CS100 Recitation 7

GKxx

Contents

- 引用
- 左值和右值
- `new` 和 `delete`

引用 (Reference)

动机

练习：使用基于范围的 `for` 语句遍历一个字符串，将大写改为小写

动机

练习：使用基于范围的 `for` 语句遍历一个字符串，将大写改为小写

```
for (char c : str)
  c = std::tolower(c);
```

这样写是**不行的**：`for (char c : str)` 相当于让 `char c` 依次成为 `str` 中的每个字符的拷贝，在 `c` 上修改不会影响 `str` 的内容。

如同

```
for (std::size_t i = 0; i != str.size(); ++i) {
  char c = str[i];
  c = std::tolower(c);
}
```

动机

练习：使用基于范围的 `for` 语句遍历一个字符串，将大写改为小写

```
for (char &c : str)
  c = std::tolower(c);
```

`char &c` 定义了一个**绑定到 `char` 的引用**，我们让 `c` 依次绑定到 `str` 中的每个字符。

如同

```
for (std::size_t i = 0; i != str.size(); ++i) {
  char &c = str[i];
  c = std::tolower(c); // effectively str[i] = std::tolower(str[i]);
}
```

引用即别名

```
int ival = 42;
int &iref = ival;
++iref;              // ival becomes 43
iref = 50;           // ival becomes 50
std::cout << ival << std::endl; // 50
std::cout << iref << std::endl; // 50
int &iref2 = iref; // Equivalent to int &iref2 = ival;
                   // iref2 also bounds to ival.
```

可以写 `Type &r`，也可以 `Type& r`

- `r` 的类型是 `Type &` 。
- 但 `Type& r1, r2, r3;` 只有 `r1` 是引用。
- 如果想定义多个引用，需要 `Type &r1, &r2, &r3;`

引用必须初始化（即在定义时就指明它绑定到谁），并且这个绑定关系不可修改。

引用绑定到的类型必须是对象类型

普通的变量、数组、指针等都是对象

- 可以定义绑定到数组的引用：

```
int a[10];
int (&ar)[10] = a;
ar[0] = 42; // a[0] = 42
```

- 可以定义绑定到指针的引用：

```
int *p;
int *&pr = p;
pr = &ival; // p = &ival;
```

但“引用”本身只是一个傀儡，是一个对象的别名。

- 不能定义绑定到引用的引用：

```
int i = 42;
int &r = i;
int & &rr = r; // No such thing!
```

- 同样地，指针指向的类型、数组的元素类型也不能是引用。

```
int &*pr = &r; // No such thing!
int &ar[N]; // No such thing!
```

引用即别名

一个引用被声明（并初始化）之后，使用它就是在使用它所绑定到的对象。

```
std::string s;
std::string &r = s;
r.push_back('a');         // 如同 s.push_back('a');
r = "hello";              // 如同 s = "hello";
std::cout << r << '\n'; // 如同 std::cout << s << '\n';
std::string &r2 = r;     // 如同 std::string &r2 = s; ,  r2 也被绑定到 s
```

- **初始化**和**赋值**对于引用类型有巨大的区别：初始化是设定它“绑定到谁”，而给一个引用赋值就是在给它所绑定到的对象赋值。

传引用参数 (pass-by-reference)

可以定义绑定到数组的引用：

```
void print_array(const int (&array)[10]) { // 只读，所以加 const
  for (int x : array)           // 基于范围的 for 语句可以用来遍历数组！
    std::cout << x << ' ';
  std::cout << std::endl;
}
int a[10], ival = 42, b[20];
print_array(a);     // Correct
print_array(&ival); // Error!
print_array(b);     // Error!
```

这个 `array` 真的是（绑定到了）一个数组，而不是指向数组首元素的指针。

- `array` 只能绑定到 `int[10]` （可带 `const`），其它类型的参数都会导致 Error 。

传引用参数 (pass-by-reference)

定义一个函数，接受一个字符串，将其中的大写字母改为小写字母

```
void to_lower(std::string &str) {
  for (char &c : str)
    c = std::tolower(c);
}
```

```
std::string hello = "Hello";
to_lower(hello); // std::string &str = hello;
```

- 还可以怎样设计？

传引用参数 (pass-by-reference)

定义一个函数，接受一个字符串，输出其中的大写字母。

```
void print_upper(std::string str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
```

传引用参数 (pass-by-reference)

```
void print_upper(std::string str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
std::string s = some_very_long_string();
print_upper(s);
```

传参的过程中发生了形如 `std::string str = s;` 的**拷贝初始化**。如果 `s` 很长，这将非常耗时。

传引用参数 (pass-by-reference)

以**引用**的方式传参，避免拷贝：

```
void print_upper(std::string &str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
std::string s = some_very_long_string();
print_upper(s); // std::string &str = s;
```

传引用参数 (pass-by-reference)

以**引用**的方式传参，避免拷贝：

```
void print_upper(std::string &str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
std::string s = some_very_long_string();
print_upper(s); // std::string &str = s;
```

但这样有问题：

```
const std::string hello = "Hello";
print_upper(hello); // Error
```

- 这不合理，因为 `print_upper` 明明是个只读的操作。

传递常量引用 (reference-to-`const`)

最好的做法：传递 reference-to-`const`

```
void print_upper(const std::string &str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
```

- 既避免了拷贝，又允许传递 `const` 对象，又能防止错误的修改。

左值 (lvalue) 和右值 (rvalue)

左值和右值

一个表达式在被使用时，有时我们使用的是它代表的**对象**，有时我们仅仅是使用了那个对象的**值**。

- `str[i] = ch` 中，我们使用的是表达式 `str[i]` 所代表的**对象**
- `ch = str[i]` 中，我们使用的是表达式 `str[i]` 所代表的对象的**值**。

左值和右值

一个表达式在被使用时，有时我们使用的是它代表的**对象**，有时我们仅仅是使用了那个对象的**值**。

- `str[i] = ch` 中，我们使用的是表达式 `str[i]` 所代表的**对象**
- `ch = str[i]` 中，我们使用的是表达式 `str[i]` 所代表的对象的**值**。

一个表达式本身带有**值类别** (value category) 的属性：它要么是左值，要么是右值

- 左值：它代表了一个实际的对象
- 右值：它仅仅代表一个值

左值和右值

一个表达式本身带有**值类别** (value category) 的属性：它要么是左值，要么是右值

- 左值：它代表了一个实际的对象
- 右值：它仅仅代表一个值

哪些表达式能代表一个实际的对象？

哪些表达式产生的仅仅是一个值？

左值和右值

一个表达式本身带有**值类别** (value category) 的属性：它要么是左值，要么是右值

- 左值：它代表了一个实际的对象
- 右值：它仅仅代表一个值

哪些表达式能代表一个实际的对象？

- `*addr`，`a[i]` 等等

哪些表达式产生的仅仅是一个值？

- `a + b`，`&val`，`cond1 && cond2` 等等。

* **“左值”和“右值”这两个名字是怎么来的？**

左值和右值

在 C 中，左值可以放在赋值语句的左侧，右值不能。

但在 C++ 中，二者的区别远没有这么简单。

目前已经见过的返回左值的表达式：`*p`，`a[i]`

特别地：在 C++ 中，前置递增/递减运算符返回**左值**，`++i = 42` 是合法的。

赋值表达式返回左值：`a = b` 的返回值是 `a` 这个对象。

- 试着解释表达式 `a = b = c` ？

左值和右值

在 C 中，左值可以放在赋值语句的左侧，右值不能。

但在 C++ 中，二者的区别远没有这么简单。

目前已经见过的返回左值的表达式：`*p`，`a[i]`

特别地：在 C++ 中，前置递增/递减运算符返回**左值**，`++i = 42` 是合法的。

赋值表达式返回左值：`a = b` 的返回值是 `a` 这个对象（的引用）。

- 赋值运算符**右结合**，表达式 `a = b = c` 等价于 `a = (b = c)` ：先执行 `b = c`，然后相当于 `a = b`。

左值和右值

右值仅仅代表一个值，不代表一个实际的对象。常见的右值有**表达式执行产生的临时对象**和**字面值**。

```
std::string fun(); // a function that returns a std::string object
std::string a = fun();
```

- 函数调用 `fun()` 生成的临时对象是**右值**。
- 特别的例外：**字符串字面值** `"hello"` **是左值**，它是长期存在于内存中的对象。
  - 相比之下，整数字面值 `42` 仅仅产生一个临时对象，是右值。

左值和右值

一种常见的右值：通过类型转换生成的临时对象

```
std::string &r1 = std::string("hello"); // Error
std::string &r2 = "hello"; // Error. This is equivalent to ↑
```

Functional-style cast expression: `Type(args...)` ，会生成一个 `Type` 类型的临时对象。

- 对于类类型，这会调用一个适当的构造函数（或者类型转换运算符）
  - 例如 `std::string(10, 'c')`, `std::string("hello")`
- 对于内置类型，就是一个普通的拷贝或者类型转换
  - `int(x)` 会生成一个 `int` 类型的临时对象，其值由 `x` 初始化。

引用只能绑定到左值

```
int ival = 42;
int &iref = 42;          // Error
int &iref2 = ival;       // Correct
int &iref3 = ival + 42; // Error
int fun();
int &iref4 = fun();      // Error
```

C++11 引入了所谓的“右值引用”，我们在介绍**移动**的时候再讲。一般来说，“引用”指的是“左值引用”。

引用只能绑定到左值

```
int arr[10];
int &subscript(int i) { // function returning int&
  return arr[i];
}
subscript(3) = 42; // Correct.
int &ref = subscript(7); // Correct. ref bounds to arr[7]
```

Reference-to-`const`

类似于“指向常量的指针”（即带有“底层 `const`”的指针），我们也有“绑定到常量的引用”

```
int ival = 42;
const int cival = 42;
const int &cref = cival; // Correct
const int &cref2 = ival; // Also correct.
int &ref = cival;  // Error: Casting-away low-level const is not allowed.
int &ref2 = cref;  // Error: Casting-away low-level const is not allowed.
int &ref3 = cref2; // Error: Even though cref2 bounds to non-const ('ival'),
                   // this is still casting-away low-level const.
```

一个 reference-to-`const` **自认为自己绑定到 `const` 对象**，所以不允许通过它修改它所绑定的对象的值，也不能让一个不带 `const` 的引用绑定到它。（不允许“去除底层 `const`”）

Reference-to-`const`

指针既可以带顶层 `const`（本身是常量），也可以带底层 `const`（指向的东西是常量），但引用**不谈**“顶层 `const`”。

- 即，只有“绑定到常量的引用”。引用本身不是对象，不谈是否带 `const`。
- 从另一个角度讲，引用本身一定带有“顶层 `const`”，因为绑定关系不能修改。
- 在不引起歧义的情况下，通常用**常量引用**这个词来代表“绑定到常量的引用”。

Reference-to-`const`

特殊规则：常量引用可以绑定到右值：

```
const int &cref = 42; // Correct
int fun();
const int &cref2 = fun(); // Correct
int &ref = fun(); // Error
```

当一个常量引用被绑定到右值时，实际上就是让它绑定到了一个临时对象。

- 这是合理的，反正你也不能通过常量引用修改那个对象的值

Pass-by-reference-to-`const`

```
void print_upper(std::string &str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
print_upper("Hello"); // Error
```

当我们传递 `"Hello"` 给 `std::string` 参数时，实际上发生了一个由 `const char [6]` 到 `std::string` 的**隐式转换**，这个隐式转换产生**右值**，无法被 `std::string&` 绑定。

Pass-by-reference-to-`const`

将参数声明为**常量引用**，既可以避免拷贝，又可以允许传递右值

```
void print_upper(const std::string &str) {
  for (char c : str)
    if (std::isupper(c))
      std::cout << c;
  std::cout << std::endl;
}
std::string s = some_very_long_string();
print_upper(s); // const std::string &str = s;
print_upper("Hello"); // const std::string &str = "Hello";, Correct
```

Pass-by-reference-to-`const`

将参数声明为**常量引用**，既可以避免拷贝，又可以允许传递右值，也可以传递常量对象，也可以**防止你不小心修改了它**。

在 C++ 中声明函数的参数时，**尽可能使用常量引用**（如果你不需要修改它）。

（如果仅仅是 `int` 或者指针这样的内置类型，可以不需要常量引用）

真正的“值类别”

（语言律师需要掌握）

C++ 中的表达式依值类别被划分为如下三种：

英文	中文	has identity?	can be moved from?
lvalue	左值	yes	no
xvalue (expired value)	亡值	yes	yes
prvalue (pure rvalue)	纯右值	no	yes

lvalue + xvalue = glvalue（广义左值），xvalue + prvalue = rvalue（右值）

- 所以实际上“左值是实际的对象”是不严谨的，右值也可能是实际的对象（xvalue）。之后讲移动的时候我们会见到一个典型的 xvalue 。

`new` 和 `delete`（初步）

`new` 表达式

动态分配内存，**并构造对象**

```
int *pi1 = new int;      // 动态创建一个默认初始化的 int
int *pi2 = new int();    // 动态创建一个值初始化的 int
int *pi3 = new int{};    // 同上，但是更 modern
int *pi4 = new int(42); // 动态创建一个 int，并初始化为 42
int *pi5 = new int{42}; // 同上，但是更 modern
```

对于内置类型：

- **默认初始化** (default-initialization)：就是未初始化，具有未定义的值
- **值初始化** (value-initialization)：类似于 C 中的“空初始化”，是各种零。

`new[]` 表达式

动态分配“数组”，**并构造对象**

```
int *pai1 = new int[n];            // 动态创建了 n 个 int，默认初始化
int *pai2 = new int[n]();          // 动态创建了 n 个 int，值初始化
int *pai3 = new int[n]{};          // 动态创建了 n 个 int，值初始化
int *pai4 = new int[n]{2, 3, 5}; // 动态创建了 n 个 int，前三个元素初始化为 2,3,5
                                   // 其余元素都被值初始化（为零）
                                   // 如果 n<3，抛出 std::bad_array_new_length 异常
```

对于内置类型：

- **默认初始化** (default-initialization)：就是未初始化，具有未定义的值
- **值初始化** (value-initialization)：类似于 C 中的“空初始化”，是各种零。

`delete` 和 `delete[]` 表达式

销毁动态创建的对象，并释放其内存

```
int *p = new int{42};
delete p;
int *a = new int[n];
delete[] a;
```

- `new` 必须对应 `delete`，`new[]` 必须对应 `delete[]`，否则是 **undefined behavior**
- 忘记 `delete`：内存泄漏

一一对应，不得混用

违反下列规则的一律是 undefined behavior:

- `delete ptr` 中的 `ptr` 必须等于某个先前由 `new` 返回的地址
- `delete[] ptr` 中的 `ptr` 必须等于某个先前由 `new[]` 返回的地址
- `free(ptr)` 中的 `ptr` 必须等于某个先前由 `malloc`, `calloc`, `realloc` 或 `aligned_alloc` 返回的地址。

`new`/`delete` vs `malloc`/`free`

C++ 的对象模型比 C 复杂得多，而 `new`/`delete` 也比 `malloc`/`free` 做了更多的事：

- `new`/`new[]` 表达式会**先分配内存，然后构造对象**。对于类类型的对象，它可能会调用一个合适的**构造函数**。
- `delete`/`delete[]` 表达式会**先销毁对象，然后释放内存**。对于类类型的对象，它会调用**析构函数**。

在 C++ 中，非必要不手动管理内存

- 当你需要创建“一列数”、“一列对象”，或者“一张表”、“一个集合”时，**优先考虑标准库容器等设施**，例如 `std::string`，`std::vector`，`std::deque` (双端队列)，`std::list` / `std::forward_list` (链表)，`std::map` / `std::set` (红黑树)，`std::unordered_map` / `std::unordered_set` (哈希表)
- 当你需要动态创建单个对象时，应该优先考虑**智能指针** (`std::shared_ptr`，`std::unique_ptr`，`std::weak_ptr`)
- 只有在特殊情况下（例如手搓一个标准库没有的数据结构，并且对效率有极高的要求），使用 `new` / `delete` 来管理动态内存
- 当你对于内存分配本身也有特殊的要求时，才需要使用 C 的内存分配/释放函数，但通常也是用它们来定制 `new` 和 `delete`