CS100 Recitation 10

GKxx

Contents

- Homework 5 讲评
- 拷贝控制：总结
  - 何时使用
  - 默认行为以及三/五法则
  - 特别技术：copy-and-swap
  - 一个例子
- 其它零碎的知识点
  - `friend`
  - `emplace_back`, `make_shared` 等函数：“完美转发”

Homework 5 讲评

2. 造 Python

算法都很简单，但要用高效的方式实现。

- `std::string` 和 `std::vector` 都必须维护连续存储的特性，所以它们的 `erase` 只能将后面的所有元素集体往前挪，非常慢。
- `s = s + a` 和 `s += a` 的区别，老生常谈的问题。

`split(str, sep)`

- 首先找到 `sep` 第一次出现的位置 $[l_1, r_1]$
- 不断循环，每次在 $[r_{i-1}+1, N]$ 里找 `sep` 第一次出现的位置 $[l_i, r_i]$，这就是 `sep` 第 i 次出现的位置。
  - 直到找不到为止。
- 假设找到了 k 次，位置分别为 $[l_1, r_1], \cdots, [l_k, r_k]$，将下标区间 $[0, l_1)$、$(r_k, N]$ 以及每一个 (r_i, l_{i+1}) 对应的子串取出来，按顺序放进 vector 里，返回。

`split(str, sep)`

- 首先找到 `sep` 第一次出现的位置 $[l_1, r_1]$
- 不断循环，每次在 $[r_{i-1}+1, N]$ 里找 `sep` 第一次出现的位置 $[l_i, r_i]$，这就是 `sep` 第 i 次出现的位置。
  - 直到找不到为止。
- 假设找到了 k 次，位置分别为 $[l_1, r_1], \cdots, [l_k, r_k]$，将下标区间 $[0, l_1)$、$(r_k, N]$ 以及每一个 (r_i, l_{i+1}) 对应的子串取出来，按顺序放进 vector 里，返回。

关键问题：在 $[r_{i-1}+1, N]$ 里找 `sep`，需要把 $[0, r_{i-1}]$ 先删掉吗？

`split(str, sep)`

- 首先找到 `sep` 第一次出现的位置 $[l_1, r_1]$
- 不断循环，每次在 $[r_{i-1}+1, N]$ 里找 `sep` 第一次出现的位置 $[l_i, r_i]$，这就是 `sep` 第 i 次出现的位置。
  - 直到找不到为止。
- 假设找到了 k 次，位置分别为 $[l_1, r_1], \cdots, [l_k, r_k]$，将下标区间 $[0, l_1)$、$(r_k, N]$ 以及每一个 (r_i, l_{i+1}) 对应的子串取出来，按顺序放进 vector 里，返回。

关键问题：在 $[r_{i-1}+1, N]$ 里找 `sep`，需要把 $[0, r_{i-1}]$ 先删掉吗？

- 无论是 `str = str.substr(...)` 还是 `str.erase(...)`，都相当于把 $[r_{i-1}+1, N]$ 全拷贝了一遍，时间复杂度直接 $O\left(N^2\right)$。

拷贝控制：总结

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，它必然调用构造函数。
- 赋值是一个**运算符**，它必然在**表达式**中。

“拷贝”是传统艺能：对于左值必然是拷贝，右值在移动操作存在的情况下被移动。但如果移动操作不存在，右值也被拷贝。（通常情况下）

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)。

为何需要 `std::move`

```
struct X {
  std::string s;
  X(X &&other) noexcept : s(other.s) {}
};
```

右值引用是左值：Anything that has a **name** is an lvalue, even if it is an rvalue reference.

从生命期的角度理解：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（左值）变量。

`other` 是左值，`other.s` 自然是左值，`s(other.s)` 是拷贝而非移动。

`= 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。

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》在这个问题上说的是不对的。

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。

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

《C++ Primer》13.4：拷贝控制实例

13.6.2 讲了 `Message` 类的移动操作。

在类外定义成员函数

`Folder` 和 `Message` 类的众多成员函数相互依赖，因此在两个类型的定义完毕之前，这些成员函数都无法定义。

我们先给出两个类的定义，其中只包括成员函数的声明。

- 某些简单的（例如默认构造函数）就顺手定义在类内了。

在类外定义一个成员函数：在函数名前面加上 `ClassName::` 就打开了类作用域，之后就和在类内一样了。

```
Message::Message(const std::string &contents) : m_contents{contents} {}
void Message::save(Folder &f) { /* ... */ }
Message &Message::operator=(const Message &other) { /* ... */ }
Message::~Message() { /* ... */ }
```

`std::string` 拷贝还是移动？

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

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

但是如果传进来的字符串是右值，能不能直接移动给 `m_contents`？

`std::string` 拷贝还是移动？

如果参数是左值，将它拷贝给 `m_contents`；否则，将它移动给 `m_contents`。

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

`std::string` 拷贝还是移动？

如果参数是左值，将它拷贝给 `m_contents`；否则，将它移动给 `m_contents`。

直接按值传递不就行了？

```
class Message {
 public:
  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`。

拷贝左值，移动右值

标准库的很多函数也为左值和右值做了这样的区分，比如 `std::vector<T>::push_back`。

见 Homework 6 客观题第 1 题。

`friend`

`Message` 的 `save(folder)` 不仅会将 `&folder` 添加进自己的 `m_folders`，还会调用 `f.addMsg(this)` 将自己加进 `folder.m_messages`。

但是如果它本来就在 `folder.m_messages` 里，这就是重复的操作。

能不能允许 `Folder` 直接操作 `Message` 的 `m_folders`？

- 反正这两个类都是我写的，我保证不搞破坏就行了。

`friend`

```
class Message {
  friend class Folder;
};
```

可以将类 `Folder` 声明为 `Message` 的 `friend`，则 `Folder` 的代码可以访问 `Message` 的 `private` 和 `protected` 成员。

- 这个 `friend class Folder;` **不是**成员声明，**不受访问限制修饰符的作用**。
- `friend` 声明可以放在任何位置，通常在一个类的开头或末尾集中声明所有 `friend`。

`friend`

也可以声明某个单独的函数

```
class Message {
  friend class Folder;
  friend void modify(Message &m, const std::string &s);
};
void modify(Message &m, const std::string &s) {
  // 在这里可以直接访问、修改 m.m_contents
}
```

- 这个 `friend void modify(...);` **不是**成员函数声明，`friend` 函数**不是**成员函数，**不受访问限制修饰符的作用**。

`friend`

`friend` 函数也可以在类内定义，但它们仍然不是成员函数。

```
class Message {
  friend class Folder;
  friend void modify(Message &m, const std::string &s) { /* ... */ }
};
```

但是将 `friend` 函数定义在类内会引发特殊的名字查找问题：

```
struct X {
  friend int add(int x, int y) {
    return x + y;
  }
};
```

```
int main() {
  auto x = add(1, 2);
  // Error: `add` was not declared
  // in this scope.
}
```

个人不推荐将 `friend` 函数定义在类内，除了一个和模板有关的特殊情况（以后再说）

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。
- `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。