CS100 Recitations 6

GKxx

Contents

- Homework 3 讲评
- `struct`
- C++ 的开始
- 初识 `iostream` 和 `std::string`

Homework 3

你真的了解 `fgets` 吗？

- `fgets` 至多读取输入中的几个字符？
- `fgets` 碰到换行符时会终止，换行符读了没有？如果读了，存了没有？
- `fgets` 会在末尾放 `'\0'` 吗？

你真的了解 `fgets` 吗？

- `fgets` 至多读取输入中的几个字符？
- `fgets` 碰到换行符时会终止，换行符读了没有？如果读了，存了没有？
- `fgets` 会在末尾放 `'\0'` 吗？

答案在这里

如果你只是道听途说“`fgets` 可以读一行”就在代码里用它，这就是通向彻夜调试大会的直达车票。

1. 简单题

用 ASCII 码循环移动一下就好了。

不需要存输入的字符串，读一个处理一个就行

不会 ASCII 码？你将无法做 HW4

2. 规则说起来有点麻烦

给一个 keyword（例如 `Wednesday`），首先制作一个表，分为以下几步：

1. 去重，填入前几位：`wednsay`
2. 从最后一个字母（`y`）开始顺着往后接，跳过前面出现过的字母，到 `z` 则循环：`wednsayzbcfghijklmopqrtuvx`
3. 将以上内容视为一个数组 `char encoding[26];`，`encoding[i]` 对应了 `i + 'a'`（或者 `i + 'A'`），按照这个对应关系处理输入的每个字符。

去重，填入前几位

```
int len_keyword = strlen(keyword);
bool used[26] = {0};
int len_encoding = 0;
for (int i = 0; i != len_keyword; ++i) {
  char ch = tolower(keyword[i]);
  if (!used[ch - 'a']) {
    used[ch - 'a'] = true;
    encoding[len_encoding++] = ch;
  }
}
assert(len_encoding < 26);
```

接着往后填

继续使用 `used` 数组

```
char ch = encoding[len_encoding - 1];
while (len_encoding < 26) {
  while (used[ch - 'a'])
    ch = (ch == 'z') ? 'a' : (ch + 1);
  used[ch - 'a'] = true;
  encoding[len_encoding++] = ch;
}
```

处理输入

密文根本不用存，来一个走一个（没人规定读完所有输入才能开始输出）

```
char ch;
while ((ch = getchar()) != EOF && ch != '\n')
  putchar(decode(ch));
```

不要看到一串文本，就想到字符串，就想到 `fgets` 读入，就想到开数组存储。~~你们的想象惟在这一层能够如此跃进~~

decode

暴力：直接在 `encoding` 数组里找

```
char decode(char ch) {
  if (isalpha(ch)) {
    int pos = -1;
    for (int i = 0; i != 26; ++i)
      if (encoding[i] == ch) {
        pos = i;
        break;
      }
    assert(pos != -1);
    return pos + 'a';
  } else
    return ch;
}
```

decode

大写与小写仅有一步之遥，没有本质区别。

```
char decode(char ch) {
  if (isalpha(ch)) {
    bool is_upper = isupper(ch);
    ch = tolower(ch);
    int pos = -1;
    for (int i = 0; i != 26; ++i)
      if (encoding[i] == ch) {
        pos = i;
        break;
      }
    assert(pos != -1);
    return pos + (is_upper ? 'A' : 'a');
  } else
    return ch;
}
```

3. 回文日期 加强版

一年至多可能有多少个回文日期？

4. 找最喜欢的游戏类型

每个游戏有一个 price 和一个 type，type 为 10^6 以内的正整数。找出 price 总和最大的那个 type。

游戏名字根本没有用： `scanf` 的 `%` 和 conversion format specifier 之间加个 `*` 就表示只匹配、不存储。

- https://en.cppreference.com/w/c/io/fscanf#Parameters
- 或者用循环 + `getchar` 跳过。

4. 找最喜欢的游戏类型

每个游戏有一个 price 和一个 type，type 为 10^6 以内的正整数。找出 price 总和最大的那个 type。

- 开个数组 `long long price_sum_of_type[1000001];`
- 对于每个游戏，`price_sum_of_type[type] += price;`
- 枚举所有可能的 `type`，找出 `price_sum_of_type[type]` 最大的那个。
  - 可以 `for (int type = 1; type <= 1000000; ++type)`
  - 也可以在读入的时候顺便记下 `type` 的最大值，缩小枚举范围。

学会计算内存用量

```
struct Game {
  char name[10000001];
  int price;
  int type;
};

struct Game games[1000000];
```

你花了 $\frac{10^7 \times 10^6}{1024 \times 1024 \times 1024} \approx 9300$ GB 来存储那一堆没有用的名字。

5. 二维版 玩具谜题

首先还是那个问题：你真的需要开数组存某个东西吗？

- 读完了 r, c, q、朝向信息以及起始位置之后，就可以读一步、动一步。
- 发现 Mistake 了就直接结束：**没人规定你必须把输入全读完**

判断一个位置是否走过

和刚才那个 `used` 类似的做法：

```
bool visited[501][501];

for (int i = 0; i != q; ++i) {
  // 输入这一步的方向、距离
  // 算出新的位置 (row, col)
  if (valid_coord(row, col) && !visited[row][col]) {
    printf("(%d, %d)\n", row, col);
    visited[row][col] = true;
  } else {
    puts("Mistake!");
    break;
  }
}
```

朝向、方向，一共 16 种情况，如何避免重复？

- 朝向：以“向上”为基准，“向上”、“向右”、“向下”、“向左”分别是顺时针转 $0°$, $90°$, $180°$, $270°$
- 在我的“前”/“右”/“后”/“左”方：再顺时针转 $0°$ / $90°$ / $180°$ / $270°$
- 角度加起来对 $360°$ 取模，就是实际转了多少度，对应四种情况。

朝向、方向，一共 16 种情况，如何避免重复？

- 朝向：以“向上”为基准，“向上”、“向右”、“向下”、“向左”分别是顺时针转 0°, 90°, 180°, 270°
- 在我的“前”/“右”/“后”/“左”方：再顺时针转 0° / 90° / 180° / 270°
- 角度加起来对 360° 取模，就是实际转了多少度，对应四种情况。
- 将四个朝向和四个方向都按顺序编码为 `0`, `1`, `2`, `3`
  - 只要看 `(direction + face) % 4`

`valid_coord`

```
bool valid_coord(int row, int col) {
  return row >= 1 && row <= r && col >= 1 && col <= c;
}
```

不要再写啰嗦的 `return condition ? 1 : 0;` 了！！

`struct`

`struct`

把几个东西结合在一起，定义成一个新的数据结构

```
struct Student {
  const char *name;
  const char *id;
  int entrance_year;
  int dorm;
};
```

```
struct Record {
  void *ptr;
  size_t size;
  int line_no;
  const char *file_name;
};
```

```
struct brainfuck_state {
  uint8_t *memory_buffer;
  size_t offset;
  // ...
};
```

```
struct Point3d {
  double x, y, z;
};
```

```
struct Line3d {
  struct Point3d p0, direction;
};
```

`struct` 类型

`struct` + 名字。C 中 `struct` 关键字不可省略，C++ 中必须省略。

```
struct Student student;
struct Record records[1000];
```

`typedef` 定义类型别名

```
typedef long long LL;
typedef struct { double x, y, z; } Point3d;

LL llval = 0; // llval is long long
Point3d p;
```

不要用 `#define` 代替 `typedef`

`struct` 的成员

`name.mem`

```
struct Student student;
student.name = "Alice";
student.id = "2023533000";
student.entrance_year = 2023;
student.dorm = 8;
printf("%d\n", student.dorm);
++student.entrance_year;
puts(student.name);
```

`struct` 的成员

`ptr->mem`：等价于 `(*ptr).name`。**不是 `*ptr.name` ！！**（`.` 优先级高于 `*`）

```
struct Student *ptr = &student;
ptr->name = "Alice";
ptr->id = "2023533000";
(*ptr).entrance_year = 2023; // equivalent to ptr->entrance_year = 2023;
ptr->dorm = 8;
printf("%d\n", ptr->dorm);
++ptr->entrance_year;
puts(ptr->name);
```

`struct` 初始化

老生常谈的问题：不显式初始化时会发生什么？

```
struct Student gs;
int main(void) {
  struct Student ls;
}
```

`struct` 初始化

老生常谈的问题：不显式初始化时会发生什么？

```
struct Student gs;
int main(void) {
  struct Student ls;
}
```

- 全局或局部 `static` ：**空初始化**：结构体的所有成员都被空初始化。
- 局部非 `static` ：不初始化，所有成员都具有未定义的值。

`struct` 的初始化

Initializer list:

```
struct Record r = {p, cnt * each_size, __LINE__, __FILE__};
```

隔壁 C++20 才有的 designators，C99 就有了！（是不是非常像 Python？）

```
struct Record r2 = {.ptr = p, .size = cnt * each_size,
                    .line_no = __LINE__, .file = __FILE__};
```

C 允许 designators 以任意顺序出现，C++不允许。

`struct` 的初始化

```
struct Record r = {p, cnt * each_size, __LINE__, __FILE__};
struct Record r2 = {.ptr = p, .size = cnt * each_size,
                    .line_no = __LINE__, .file = __FILE__};
```

赋值不行：

```
r = {p, cnt * each_size, __LINE__, __FILE__};          // Error
records[i] = {.ptr = p, .size = cnt * each_size,
              .line_no = __LINE__, .file = __FILE__}; // Error
```

但加个类型转换就好了：

```
r = (struct Record){p, cnt * each_size, __LINE__, __FILE__};          // OK
records[i] = (struct Record){.ptr = p, .size = cnt * each_size,
                             .line_no = __LINE__, .file = __FILE__}; // OK
```

在函数之间传递 `struct`

传参的语义是**拷贝**。

```
void print_info(struct Record r) {
  printf("%p, %zu, %d, %s\n", r.ptr, r.size, r.line_no, r.file_name);
}

print_info(records[i]);
```

传参时发生了这样的**初始化**：

```
struct Record r = records[i];
```

`struct` 的拷贝：**逐元素拷贝**。

在函数之间传递 `struct`

传参时发生了这样的**初始化**：

```
struct Record r = records[i];
```

`struct` 的拷贝：**逐元素拷贝**。就如同

```
r.ptr = records[i].ptr;
r.size = records[i].size;
r.line_no = records[i].line_no;
r.file_name = records[i].file_name;
```

在函数之间传递 `struct`

返回一个 `struct`：严格按照语法来说，也是**拷贝**：

```
struct Record fun(void) {
  struct Record r = something();
  some_computations(r);
  return r;
}

records[i] = fun();
```

`return r;`：发生了形如 `struct Record tmp = r;` 的**拷贝**，临时对象 `tmp` 是表达式 `fun()` 的求值结果。然后发生了形如 `records[i] = tmp;` 的**拷贝**。

但实际上这个过程会被编译器优化，标准也是允许这种优化的。（我们以后在 C++ 里进一步讨论这个问题）

数组成员

```
struct A {
  int array[10];
  // other members
};
```

虽然编译器拒绝直接拷贝数组，但它其实有能力做到。

拷贝一个 `struct A` 时，编译器会自动逐元素拷贝数组。

```
int a[10];
int b[10] = a; // Error!
```

```
struct A a;
struct A b = a; // OK
```

`struct` 的大小

```
struct A {
  int x;
  char y;
  double z;
};
```

`sizeof(struct A)` 是多少？

`struct` 的大小

```
struct A {
  int x;
  char y;
  double z;
};
```

`sizeof(struct A) >= sizeof(int) + sizeof(char) + sizeof(double)`。由于内存对齐的问题，编译器可能会在某些地方插入一定的空白。

`struct` 的大小

```
struct A {
  int x;
  struct A a;
};
```

`sizeof(struct A)` 是多少？

`struct` 的大小

```
struct A {
  int x;
  struct A a;
};
```

`sizeof(struct A)` $= +\infty$。因此这种行为是**不允许的**。

- 从物理上讲：这样的东西无法存储。
- C 类型系统认为：在 `};` 之前（定义完毕之前）这个类型是**不完全类型** (incomplete type)。对于不完全类型，不能定义这个类型的对象，不能访问这个类型的成员，只能定义这个类型的指针，并且不能解引用。

练习

考虑三维空间中的点 (x, y, z) 以及直线 $\mathbf{P}(t) = \mathbf{P}_0 + t\mathbf{v}$。定义 `struct` 表示这两个概念。定义一些函数，例如计算点到直线的距离、计算给定参数 t 的值时的点、输出一些信息。试着使用 initializer list 和 designators 进行初始化。

```
double dist(struct Point3d p, struct Line3d line);
struct Point3d line_at(struct Line3d line, double t);
```

练习

```
struct Point3d {
  double x, y, z;
};
struct Line3d {
  struct Point3d p0, v;
};
struct Point3d line_at(struct Line3d line, double t) {
  return (struct Point3d) {
    .x = line.p0.x + t * line.v.x,
    .y = line.p0.y + t * line.v.y,
    .z = line.p0.z + t * line.v.z
  };
}
```

总结

没有总结。独立完成上面的练习即可真正理解 `struct`。

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

```
#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