指针和引用

编码规范：* 和 & 紧挨变量名且与关键字之间保留一个空格。引用传递 int &b = a 修改 b 等于修改 a。地址传递 int *b = &a 修改 *b 等于修改 a。指针 * 的优先级低于 []，故 int *nums[3] 是指针数组。[捕获变量](参数列表) -> 返回类型 {函数主体}是 Labmda 表达式，即匿名函数，其返回类型可以省略。

字符数组的初始化

```
char arr[] = {'A', 0}; // 必须有 0 终止符
int arr[10]{ 0 }; // 必须赋初值 0，否则乱码
char *arr[] = {"ABC", "DEF"}; // 指针数组
```

字符数组的修改

```
char arr[] = "hello";
arr[0] = 'X'; // 正确，可以修改栈区
char *ptr = "world";
ptr[0] = 'X'; // 错误，不能修改常量区
```

字符数组的复制

```
// 将数组复制到栈区
char str[] = "hello";
char buf[10];                // 开辟栈区空间
int sz = sizeof(str) / sizeof(str[0]);
//strcpy(buf, str);          // UNIX 习惯
strcpy_s(buf, sz, str);      // 不能用 b = a
strcmp(buf, str) == 0;       // 不能用 b == a

// 将数组复制到堆区
size_t len = strlen(str);
char *p = nullptr;
p = (char *) malloc(sizeof(char) * (len + 1));
strcpy_s(buf, len + 1, str);// 不能忘记 + 1
if (strcmp(buf, str) == 0)  // 不能用 b == a
  free(p);                  // 释放动态开辟的空间
```

动态开辟空间 new / delete

```
Object *ptr = new Object[100];
delete[] ptr; // 删除对象数组，防止内存泄漏
char **ptr = new char *[100];
delete[] ptr; // 动态创建指针（字符串）数组
```

类型转换 char chs[100]{ 0 }

```
ptr = (char *) str.c_str();// string -> char*
double d = atof("0.23");  // string -> double
int i = atoi("1021");     // string -> int
long l = atol("303992");  // string -> long
sprintf(chs, "%f", 2.3);  // 保存到 chs
char *ptr = chs;          // char[] -> char*
strcpy(chs, ptr, len);    // char* -> char[]
string str = chs;         // char[] -> string
```

动态数组 vector

```
vector<int> arr(sz, val); // sz,val 均可缺省
vector<vector<int>> dp(m, vector<int>(n));
empty(), size()
push_back(), pop_back(), dp[i][j] = 13
```

字符串 string

```
string str = "ABCDEFG";
size(), empty()
push_back(), pop_back(), str[i] = 'X'
s1 == s2, substr(start, len)
```

哈希表 unordered_map

```
unordered_map<string, int> map;
size(), empty(), count(key), emplace(k, v)
insert({key, val}), erase(key), at(key) = val
```

哈希集合 unordered_set

```
unordered_set<string> set;
size(), empty(), count(key)
insert(key), erase(key)
```

队列 queue

```
queue<sting> q;
size(), empty(), push(), pop(), front()
```

堆栈 stack

```
stack<string> s;
size(), empty(), push(), pop(), top()
```

运算符重载

作为类成员时，重载二元运算符参数只有一个，重载一元运算符不需要参数。作为全局函数时，重载二元运算符需要两个参数，重载一元运算符需要一个参数。

```
// 作为成员函数
Complex Complex::operator+(Complex &a) {
  Complex b; // 复数
  b.real = this->real + a.real; // 实部
  b.img = this->img + a.img; // 虚部
  return b;
}
// 作为全局函数
Complex operator+(Complex &a, int b) {
  return Complex(a.real + b, a.img);
}
```

将运算符重载函数作为全局函数时，一般都需要在类中将该函数声明为友元函数。

工具函数

```
sort(v1.begin(), v1.end(), greater<int>());
sort(v1.begin(), v1.end(), [](int a, int b) {
  return a > b; // 降序排列
});
reverse(v1.begin(), v1.end());
binary_search(v1.begin(), v1.end(), target);
```