CS100 Recitation 8

GKxx

`class` 初步

一个简单的 `class`

`class` 除了可以有数据成员，还可以有成员函数等。

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
  void printInfo() const {
    std::cout << "I am " << name << ", id " << id
              << ", entrance year: " << entranceYear << std::endl;
  }
  bool isGraduated(int year) const {
    return year - entranceYear >= 4;
  }
};
```

访问限制

```
class Student {
private:
  std::string name;
  std::string id;
  int entranceYear;
public:
  void printInfo() const {
    std::cout << "I am " << name << ", id " << id
              << ", entrance year: " << entranceYear << std::endl;
  }
  bool isGraduated(int year) const {
    return year - entranceYear >= 4;
  }
};
```

- `private`：外部代码不能访问，只有类内和 `friend` 代码可以访问。
- `public`：所有代码都可以访问。

访问限制

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
public:
  void printInfo() const;
  bool isGraduated(int year) const;
};
```

一个**访问限制说明符**的作用范围是从当前位置开始，一直到下一个访问限制说明符之前（或者到类的定义结束）。

开头有一段没有写访问限制的成员？

- 如果是 `class`，它们是 `private`。
- 如果是 `struct`，它们是 `public`。

C++ `struct` 和 `class` **仅有两个区别**，这里是区别之一。

- 为何需要将数据成员“藏起来”？《Effective C++》条款 22。

`this` 指针

访问一个普通成员的方法是 `a.mem`，其中 `a` 是该类类型的一个**对象**。

- 成员是属于一个对象的：每个学生都有一个姓名、学号、入学年份。你需要指明"谁的"姓名/学号…

```
Student s = someValue(); // 假设它具有一定的值
s.printInfo(); // 调用它的 printInfo() 输出相关信息
if (s.isGraduated(2023)) {
  // ...
}
```

注：我们现在暂时只讨论 non-`static` 成员，以下不再强调。

`this` 指针

```
class Student {
  // ...
public:
  bool isGraduated(int year) const;
};
Student s = someValue();
if (s.isGraduated(2023))
  // ...
```

`isGraduated` 函数有几个参数?

`this` 指针

```
class Student {
  // ...
public:
  bool isGraduated(int year) const;
};
Student s = someValue();
if (s.isGraduated(2023))
  // ...
```

`isGraduated` 函数有几个参数?

- **看似是一个，其实是两个**：`s` 也是调用这个函数时必须知道的信息!

`this` 指针

```
class Student {
public:
  bool isGraduated(int year) const {
    return year - entranceYear >= 4;
  }
};
Student s = someValue();
if (s.isGraduated(2023))
  // ...
```

左边的代码就如同：

```
bool isGraduated(const Student *this,
                 int year) {
  return year - this->entranceYear >=4;
}
Student s = someValue();
if (isGraduated(&s, 2023))
  // ...
```

`this` 指针

在成员函数内部，正有一个名为 `this` 的指针：它具有 `X *` 或者 `const X *` 类型（其中 `X` 是这个类的名字）。

在成员函数内部对任何成员 `mem` 的访问，实际上都是 `this->mem`。

你也可以显式地写 `this->mem`。

```
class Student {
public:
  bool isGraduated(int year) const {
    return year - this->entranceYear >= 4;
  }
};
```

许多语言都有类似的设施：Python 里有 `self`。（C++23 当然也可以有 `self`）

`const` 成员函数

在参数列表后面、函数体的 `{` 前面标一个 `const`

```
class Student {
public:
  bool isGraduated(int year) const;
  void setID(const std::string &);
};

const Student s = someValue();
if (s.isGraduated(2023)) // OK
  // ...
s.setID("2024533123"); // Error
```

- 左边的代码如同：

```
bool isGraduated(const Student *this,
                 int year);
void setID(Student *this,
           const std::string &);
const Student s = someValue();
if (isGraduated(&s, 2023)) // OK
  // ...
setID(&s, "2024533123"); // Error
```

这个 `const` 表明 `this` 的类型是 `const Student *` 。如果没有，则是 `Student *` 。

`const` 成员函数

规则：

1. 如果 `ptr` 的类型是 `const T *` ， `T` 具有类型为 `U` 的数据成员 `member` ，则 `ptr->member` 的类型是 `const U` 。
   - `const` ness 会在成员访问时传递下去。
2. 在类 `X` 的 `const` 成员函数中， `this` 的类型是 `const X *` 。
   - 在 non-`const` 成员函数中， `this` 的类型是 `X *` 。
3. 对成员的访问 `member` 实际上都是 `this->member` ，这里的 `member` 。
   - 这里的 `member` 既可以是数据成员，也可以是成员函数。
4. 不允许去除底层 `const` 。

`const` 成员函数

一些推论：

1. 在 `const` 对象上（包括用 reference-to-`const` 时），只能调用 `const` 成员函数。
2. 在 `const` 成员函数内部，不允许修改自身的数据成员，不能调用自身的 non-`const` 成员函数。
    - 对于类类型的数据成员，也不能调用它的 non-`const` 成员函数。

[Best practice] 如果一个成员函数逻辑上不修改这个对象的状态，那么它就应该是 `const` 成员函数。

《Effective C++》条款 3：尽可能使用 `const`。（下次讲）

控制数据成员的读写

方案一： Getters and setters

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
public:
  const std::string &getName()          const { return name; }
  const std::string &getID()            const { return id; }
  int                getEntranceYear() const { return entranceYear; }
  void setName(const std::string &newName)   { name = newName; }
  void setID(const std::string &newID)       { id = newID; }
  void setEntranceYear(int year)             { entranceYear = year; }
};
```

* 为什么要控制数据成员的读写？这和直接让数据成员变成 public 有什么区别?

控制数据成员的读写

和直接让数据成员变成 public 有什么区别?

- 你可以修改，但是我得知道你在改，而且我不能让你胡改。

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
public:
  void setID(const std::string &newID) {
    id = newID;
    entranceYear = std::stoi(id.substr(0, 4)); // 改 id 的同时更新 entranceYear
  }
  void setEntranceYear(int year) {
    if (year < 2013) // 制止不正确的改动
      throw std::invalid_argument{"Entrance year impossible!"};
    entranceYear = year;
  }
};
```

控制数据成员的读写

和直接让数据成员变成 public 有什么区别?

- 我可以修改实现细节，不会影响用户代码。

```
class Student {
  std::string name;
  std::string id; // 我不想存 entranceYear 了
                  // 没关系，用户本来也不能直接访问 entranceYear
public:
  int getEntranceYear() const { return std::stoi(id.substr(0, 4)); }
};
```

控制数据成员的读写

方案二：读/写重载

```
class Point2d {
  double mX;
  double mY;

public:
  double x()             const { return mX; }
  double y()             const { return mY; }
  void   x(double newX)        { mX = newX; }
  void   y(double newY)        { mY = newY; }
};
```

```
Point2d p = /* ... */;
p.x(3.14);                    // 将 x 改为 3.14
std::cout << p.x() << '\n'; // 访问 x
```

控制数据成员的读写

方案三： `const` 和 non-`const` 重载

- 实际上没有完全控制住“写”操作。
- 容器的下标运算符普遍采用这种方式。

```
class Point2d {
  double mX;
  double mY;
public:
  double  x() const { return mX; }
  double  y() const { return mY; }
  double &x()       { return mX; }
  double &y()       { return mY; }
};
```

```
Point2d p = /* ... */;
p.x() = 3.14;
```

构造函数

构造函数 (constructors, ctors) 负责对象的初始化方式。

- 构造函数往往是**重载** (overloaded) 的，因为一个对象很可能具有多种合理的初始化方式。

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
public:
  Student(const std::string &name_, const std::string &id_, int ey)
    : name(name_), id(id_), entranceYear(ey) {}
  Student(const std::string &name_, const std::string &id_)
    : name(name_), id(id_), entranceYear(std::stoi(id_.substr(0, 4))) {}
};
Student a("Alice", "2020123123", 2020);
Student b("Bob", "2020123124"); // entranceYear = 2020
```

构造函数

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
public:
  Student(const std::string &name_, const std::string &id_)
    : name(name_), id(id_), entranceYear(std::stoi(id_.substr(0, 4))) {}
};
```

- 构造函数的函数名就是类名本身：`Student`
- 构造函数**没有返回值类型**，可以含有 `return;` 但不能返回一个值。
- 上面这个构造函数的函数体是空的：`{}`

构造函数初始值列表

构造函数负责这个对象的初始化，包括**所有成员**的初始化。

一切成员的初始化过程都在**进入函数体之前结束**，它们的初始化方式（部分是）由**构造函数初始值列表** (constructor initializer list) 决定：

```
class Student {
  // ...
public:
  Student(const std::string &name_, const std::string &id_)
    : name(name_), id(id_), entranceYear(std::stoi(id_.substr(0, 4))) {}
};
```

构造函数初始值列表由冒号 `:` 开头，依次给出各个数据成员的初始化器，用 `,` 隔开。

成员的初始化顺序

数据成员按照**它们在类内声明的顺序**进行初始化，而非它们在构造函数初始值列表中出现的顺序。

- 如果你的构造函数初始值列表中的成员的顺序和它们在类内的声明顺序不同，编译器会以 warning 的方式提醒你。

典型的错误：初始化 `entranceYear` 时用到了成员 `id`，而此时 `id` 还未初始化！

```
class Student {
  std::string name;
  int entranceYear;
  std::string id;
public:
  Student(const std::string &name_, const std::string &id_)
    : name(name_), id(id_), entranceYear(std::stoi(id.substr(0, 4))) {}
};
```

构造函数初始值列表

数据成员按照**它们在类内声明的顺序**进行初始化，而非它们在构造函数初始值列表中出现的顺序。

- 如果你的构造函数初始值列表中的成员的顺序和它们在类内的声明顺序不同，编译器会以 warning 的方式提醒你。
- 未在初始值列表中出现的成员：
  - 如果有**类内初始值**（稍后再说），则按照类内初始值初始化；
  - 否则，**默认初始化**。

默认初始化对于内置类型来说就是*不初始化*，但是对于类类型呢？

构造函数初始值列表

下面是典型的“先默认初始化，后赋值”：

```
class Student {
  // ...
public:
  Student(const std::string &name_, const std::string &id_) {
    name = name_;
    id = id_;
    entranceYear = std::stoi(id_.substr(0, 4));
  }
};
```

“先默认初始化，后赋值”有什么坏处？

构造函数初始值列表

“先默认初始化，后赋值”有什么坏处？

- 不是所有类型都能默认初始化，不是所有类型都能赋值。（有哪些反例？）

构造函数初始值列表

“先默认初始化，后赋值”有什么坏处？

- 不是所有类型都能默认初始化，不是所有类型都能赋值。
  - 引用无法被默认初始化，无法被赋值。`const` 对象无法被赋值。
  - （内置类型的）`const` 对象无法被默认初始化。
  - 我们将在后面看到，类类型可以拒绝支持默认初始化和赋值，这取决于设计。
- 如果你把能默认初始化、能赋值的成员在函数体里赋值，其它成员在初始值列表里初始化，**你的成员初始化顺序将会非常混乱**，很容易导致错误。

[Best practice] 尽可能用构造函数初始值列表，按照成员的声明顺序逐个初始化它们。

默认构造函数

特殊的构造函数：不接受参数。

- 专门负责对象的**默认初始化**，因为调用它时不需要传递参数，相当于不需要任何初始化器。
- 类类型的**值初始化**（几乎）就是调用默认构造函数。

```
class Point2d {
  double x, y;
public:
  Point2d() : x(0), y(0) {}
  Point2d(double x_, double y_) : x(x_), y(y_) {}
};
Point2d p1;        // 调用默认构造函数，(0, 0)
Point2d p2{};      // value-initialization ，在这里也是调用默认构造函数，  (0, 0)
Point2d p3(3, 4); // 调用 Point2d(double, double), (3, 4)
Point2d p4();      // 这是在调用默认构造函数吗?
```

你的类需要一个默认构造函数吗?

首先，如果需要开数组，你几乎肯定需要默认构造函数：

```
Student s[1000]; // 全部默认初始化
Student s2[1000] = {a, b}; // 前两个元素是 a 和 b，后面的全部值初始化，
                           // 而值初始化也是调用默认构造函数
```

但关键问题是**它是否具有一个合理的定义?**

- 一个“默认的学生”应该具有什么姓名和学号?

你的类需要一个默认构造函数吗?

如果一个类没有 user-declared 的构造函数，编译器会试图帮你合成一个默认构造函数。

```
struct X {}; // 什么构造函数都没写
X x; // OK：调用了编译器合成的默认构造函数
```

编译器合成的默认构造函数的行为非常简单：逐个成员地进行默认初始化

- 如果有成员有类内初始值，则使用类内初始值。
- 如果有成员没有类内初始值，但又无法默认初始化，则编译器会放弃合成这个默认构造函数。

你的类需要一个默认构造函数吗?

如果一个类有至少一个 user-declared 的构造函数，但没有默认构造函数，则编译器**不会**帮你合成默认构造函数。

- 在它看来，这个类的所有合理的初始化行为就是你给出的那几个构造函数，
- 因此不应该再画蛇添足地定义一个默认行为。

你可以通过 `= default;` 来显式地要求编译器合成一个具有默认行为的默认构造函数：

```
class Student {
public:
  Student(const std::string &name_, const std::string &id_, int ey)
    : name(name_), id(id_), entranceYear(ey) {}
  Student(const std::string &name_, const std::string &id_)
    : name(name_), id(id_), entranceYear(std::stoi(id_.substr(0, 4))) {}
  Student() = default;
};
```

你的类需要一个默认构造函数吗?

宁缺毋滥：如果它没有一个合理的默认构造行为，就不应该有默认构造函数!

- 而不是胡乱定义一个或者 `=defualt`，这会留下隐患。
- 对默认构造函数的调用应当引发**编译错误**，而不是随意地允许。

《Effective C++》条款 6：如果编译器合成的函数你不需要，应当显式地禁止。

类内初始值

default member initializer, 或者叫 in-class initializer

可以在声明数据成员的时候顺便给它一个初始值。

```
struct Point2d {
  double x;
  double y = 0;
  Point2d() = default;
  Point2d(double x_) : x(x_) {} // y = 0
};
Point2d p; // x 具有未定义的值，而 y = 0
Point2d p2(3.14); // x = 3.14, y = 0
```

如果这个成员在某个构造函数初始值列表里没有出现，它会采用类内初始值，而不是默认初始化。

类内初始值

但是类内初始值不能用圆括号...

```
struct X {
  // 错误：受限于编译器的设计，它会在语法分析阶段被认为是函数声明
  std::vector<int> v(10);

  // 这毫无疑问是声明一个函数，而非设定类内初始值
  std::string s();
};
```

成员初始化的方式

从特殊到一般：

1. 如果这个成员在当前构造函数的初始值列表中出现了，则遵循初始值列表中给它的初始化器。
2. 否则，如果这个成员具有类内初始值，则使用类内初始值。
3. 否则，如果它能默认初始化，则默认初始化。
4. 否则，这个成员在当前构造函数中无法被初始化。
    - 如果这是用户自己写的构造函数，则产生编译错误。
    - 如果是编译器合成的构造函数，这个构造函数就是 deleted 的。