CS100 Recitation 8

GKxx

Contents

- Homework 4 讲评
- `class` 初步
  - 成员
  - 访问限制
  - 构造函数
- 一个简单的资源管理类：`Dynarray`

Homework 4

1. 造标准库

出题意图：

- 逼着你们读 cppreference，了解这些标准库函数
- 学会使用 ASCII 码
- 学会用简单的方式实现简单的功能

1. 造标准库

- `islower`, `isupper`, `isalpha`, `isdigit`, `tolower`, `toupper`：用 ASCII 码
- `strcat`, `strlen`, `strcpy`, `strchr`, `strcmp`：“扫一遍”就应该解决问题。

`tolower`, `toupper`：如果不是大写/小写字母，就原样返回。

`strchr`：

> The terminating null character is considered to be a part of the string and can be found when searching for `'\0'`.

`strcmp`：返回值是负值、零、正值，而非一些人一开始写的 0 / 1。

错误处理

谈一个设计问题：应该如何处理参数非法的情况？

```
char *strchr(const char *str, int ch) {
  if (str == NULL)
    return NULL;
  // ...
}
```

这个做法合理吗？

错误处理

```
char *strchr(const char *str, int ch) {
  if (str == NULL)
    return NULL;
  // ...
}
```

标准规定这属于 undefined behavior，因此你自然可以任意处置。

- 但是，`return NULL;` 表示“未找到”，不应该将它和错误行为混起来。

错误处理

使用 `assert`，在条件不满足时退出程序　　对错误静默处理，直接假定参数正确

```
char *strchr(const char *str, int ch) {
  assert(str != NULL);
  // ...
}
```

```
char *strchr(const char *str, int ch) {
  // 压根不检查 str != NULL
  // ...
}
```

- 在 debug 模式下（未定义 `NDEBUG` 宏，`assert` 有效），将错误显式地报告出来。
- 在 release 模式下（定义 `NDEBUG` 以关闭所有 `assert`），直接假定错误行为不会发生，追求效率。

看看孟岩老师的观点以及为什么 C++ IOStream 在这方面是失败的。

为什么用 `int`？

`char` 可能无符号（这是 implementation-defined 的），但有一个重要的字符是负数：**文件结束符** `EOF`（定义为 `-1`）。

- 这意味着用 `char` 存储 `EOF` 可能会在之后的运算中迷失。

`getchar()` 的返回值类型也是 `int`。如果 `char` 被实现为无符号的，那么下面的代码就会出问题：

```
char ch;
while ((ch = getchar()) != EOF && ch != '\n')
  // ...
```

`EOF` 赋值给 `ch` 就立刻变成了 `255`，不会和 `-1` 相等。

2. Memory Leak Checker

目的：实现内存泄漏和非法 `free` 的检查，要对于用户代码的改动尽可能小。

通过 `#define` 将 `malloc`，`calloc` 和 `free` 替换成我们自己的函数，就可以为所欲为了

- 只需要在用户代码的开头 `#include` 我们的代码即可。

保姆级教程，阅读量大，实际上很简单。

使用了 GCC extension，向 VS 用户们道歉，以后会尽量避免。

2. Memory Leak Checker

用一个数据结构记录内存分配申请。需要支持以下操作：

- 添加一条记录
- 查找一条记录
- 删除一条记录

本题数据规模限制在了 `1000` 次分配以内，所以数组即可解决：用一个变量记录“当前使用到哪了”。

- 添加一条记录：`records[cnt++] = ...`
- 查找的时候，只需遍历下标在 `[0, cnt)` 范围内的。

2. Memory Leak Checker

注意 `free(NULL)` 不应该做任何事，不要浪费时间去查找。测试数据中有连着调用 `free(NULL)` 一百万次的。

删除一条记录：“懒惰删除”，直接给 `ptr` 设置为 `NULL`，相当于打了个标记。

其它注意事项在题目里已经唠叨过了，例如 `malloc` 和 `calloc` 在 `size == 0` 时的行为，以及它们可能分配更多内存等等。

2. Memory Leak Checker

用一个数据结构记录内存分配申请。需要支持以下操作：

- 添加一条记录
- 查找一条记录
- 删除一条记录

适合这个问题的数据结构：哈希表、红黑树等。（大家 CS101 再见）

我们会发一个哈希表的实现给大家看一看。理想状况下，哈希表可以做到 $O(1)$ 完成以上三种操作，而朴素的数组上查找只能 $O(n)$，其中 n 是总的记录条数。

3. Brainfuck

运算符	C语言的等价表达
`>`	`ptr++`
`<`	`ptr--`
`+`	`(*ptr)++`
`-`	`(*ptr)--`
`.`	`putchar(*ptr)`
`,`	`*ptr = getchar()`
`[`	`while (*ptr != 0) {`
`]`	`}`

一个 brainfuck 程序有一个 30000 字节的数组，以及一个指向其中某位置的指针 `ptr`。

这个数组是谁?

3. Brainfuck

运算符	C语言的等价表达
`>`	`ptr++`
`<`	`ptr--`
`+`	`(*ptr)++`
`-`	`(*ptr)--`
`.`	`putchar(*ptr)`
`,`	`*ptr = getchar()`
`[`	`while (*ptr != 0) {`
`]`	`}`

一个 brainfuck 程序有一个 30000 字节的数组，以及一个指向其中某位置的指针 `ptr`。

- `uint8_t` 代表“字节”（8 位）
- `state->memory_buffer` 正是一个指向“字节”的指针，自然是让它指向一片 30000 个 `uint8_t` 的内存，来表示这个“数组”。

如何表示 `ptr`？

3. Brainfuck

运算符	C语言的等价表达
`>`	`ptr++`
`<`	`ptr--`
`+`	`(*ptr)++`
`-`	`(*ptr)--`
`.`	`putchar(*ptr)`
`,`	`*ptr = getchar()`
`[`	`while (*ptr != 0) {`
`]`	`}`

一个 brainfuck 程序有一个 30000 字节的数组，以及一个指向其中某位置的指针 `ptr`。

- `uint8_t` 代表“字节”（8 位）
- `state->memory_buffer` 正是一个指向“字节”的指针，自然是让它指向一片 30000 个 `uint8_t` 的内存，来表示这个“数组”。

如何表示 `ptr`？

- 要么真的用一个指针，要么用下标。
- `state->memory_pointer_offset` 是 `size_t` 类型的，自然是一个下标。

3. Brainfuck

如同在 C 里做“面向对象编程”：

- `bf_state_new()` 就是“构造函数”
- `bf_state_delete(state)` 是“析构函数”

`bf_state_new()` 应当创建一个能长久存在的 `bf_state` 对象，直到它被传给 `bf_state_delete` 才被销毁。

- 要么用动态内存，要么用 `static`。
- `static` 类似于全局变量，这要求在每个时刻只能有一个 `bf_state` 对象。
  （如果确实如此，可以这样做）

```c
struct bf_state {
  uint8_t *buffer;
  size_t offset;
};

struct bf_state *bf_state_new(void) {
  struct bf_state *state
    = malloc(sizeof(struct bf_state));
  state->buffer
    = calloc(30000, sizeof(uint8_t));
  state->offset = 0;
  return state;
}

void bf_state_delete
  (struct bf_state *state) {
  free(state->buffer);
  free(state);
}
```

3. Brainfuck

题目要求 `(*ptr)++` 需要对 256 取模

- 这事实上不需要我们手动判断，因为无符号整数不会溢出
- 直接 `++state->memory_buffer[state->memory_pointer_offset]` 即可

`--` 也是如此。

Brainfuck 代码中的任何其它字符都**视为注释**，跳过，而非判为语法错误。

- **常看 Piazza！**

3. Brainfuck

难点：循环

- 如果 `src` 中出现了未匹配的 `[`，那这行代码就不能执行，需要你把它存进某个 buffer。
- 每次如果那个 buffer 里有代码，都要将它和当前的 `src` 连起来看，如果仍然有未匹配的 `[` 就仍然不能执行，继续往后存。
- 直到所有 `[` 都被匹配再执行。

3. Brainfuck

但是其实这件事我们已经帮你做了...

- 如果你阅读 `ibf.h` 中调用 `brainfuck_main` 的上下文，或者在 `brainfuck_main` 的开头 `puts(src);`，就会发现这一点：`src` **中并不存在不匹配的方括号。**

因此 `struct brainfuck_state` 中不需要添加任何成员。

3. Brainfuck

```
const char *cur = src;
while (cur != '\0') {
  switch (*cur) {
    // cases +-<>,.
    case '[':
      if (!loop_condition(state))
        cur = find_matching_end(cur); // Now *cur == ']'
      break;
    case ']':
      if (loop_condition(state))
        cur = find_matching_start(cur); // Now *cur == '['
      break;
  }
  ++cur;
}
```

3. Brainfuck

如何找到和某个 `[` 匹配的 `]`？~~阅读 `ibf.h` 即可找到答案。~~

- 使用一个变量 `unmatched` 表示目前未匹配的 `'['` 的个数。
- 向右扫描，遇到 `'['` 则 `++unmatched`，遇到 `']'` 则 `--unmatched`
- 当 `unmatched == 0` 时返回当前位置。

当然可以用 stack 实现得更精细一些，但这个做法的效率已经足够。

绝大多数 TLE 的原因

```
while (i < strlen(src))
```

(当然也有 `for` 的)

`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 graduated(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 graduated(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 graduated(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。（下次有时间的话讲）

`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**。

- **（非 `static`）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.graduated(2023)) {
  // ...
}
```

`this` 指针

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

`graduated` 函数有几个参数?

`this` 指针

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

`graduated` 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 graduated(int year) const {
    return year - entranceYear >= 4;
  }
};
Student s = someValue();
if (s.graduated(2023))
  // ...
```

左边的代码就如同：

```
bool graduated(const Student *this,
               int year) {
  return year - this->entranceYear >=4;
}
Student s = someValue();
if (graduated(&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数据成员 `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 graduated(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`

- 在 `const` 对象上（包括用 reference-to-`const` 时），**只能调用 `const` 成员函数**
- C++ 中对于 `const` 和 reference-to-`const` 的使用随处可见，
- 因此如果一个成员函数逻辑上不修改这个对象的状态，它就应该是 `const`，否则在 `const` 对象上无法调用。

这个 `const` 相当于施加在 `this` 指针上，作为**底层 `const`**。

- `const` 成员函数不能修改数据成员，不能调用非 `const` 的成员函数。
- `const` 成员函数不能调用数据成员的非 `const` 成员函数。

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

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`
- 构造函数不声明返回值类型，可以含有 `return;` 但不能返回一个值，**但不能认为它的返回值类型是 `void`。**
- 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

只要有可能，就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。

- 猜猜是用来干什么的?

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。
- 特别地，类类型的**值初始化**就是**默认初始化**。

```
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(3, 4); // 调用 Point2d(double, double), (3, 4)
Point2d p3(); // 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

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();
};
```

一个简单的资源管理类：`Dynarray`

设计

实现一个“动态数组”：

```
int n; std::cin >> n;
Dynarray arr(n); // arr has n value-initialized ints.
for (int i = 0; i != n; ++i)
  std::cin >> arr.at(i); // arr.at(i) is just like arr[i]
                         // 如何做到 arr[i]? 以后再说
for (std::size_t i = 0; i != arr.size(); ++i)
  arr.at(i) += arr.at(i - 1) * 2;
// ...
```

最后 `arr` 应当自己释放自己所占用的内存。

Dynarray：成员、构造函数

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
 public:
  Dynarrray(std::size_t n)
    : m_size(n), m_storage(new int[n]{}) {}
  std::size_t size() const { return m_size; }
};
```

- 这个类需要一个默认构造函数吗?

Dynarray：成员、构造函数

假如我们需要一个默认构造函数，它应当具有什么行为？

- 初始化为 `0` 个元素的数组？
  - 这样的数组能使用吗？
- 初始化为 `DYNARRAY_DEFAULT_SIZE` 个元素的数组？

Dynarray：成员、构造函数

假如我们需要一个默认构造函数，它应当具有什么行为？

- 初始化为 `0` 个元素的数组？
  - 在不知道如何拷贝的情况下，我们似乎没有什么能改变数组大小的操作。
  - 如果不能改变大小，那“`0` 个元素的数组”有什么用？
- 初始化为 `DYNARRAY_DEFAULT_SIZE` 个元素的数组？
  - 也许在某些应用场景下，这是合理的。

析构函数 (destructor, dtor)

`Dynarray` 必须在自己不再被使用的时候释放所拥有的内存：

```
void fun() {
  Dynarray a(n);
  if (condition) {
    Dynarray b(m);
    // ...
  } // Now b should release the memory it allocated.
  // ...
} // Now a should release the memory it allocated.
```

析构函数 (destructor, dtor)

析构函数是在这个对象被销毁的时候自动调用的函数。

- 它通常用来完成一些最后的清理
- 特别地，那些**拥有一定资源**的类通常在析构函数里释放它们所拥有的资源

定义一个析构函数

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
 public:
  ~Student() {
    std::cout << "dtor of "
              << name << "called.\n";
  }
};
```

```
{
  Student alice("Alice", "2020123123");
  if (condition) {
    Student bob("Bob", "2020123124");
    // ...
  } // Output "dtor of Bob called."
} // Output "dtor of Alice called."
```

- 析构函数的名字是 `~ClassName`
- 析构函数不接受参数，不声明返回值类型
- 一个类只能有一个析构函数 (until C++20)

Dynarray：析构函数

非常直接的实现

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
 public:
  Dynarray(std::size_t n)
    : m_size(n), m_storage(new int[n]{}) {}
  ~Dynarray() {
    delete[] m_storage;
  }
};
```

析构函数的具体行为

考虑一个问题：`Student` 是否拥有什么资源?

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
  // ...
};
```

析构函数的具体行为

`Student` 本身不拥有什么资源，但是 `std::string` 有。

- `std::string` 当然会在它自己的析构函数里释放内存。

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
  // ...
};
```

在一个 `Student` 被销毁的时候：

- 它显然应该销毁它的所有成员
- 销毁一个成员，显然应该调用那个成员的析构函数。

析构函数的具体行为

析构函数在执行完函数体之后，会自动销毁它的所有数据成员

- 对于类类型成员，会调用它的析构函数来销毁它。
- 猜猜销毁成员的顺序?

析构函数的具体行为

析构函数在执行完函数体之后，会自动销毁它的所有数据成员

- 对于类类型成员，会调用它的析构函数来销毁它。
- 按照成员的声明顺序**倒序**销毁它们。

对比一下构造函数：

- 在执行函数体**之前**初始化所有成员。
- 对于类类型成员，调用它的构造函数进行初始化。
- 初始化的顺序是成员的声明顺序。

析构函数的具体行为

对于成员的销毁是不需要我们写的，会自动完成。

```
class Student {
  std::string name;
  std::string id;
  int entranceYear;
 public:
  ~Student() {} // 编译器会在最后插入代码来调用 id 和 name 的析构函数。
};
```

`Student` 类的析构函数只需一个空函数体即可。

等价的写法：`~Student() = default;`

一个类不能没有析构函数，就像...

如果一个类的析构函数是不可调用的，就意味着它无法被销毁

- 就如同不可降解的塑料

因此 C++ 不允许定义这样的类的对象！

什么情况下析构函数不可调用？

一个类不能没有析构函数，就像...

如果一个类的析构函数是不可调用的，就意味着它无法被销毁

因此 C++ 不允许定义这样的类的对象！

什么情况下析构函数不可调用？

- 你可以显式地 `~ClassName() = delete;` 将析构函数定义为“删除的”。
- 如果析构函数是 `private` 的，它就不能在类外、 `friend` 外被调用。

`new` 和 `delete`

```
std::string *p = new std::string("Hello");
std::cout << *p << std::endl;
delete p;
p = new std::string; // 调用默认构造函数，*p 为空串 ""
std::cout << p->size() << std::endl;
delete p;
```

`new` 表达式会**先分配内存**，然后**构造对象**。

- 如果这是一个类类型，它必然会调用一个构造函数来构造对象。在没有指定如何构造的情况下，它会调用**默认构造函数**。
- 相比之下，来自 C 的 `malloc` 只会分配内存，不构造任何对象（不做任何初始化）。`calloc` 会将这个内存**清零**，而非调用默认构造函数。

`new` 和 `delete`

```
std::string *p = new std::string("Hello");
std::cout << *p << std::endl;
delete p;
p = new std::string; // 调用了默认构造函数，*p 为空串 ""
std::cout << p->size() << std::endl;
delete p;
```

`delete` 表达式会**先销毁对象**，然后释放这片内存。

- 如果这是一个类类型，它必然会调用析构函数来销毁这个对象。
- 相比之下，`free` 只释放内存，不调用析构函数。

`Dynarray`：元素访问

`.at(i)` 应该返回什么？

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
 public:
  ??? at(std::size_t n) {
    return m_storage[n];
  }
};
```

`Dynarray`：元素访问

如果我们希望通过 `arr.at(i)` 来修改这个元素，那必然要返回引用。

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
 public:
  int &at(std::size_t n) {
    return m_storage[n];
  }
};
```

这样可以吗?

`Dynarray`：元素访问

试一试：

```
void print(const Dynarray &arr) {
  for (std::size_t i = 0; i != arr.size(); ++i)
    std::cout << arr.at(i) << ' ';
}
```

无法编译：`at` 不是 `const` 成员函数，无法在 `const Dynarray &` 上调用！

`Dynarray`：元素访问

加个 `const`？

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
 public:
  int &at(std::size_t n) const {
    return m_storage[n];
  }
};
```

结果是在 `const Dynarray` 上，你可以得到其中元素的 non-`const` 引用，进而修改它！

`Dynarray`：元素访问

正确的解决方案：`const` 和 non-`const` 的重载

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;
 public:
  const int &at(std::size_t n) const {
    return m_storage[n];
  }
  int &at(std::size_t n) {
    return m_storage[n];
  }
};
```

- 在 `const` 对象上，它只能调用 `const` 版本，得到 reference-to-`const`，无法修改
- 在非 `const` 对象上会调用哪个?

`Dynarray`：元素访问

```
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;
```

```
// 左边的代码就如同：
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;
```

- 在 `const` 对象上，它只能调用 `const` 版本，得到 reference-to-`const`，无法修改
- 在非 `const` 对象上：两个版本都可以调用，但是
  - 调用 non-`const` 版本是完美匹配，调用 `const` 版本是**添加底层** `const`，因此前者是更好的匹配。

在 `const` vs non-`const` 重载中避免重复

假如我们要模仿标准库的行为的话，`at` 函数应该提供边界检查...

```
class Dynarray {
 public:
  const int &at(std::size_t n) const {
    if (n >= m_length) // 为什么不需要判断 n < 0?
      throw std::out_of_range{"Dynarray subscript out of range."};
    return m_storage[n];
  }
  int &at(std::size_t n) {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range."};
    return m_storage[n];
  }
};
```

在 `const` vs non-`const` 重载中避免重复

假如我们还想在 `at` 访问中做一些其它的记录和检查...

```
class Dynarray {
 public:
  const int &at(std::size_t n) const {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range."};
    log_access();
    verify_integrity();
    return m_storage[n];
  }
  int &at(std::size_t n) {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range."};
    log_access();
    verify_integrity();
    return m_storage[n];
  }
};
```

在 `const` vs non-`const` 重载中避免重复

将一模一样的代码编写两遍实在是太麻烦了... 有没有办法避免重复?

- C++23 deducing-this 能帮得上忙

假如没有额外的语法特性... 能不能让一个函数调用另一个?

- 让谁调用谁?

在 `const` vs non-`const` 重载中避免重复

将一模一样的代码编写两遍实在是太麻烦了... 有没有办法避免重复?

- C++23 deducing-this 能帮得上忙

假如没有额外的语法特性... 能不能让一个函数调用另一个?

假如我们让 non-`const` 版本的函数调用 `const` 版本的函数:

- 首先，我们需要显式地为 `this` 添加底层 `const`。
- `const` 版本的函数返回的是 `const int &`，我们得把它的底层 `const` 去除。

在 `const` vs non-`const` 重载中避免重复

- 先用 `static_cast<const Dynarray *>(this)` 为 `this` 添加底层 `const`
- 这时调用 `->at(n)`，就会匹配 `const` 版本的 `at`
- 将返回的 `const int &` 的底层 `const` 用 `const_cast` 去除

```
class Dynarray {
 public:
  const int &at(std::size_t n) const {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range."};
    log_access();
    verify_integrity();
    return m_storage[n];
  }
  int &at(std::size_t n) {
    return const_cast<int &>(static_cast<const Dynarray *>(this)->at(n));
  }
};
```

在 `const` vs non-`const` 重载中避免重复

能不能反过来，让 `const` 版本调用 non-`const` 版本?

```
class Dynarray {
 public:
  int &at(std::size_t n) {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range."};
    log_access();
    verify_integrity();
    return m_storage[n];
  }
  const int &at(std::size_t n) const {
    return const_cast<Dynarray *>(this)->at(n);
  }
};
```

在 `const` vs non-`const` 重载中避免重复

能不能反过来，让 `const` 版本调用 non-`const` 版本?

- **不能**！ `const` 成员函数里一定不会修改对象的状态，但是 non-`const` 成员函数并没有这般承诺！
- 如果在 non-`const` 版本的实现里一不小心修改了对象的状态，让 `const` 版本调用它将导致灾难。

比较一下两种方法中对于“危险的” `const_cast` 的使用?

为何要将数据成员私有化?

—— 为了“封装”

—— 那什么是“封装”?

当我说“你的这个代码封装性不好”的时候，我究竟表达的是什么意思?

《Effective C++》条款22。