarray {
public:
  void swap(Dynarray &other) noexcept {
    std::swap(m_storage, other.m_storage);
    std::swap(m_length, other.m_length);
  }
};
```

`std::swap` is robust and safe thus it is recommended to be used.

Copy-and-swap

> 赋值 = 拷贝构造 + 析构：拷贝新的数据，销毁原有的数据。

The first version:

```cpp
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    auto tmp = other;
    swap(tmp);
    return *this;
  }
};
```

- copy constructor will create `tmp`.
- destructor of `tmp` will "delete" it correctly.

Copy-and-swap

A more concise version:

```
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    Dynarray(other).swap(*this);
    return *this;
  }
};
```

That is, we will not explictly create the `tmp` object.

"Copy"-and-swap

Let's make a step further:

```cpp
class Dynarray {
public:
  Dynarray &operator=(Dynarray other) noexcept {
    this->swap(other);
    return *this;
  }
};
```

- If the argument is an lvalue, `other` will be copy-initialized, but we will swap `other` with `*this` thus we have "copied" `other`.
- If the argument is an rvalue, `other` will be move-initialized instead of copy-initialized.
- This means that this assignment operator serves as both a **copy assignment operator** and a **move assignment operator**!

Copy-and-swap

By implementing a fast, `noexcept` `swap` function, multiple advantages are achieved.

Using this `swap` to implement the assignment operator requires no additional operations.

- **Self-assignment safe**.
- **Exception safe** (provides strong exception safety guarantees).
- Combines both **copy assignment** and **move assignment** operators.