CS100 Recitation 6

GKxx

目录

C++ 的开始

C++ 的开始

首先，我们采用 C++17 语言标准。

- `settings.json`：`code-runner.executorMap` 里 `"cpp"` 项，把 `-std=c++20` 改成 `-std=c++17`
- `c_cpp_properties.json`：`"cppStandard"` 项设置为 `c++17`

调试：最简单粗暴的方法是把 `tasks.json` 和 `launch.json` 都删掉，然后调试 C++ 程序，VSCode 会自动生成。

- 调试 C++ 时要选 `g++.exe`。

Hello C++ World

```cpp
#include <iostream>

int main() {
  std::cout << "Hello C++ World\n";
  return 0;
}
```

`iostream`

```cpp
#include <iostream>

int main() {
  int a, b;
  std::cin >> a >> b;
  std::cout << a + b << '\n';
  return 0;
}
```

`std::cin` 和 `std::cout` 是定义在 `<iostream>` 中的两个**对象**，分别表示标准输入流和标准输出流。

`iostream`

`std::cin >> x` 输入一个东西给 `x`。

- `x` 可以是任何受支持的类型，例如整数、浮点数、字符、字符串。
- **C++ 有能力识别 `x` 的类型并选择正确的方式输入，不需要丑陋的 "%d"、"%c" 来告诉它。**
- C++ 有办法获得 `x` 的**引用**，因此不需要取 `x` 的地址。
- 表达式 `std::cin >> x` 执行完毕后会把 `std::cin` 返回出来，所以可以连着写：`std::cin >> x >> y >> z`

输出也是一样的：`std::cout << x << y << z`

`iostream`

```
std::cout << std::endl;
```

- `std::endl` 是一个“操纵符”。`std::cout << std::endl` 的含义是**输出换行符，并刷新输出缓冲区**。

如果你不手动刷新缓冲区，`std::cout` 自有规则在特定情况下刷新缓冲区。（C `stdout` 也是一样）

namespace `std`

C++ 有一套非常庞大的标准库，为了避免名字冲突，所有的名字（函数、类、类型别名、模板、全局对象等）都在一个名为 `std` 的**命名空间**下。

- 你可以用 `using std::cin;` 将 `cin` 引入当前作用域，那么在当前作用域内就可以省略 `std::cin` 的 `std::`。
- 你可以用 `using namespace std;` 将 `std` 中的所有名字都引入当前作用域，**但这将使得命名空间形同虚设，并且重新引入了名字冲突的风险**。（我个人极不推荐，并且我自己从来不写）

对 C 标准库的兼容

C++ 标准库包含了 C 标准库的设施，**但并不完全一样**。

- 因为一些历史问题（向后兼容），C 有很多不合理之处，例如 `strchr` 接受 `const char *` 却返回 `char *`，某些本应该是函数的东西被实现为宏。
- C 缺乏 C++ 的 function overloading 等机制，因此某些设计显得繁琐。
- C++ 的编译期计算能力远远强过 C，例如 `<cmath>` 里的数学函数自 C++23 起可以在编译时计算。

C 的标准库文件 `<xxx.h>` 在 C++ 中的版本是 `<cxxx>`，并且所有名字也被引入了 `namespace std`。

```
#include <cstdio>
int main() { std::printf("Hello world\n"); }
```

*** 在 C++ 中使用来自 C 的标准库文件时，请使用 `<cxxx>` 而非 `<xxx.h>`**

C++ 中的 C

更合理的设计

- `bool`、`true`、`false` 是内置的，不需要额外头文件
- 逻辑运算符和关系运算符的返回值是 `bool` 而非 `int`
- `"hello"` 的类型是 `const char [6]` 而非 `char [6]`
- 字符字面值 `'a'` 的类型是 `char` 而非 `int`
- 所有有潜在风险的类型转换都不允许隐式发生，不是 warning，而是 error。
- 由 `const int maxn = 100;` 声明的 `maxn` 是编译期常量，可以作为数组大小。
- `int fun()` **不接受参数**，而非接受任意参数。

`std::string`

定义在标准库文件 `<string>` 中（**不是 `<string.h>`，不是 `<cstring>`！！**）

真正意义上的“字符串”。

定义并初始化一个字符串

```
std::string str = "Hello world";
// equivalent: std::string str("Hello world");
// equivalent: std::string str{"Hello world"}; (modern)
std::cout << str << std::endl;

std::string s1(7, 'a');
std::cout << s1 << std::endl; // aaaaaaa

std::string s2 = s1; // s2 is a copy of s1
std::cout << s2 << std::endl; // aaaaaaa

std::string s; // "" (empty string)
```

默认初始化一个 `std::string` 对象会得到空串，而非未定义的值！

一些问题

- `std::string` 的内存：**自动管理，自动分配，允许增长，自动释放**
- `std::string` **不是以空字符结尾的**，它自有办法知道在哪里结束。
  - 它也可能被实现为以空字符结尾的，但**你看不见那个空字符**
- 使用 `std::string` 时，**关注字符串的内容本身，而非它的实现细节**

`std::string` 的长度

`s.size()` 成员函数

```
std::string str{"Hello world"};
std::cout << str.size() << std::endl;
```

不是 `strlen`，更不是 `sizeof` ！！！

`s.empty()` 成员函数

```
if (str.empty()) {
  // ...
}
```

字符串的连接

直接用 `+` 和 `+=` 就行了！

不需要考虑内存怎么分配，不需要 `strcat` 这样的函数。

```
std::string s1 = "Hello";
std::string s2 = "world";
std::string s3 = s1 + ' ' + s2; // "Hello world"
s1 += s2; // s1 becomes "Helloworld"
s2 += "C++string"; // s2 becomes "worldC++string"
```

字符串的连接

`+` 两侧至少有一个得是 `std::string` 对象。

```
const char *old_bad_ugly_C_style_string = "hello";
std::string s = old_bad_ugly_C_style_string + "aaaaa"; // Error
```

下面这个是否合法？

```
std::string hello{"hello"};
std::string s = hello + "world" + "C++";
```

字符串的连接

`+` 两侧至少有一个得是 `std::string` 对象。

```
const char *old_bad_ugly_C_style_string = "hello";
std::string s = old_bad_ugly_C_style_string + "aaaaa"; // Error
```

下面这个是否合法？

```
std::string hello{"hello"};
std::string s = hello + "world" + "C++";
```

合法：`+` 是**左结合**的，`(hello + "world")` 是一个 `std::string` 对象。

使用 `+=`

`s1 = s1 + s2` 会先为 `s1 + s2` 构造一个临时对象，必然要拷贝一遍 `s1` 的内容。

而 `s1 += s2` 是直接在 `s1` 后面连接 `s2`。

试一试：

```
std::string result;
for (int i = 0; i != n; ++i)
  result += 'a';
```

```
std::string result;
for (int i = 0; i != n; ++i)
  result = result + 'a';
```

字符串比较（字典序）

直接用 `<`，`<=`，`>`，`>=`，`==`，`!=`，不需要循环，不需要其它函数！

字符串的拷贝赋值

直接用 `=` 就行

```
std::string s1{"Hello"};
std::string s2{"world"};
s2 = s1; // s2 is a copy of s1
s1 += 'a'; // s2 is still "Hello"
```

`std::string` 的 `=` 会**拷贝**这个字符串的内容。

遍历字符串：基于范围的 `for` 语句

例：输出所有大写字母（`std::isupper` 在 `<cctype>` 里）

```
for (char c : s)
  if (std::isupper(c))
    std::cout << c;
std::cout << std::endl;
```

等价的方法：使用下标，但不够 modern，比较啰嗦。

```
for (std::size_t i = 0; i != s.size(); ++i)
  if (std::isupper(s[i]))
    std::cout << s[i];
std::cout << std::endl;
```

基于范围的 `for` 语句**更好，更清晰，更简洁，更通用，更现代，更推荐**。

字符串的 IO

`std::cin >> s` 可以输入字符串，`std::cout << s` 可以输出字符串。

- `std::cin >> s` 跳不跳过空白？是读到空白结束还是读一行结束？可以自己试试

`std::getline(std::cin, s)`：从当前位置开始读一行，**换行符会读掉，但不会存进来**

总结

真正意义上的“字符串”：`std::string`

- **不以空字符结尾，并且所有内存自动管理。**
- `s.size()` 获得长度，`s.empty()` 判断是否为空串
- 用 `+` 和 `+=` 连接，`<`，`<=`，`>`，`>=`，`==`，`!=` 字典序比较，`=` 拷贝赋值
- `>>` 和 `<<` IO，以及 `std::getline`
- 可以用 `s[i]` 访问元素。
- 遍历：**使用基于范围的 `for` 语句** (range-based `for` loops)
- `std::string` 所有函数（成员、非成员）的完整列表：
  https://en.cppreference.com/w/cpp/string/basic_string

`std::vector`

定义在标准库文件 `<vector>` 中

真正好用的“动态数组”

创建一个 `std::vector` 对象

`std::vector` 是一个**类模板**，只有给出了模板参数之后才成为一个真正的类型。

```
std::vector<int> vi;          // An empty vector of ints
std::vector<std::string> vs; // An empty vector of strings
std::vector<double> vd;       // An empty vector of doubles
```

不同模板参数的 `vector` 是**不同的类型**。

创建一个 `std::vector` 对象

```
std::vector<int> v{2, 3, 5, 7}; // A vector of ints,
                                // whose elements are {2, 3, 5, 7}.
std::vector<int> v2 = {2, 3, 5, 7}; // Equivalent to ↑

std::vector<std::string> vs{"hello", "world"}; // A vector of strings,
                           // whose elements are {"hello", "world"}.
std::vector<std::string> vs2 = {"hello", "world"}; // Equivalent to ↑

std::vector<int> v3(10); // A vector of ten ints, all initialized to 0.
std::vector<int> v4(10, 42); // A vector of ten ints, all initialized to 42.
```

`vector<T> v(n)` 这种构造方式会将 `n` 个元素都**值初始化**（类似于 C 中的"空初始化"），而不是得到一串未定义的值！

创建一个 `std::vector` 对象

```cpp
std::vector<int> v{2, 3, 5, 7};
std::vector<int> v2 = v; // v2 is a copy of v
std::vector<int> v3(v);  // Equivalent
std::vector<int> v4{v};  // Equivalent
```

去年 CS100 一直到期末居然还有人用循环一个元素一个元素拷贝 `vector`，**太愚蠢了！**

```cpp
std::vector<std::vector<int>> v;
```

“二维 `vector`”，也就是“`vector` of `vector`”，当然也是可以的。

C++17 CTAD

Class Template Argument Deduction：只要你给出了足够的信息，编译器可以自动推导元素的类型！

```
std::vector v{2, 3, 5, 7}; // vector<int>
std::vector v2{3.14, 6.28}; // vector<double>
std::vector v3(10, 42); // vector<int>
std::vector v4(10); // Error: cannot deduce template argument type
```

这是什么？

```
std::vector matrix(n, std::vector(m, 0.0));
```

C++17 CTAD

Class Template Argument Deduction：只要你给出了足够的信息，编译器可以自动推导元素的类型！

```
std::vector v{2, 3, 5, 7}; // vector<int>
std::vector v2{3.14, 6.28}; // vector<double>
std::vector v3(10, 42); // vector<int>
std::vector v4(10); // Error: cannot deduce template argument type
```

这是什么？

```
std::vector matrix(n, std::vector(m, 0.0));
```

- `std::vector(m, 0.0)` 创建了一个临时对象，其类型为 `std::vector<double>`（因为 `0.0` 是 `double`），包含 `m` 个元素，每个元素都是 `0.0`。
- `matrix` 的类型是 `std::vector<std::vector<double>>`，包含 `n` 个元素，每个元素都是上面所说的临时对象的拷贝。

`std::vector` 的大小

`v.size()` 和 `v.empty()`

```
std::vector v{2, 3, 5, 7};
std::cout << v.size() << std::endl;
if (v.empty()) {
  // ...
}
```

清空 `std::vector`

`v.clear()`。不要写愚蠢的 `while (!v.empty()) v.pop_back();`

向 `std::vector` 添加元素

`v.push_back(x)` 将元素 `x` 添加到 `v` 的末尾

```
int n;
std::cin >> n;
std::vector<int> v;
for (int i = 0; i != n; ++i) {
  int x;
  std::cin >> x;
  v.push_back(x);
}
```

删除 `std::vector` 最后一个元素

`v.pop_back()`

练习：将末尾的偶数删掉，直到末尾是奇数为止

```
while (!v.empty() && v.back() % 2 == 0)
  v.pop_back();
```

`v.back()`：获得末尾元素**的引用**（这意味着什么？）

`v.back()` 和 `v.front()`

分别获得最后一个元素、第一个元素的**引用**。

“引用”意味着你可以通过这两个成员函数修改它们：

```
v.front() = 42;
++v.back();
```

`v.back()`, `v.front()`, `v.pop_back()` 在 `v` 为空的情况下是 undefined behavior，而且实际上是**严重的运行时错误**。

- 下节习题课再重点讲“引用”。

基于范围的 `for` 语句

遍历一个 `std::vector`，同样可以使用基于范围的 `for` 语句：

```
std::vector<int> vi = some_values();
for (int x : vi)
  std::cout << x << std::endl;
std::vector<std::string> vs = some_strings();
for (const std::string &s : vs) // use reference-to-const to avoid copying
  std::cout << s << std::endl;
```

练习：使用基于范围的 `for` 语句，将一个 `vector<string>` 中的每个字符串的大写字母打印出来。

基于范围的 `for` 语句

练习：使用基于范围的 `for` 语句，将一个 `vector<string>` 中的每个字符串的大写字母打印出来。

```
for (const std::string &s : vs) {
  for (char c : s)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
```

使用下标访问

可以使用 `v[i]` 来获得第 `i` 个元素

- `i` 的有效范围是 $[0, N)$，其中 `N = v.size()`
- 越界访问是**未定义行为**，并且通常是**严重的运行时错误**。
- `std::vector` 的下标运算符 `v[i]` **并不检查越界**，目的是为了保证效率。
  - 事实上标准库容器的大多数操作都没有对合法性进行检查，为了效率。
- 一种检查越界的下标是 `v.at(i)`，它会在越界时抛出 `std::out_of_range` 异常。
  - 不妨自己试一试。

接口的统一性

事实上 `std::string` 也有 `.at()`，`.front()`，`.back()`，`.push_back(x)`，`.pop_back()`，`.clear()` 等函数。C++ 标准库的各种设施是讲究统一性的。

完整列表

练习：实现 Python 的 `rstrip` 函数，接受一个 `std::string`，返回它删去末尾的连续空白后的结果。

接口的统一性

练习：实现 Python 的 `rstrip` 函数，接受一个 `std::string`，返回它删去末尾的连续空白后的结果。

```
std::string rstrip(std::string str) {
  while (!str.empty() && std::isspace(str.back()))
    str.pop_back();
  return str;
}
```

在 C++17 下，这个 `return str;` 是不会产生拷贝的，不必担心。

`std::vector` 的增长策略

考虑像这样连续 `push_back` `n` 次得到一个 `vector` 的代码：

```
std::vector<int> v;
for (int i = 0; i != n; ++i)
  v.push_back(i);
```

`vector` 是如何做到快速增长的？

`std::vector` 的增长策略

$0 \to 1 \to 2 \to 4 \to 8 \to 16 \to \cdots$

假设 $n = 2^m$，那么总的拷贝次数就是 $\sum_{i=0}^{m-1} 2^i = O(n)$，**平均**（"均摊"）一次 `push_back` 的耗时是 $O(1)$（常数），可以接受。

使用 `v.capacity()` 来获得它目前所分配的内存的实际容量，看看是不是真的这样。

- 注意：这仅仅是一种可能的策略，标准并未对此有所规定。

* **看看 hw3/prob4 `attachments/tests/vector.c`**

`std::vector` 动态增长带来的影响

我们已经看到，改变 `vector` 的大小可能会导致它所保存的元素“搬家”，这会使得所有指针、引用、迭代器失效。

- 最直接的影响：下面的代码是 undefined behavior，因为基于范围的 `for` 语句本质上依赖于迭代器。

```
for (int i : vec)
  if (i % 2 == 0)
    vec.push_back(i + 1);
```

不要在用基于范围的 `for` 语句遍历容器的同时改变容器的大小！