CS100 Recitation 10

GKxx

目录

左/右值与拷贝/移动

左/右值与拷贝/移动

假设 `x` 拥有一定的资源。获取它的资源的方式有以下两种：

- 拷贝：将它的资源拷贝出来
- 移动：将它的资源据为己有

左值持久，右值短暂：

- “左值”通常代表具有长久的生命期的对象，你不能随便移动它的资源。
- “右值”通常代表“即将死亡”的对象（当然也有可能是“自杀”），我们可以直接把它的资源据为己有，完全没有必要拷贝。

左/右值与拷贝/移动

一般而言，我们希望**拷贝左值，移动右值。**

- 但如果移动操作不可用，我们也希望能正常拷贝右值。

```
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(m_contents)} {}
};

std::string s1 = foo(), s2 = bar();
Message m1(s1);       // s1 经过 contents 被拷贝给 m_contents
Message m2(s1 + s2); // s1 + s2 产生的临时结果被移动给 m_contents
```

Pass-by-reference-to-`const` 不够好了！

我们已经习惯于将不修改的参数声明为 reference-to-`const`：

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

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

但是如果传进来的字符串是右值，它仍然会被拷贝。怎样移动这个右值？

一种解决方案：像前一页那样提供一组重载，分别接受左值和右值。

Pass-by-value, 然后移动

直接按值传递！

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

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

拷贝/移动会在对参数 `contents` 的初始化中自动决定，而我们只需要把 `contents` 移给 `m_contents`。

```
std::string s1 = foo(), s2 = bar();
Message m1(s1);       // 用左值 s1 初始化参数 contents ，这是拷贝
Message m2(s1 + s2); // 用右值 s1 + s2 初始化 contents ，这是移动
```

`std::move`

`std::move(x)` 并不移动 `x` 。但是假如有以下重载：

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

那么对于一个 `T` 类型的变量 `x`，`foo(x)` 会选择 (1)，`foo(std::move(x))` 会选择 (2)。

例：`std::vector<T>::push_back` 提供一组重载，拷贝左值，移动右值。

```
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。
}
```

只要有名字，就是左值

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

`other` 是一个**左值**，直接取 `other.s` 会得到一个左值，使得这个字符串被拷贝。

- “左值持久，右值短暂”：右值引用延长了右值的生命期，使用右值引用时就如同在使用一个普通的变量。

要想移动 `other.s` ，必须套一层 `std::move` 。

参数转发

回顾 `std::make_shared` 和 `std::make_unique`：

```
auto sp = std::make_shared<std::string>(10, 'c');  // "cccccccccc"
auto sp2 = std::make_shared<std::string>("hello"); // "hello"
auto up = std::make_unique<Student>("Alice", "2020123123");
```

甚至，如果传入右值，它们会移动构造那个对象：

```
auto sp3 = std::make_shared<std::string>(std::move(*sp));
std::cout << *sp << std::endl; // empty string
```

这种将参数转发给另一个函数，又能保持它们的值类别的操作叫做**完美转发** (perfect forwarding)

参数转发

`std::make_shared/unique<T>(...)` 可以接受任意多个任意类型的参数，并将它们原封不动地转发给 `T` 的构造函数，不丢失值类别，不丢失 `const`。

等学了模板，就知道是咋回事了。

标准库很多函数都支持这样的操作，其中非常典型的是**容器的 `emplace` 系列操作**：

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

标准库容器的 `emplace`

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

`emplace` 系列操作利用传入的参数直接原地构造出那个对象，而不是将构造好的对象拷贝/移动进去。

- 提高效率。
- 对所存储的数据类型的要求进一步降低。尤其是 `std::list<T>`（链表）自 C++11 起不需要 `T` 具备任何拷贝/移动操作，只要有办法构造和析构即可。
- `std::vector` 由于需要搬家（增长时重新分配内存），无法存储不可拷贝、不可移动的元素，除非你不需要它搬家。

拷贝控制成员

拷贝控制成员

- 拷贝构造函数 copy constructor
- 拷贝赋值运算符 copy assignment operator
- 移动构造函数 move constructor
- 移动赋值运算符 move assignment operator
- 析构函数 destructor

虽然后三个名字里没有“拷贝”，但也属于“copy control members”。

两个移动操作是 C++11 开始有的。

何时需要

首先，**分清初始化和赋值**。

- 初始化是在变量声明语句中的，它需要调用构造函数。
- 赋值是一个**运算符**，它必然在**表达式**中。
- 构造和赋值有一定相关性，但归根结底调用的是**不同的函数**。

拷贝左值，移动右值。但如果移动操作不可用，右值也被拷贝。

析构函数的调用意味着对象生命期的结束。

- 超出作用域时，程序结束时，以及 `delete` / `delete[]` 表达式

默认行为

在某些情况下（包括用 `= default` 显式要求时），编译器会合成一个具有默认行为的拷贝控制成员。

- 默认行为：**先父类，后自己的成员**，且成员按**声明顺序**，逐个执行对应的操作。
  - 析构顺序相反。
- 默认的移动行为：等同于将 `std::move` 作用于每个成员。
  - 并不是苛求每个成员或父类都采用移动操作。
  - 能移动就移动，不能移动就拷贝。

如果默认行为中涉及的任何一个操作无法正常进行（不存在或不可访问），这个函数就是删除的 (deleted function)。

`= delete`

删除的函数 (deleted function)

- **仍然参与重载决议，**
- **但如果被匹配到，就是 error。**
- 任何函数都可以是删除的，并不局限于特殊成员函数。

特别例外：如果编译器合成了一个删除的移动操作，它不会参与重载决议，这是为了让右值被拷贝。[CWG1402]

- 《C++ Primer》在这个问题上没有及时更新。

三/五法则

C++11 以前是“三”，C++11 以后是“五”。

如果你认为有必要自定义这五个函数中的任何一个，通常意味着这五个你都应该定义。

"Define zero or five or them."

三/五法则

根据"五法则"：如果五个函数中的任何一个具有用户自定义 (user-provided) 的版本，编译器就不应该再合成其它那些用户没有定义的函数。

- 重要例外：一个类不能没有析构函数 ~~就像...~~
- 另一个例外：兼容旧的代码
  - 在 C++98 时代，"三法则"并未在编译器的行为上予以体现。
  - 如果一个类有自定义的拷贝构造函数或析构函数，而没有自定义拷贝赋值运算符，C++98 编译器会合成这个拷贝赋值运算符。（拷贝构造函数同理）
  - 为了兼容旧的代码，不能直接禁止这种行为，只能将它判定为 deprecated。

`Dynarray` 的拷贝赋值运算符

这个对不对？

```
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    delete[] m_storage;
    m_storage = new int[other.m_length];
    std::copy_n(other.m_storage, other.m_length, m_storage);
    m_length = other.m_length;
    return *this;
  }
};
```

`Dynarray` 的拷贝赋值运算符

这个对不对？

```
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    delete[] m_storage;
    m_storage = new int[other.m_length];
    std::copy_n(other.m_storage, other.m_length, m_storage);
    m_length = other.m_length;
    return *this;
  }
};
```

错误！没有做到**自我赋值安全**。

`Dynarray` 的拷贝赋值运算符

这个对不对？

```
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    auto new_data = new int[other.m_length];
    std::copy_n(other.m_storage, other.m_length, new_data);
    m_storage = new_data;
    m_length = other.m_length;
    return *this;
  }
};
```

`Dynarray` 的拷贝赋值运算符

```
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    // if (this != &other) { // 加个判断也可以
      auto new_data = new int[other.m_length];
      std::copy_n(other.m_storage, other.m_length, new_data);
      delete[] new_data;
      m_storage = new_data;
      m_length = other.m_length;
    // }
    return *this;
  }
};
```

`Dynarray` 的移动操作

```
class Dynarray {
public:
  Dynarray(Dynarray &&other) noexcept
      : m_length{other.m_length}, m_storage{other.m_storage} {
    // 为何要有以下两步？
    other.m_length = 0; other.m_storage = nullptr;
  }
  Dynarray &operator=(Dynarray &&other) noexcept {
    if (this != &other) {
      delete[] m_storage;
      m_length = other.m_length; m_storage = other.m_storage;
      // 为何要有以下两步？
      other.m_length = 0;        other.m_storage = nullptr;
    }
  }
};
```

移动操作应将被移动的对象置于**有效的、可被安全析构的**状态。

`Dynarray` 的移动操作

```
class Dynarray {
public:
  Dynarray(Dynarray &&other) noexcept
      : m_length{std::exchange(other.m_length, 0)},
        m_storage{std::exchange(other.m_storage, nullptr)} {}
  Dynarray &operator=(Dynarray &&other) noexcept {
    if (this != &other) {
      delete[] m_storage;
      m_length = std::exchange(other.m_length, 0);
      m_storage = std::exchange(other.m_storage, nullptr);
    }
  }
};
```

Copy-and-swap

能不能写出一个简单的 `swap` 函数，交换两个 `Dynarray` 对象的值？

```
class Dynarray {
public:
  void swap(Dynarray &) noexcept;
};
```

Copy-and-swap

能不能写出一个简单的 `swap` 函数，交换两个 `Dynarray` 对象的值？

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

直接交换 `m_storage` 指针，就可以快速交换两个“动态数组”。这个 `swap` 甚至是 `noexcept` 的，它远远好过传统的 `auto tmp = a; a = b; b = tmp;` 写法。

Copy-and-swap

赋值 = 拷贝构造 + 析构：拷贝新的数据，销毁原有的数据。

能不能利用拷贝构造函数和析构函数写出一个拷贝赋值运算符？

Copy-and-swap

赋值 = 拷贝构造 + 析构：拷贝新的数据，销毁原有的数据。

为 `other` 建立一个拷贝 `tmp`，直接将自己和 `tmp` 交换！

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

- 拷贝构造函数会负责正确拷贝 `other`。
- `tmp` 的析构函数会正确销毁旧的数据。

Copy-and-swap

更简洁些：

```
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    Dynarray(other).swap(*this); // C++23: auto{other}.swap(*this);
    return *this;
  }
};
```

自我赋值安全吗？

Copy-and-swap

```
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    Dynarray(other).swap(*this); // C++23: auto{other}.swap(*this);
    return *this;
  }
};
```

不仅好写，还自我赋值安全，还提供强异常安全保证！

"Copy"-and-swap

更进一步，直接在传参的时候做好“拷贝”。

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

且慢——传参的时候真的发生了拷贝吗？

"Copy"-and-swap

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

如果参数是右值，`other` 将被**移动初始化**，而不是**拷贝初始化**。

也就是说，这个赋值运算符**既是一个拷贝赋值运算符，又是一个移动赋值运算符！**

Copy-and-swap

通过实现一个快速、`noexcept` 的 `swap` 函数，一举多得。

利用这个 `swap` 实现赋值运算符：不需要额外做任何操作。

- 自我赋值安全
- 异常安全（提供强异常安全保证）
- 同时获得拷贝赋值运算符和移动赋值运算符

再看 `Dynarray`

我们将数据成员 `m_length` 和 `m_storage` “藏起来”，并定义了各种接口。有什么好处？

```
struct Dynarray { std::size_t length; int *storage; };
void createDynarray(Dynarray &array, size_t length) {
  array.length = length; array.storage = new int[length]{};
}
void destroyDynarray(Dynarray &array) { delete[] array->storage; }
void dynarrayAssign(Dynarray &lhs, const Dynarray &rhs) { /* ... */ }
```

```
{ // 正确使用：小心翼翼
  Dynarray a;
  createDynarray(a, 10);
  Dynarray b;
  dynarrayAssign(b, a);
  destroyDynarray(a);
  destroyDynarray(b);
}
```

```
// 一不小心就用错
Dynarray a;
createDynarray(a, 10);
Dynarray b = a; // !!
// ...
```

```
// 我没看见 createDynarray
// 但反正都是 public 的，
// 我自己动手！
Dynarray a;
a.length = 1;
a.storage = new int{10};
destroyDynarray(a); // !!
```

`Dynarray` 的 class invariants

A class invariant is a property that must always hold for an object of that class.

- 指针 `m_storage` 要么是空指针，要么指向由 `new[]` 分配的一片内存。
- `m_length` 总是表示 `m_storage` 指向的内存的长度，或者（在 `m_storage` 是空指针时）为零。

用户只能使用我们定义的接口，不能直接访问 `m_length` 或 `m_storage` ，所以**不能轻易破坏 class invariants**

- 构造函数和析构函数在对象被创建和被销毁时会被自动调用，更加保证了这一点，也防止了内存泄漏。