变量类型

使用关键字auto可以自动推导变量的类型，相当于java的var。

```
auto x = 10;         // int
auto y = 3.14;       // double
auto str = "Hello"; // const char*
```

字符串常量

```
string greeting = "hello, runoob";
```

类型限定符

const	**const 定义常量，表示该变量的值不能被修改。**
static	用于定义静态变量，表示该变量的作用域仅限于当前文件或当前函数内，不会被其他文件或函数访问。
restrict	由 **restrict** 修饰的指针是唯一一种访问它所指向的对象的方式（C99新特性）
mutable	表示类中的成员变量可以在 const 成员函数中被修改。
volatile	表示变量的值可能会被程序以外的因素改变，如硬件或其他线程。

```
// mutable与const作用相反，mutable可以突破const的限制
// 被mutable修饰的变量永远处于可变的状态
#include <iostream>
using namespace std;

class Example {
    mutable int counter;

public:
    Example() : counter(0) {}

    void increment() const {
        counter++;  // mutable 变量可以在 const 函数中修改
    }

    int getCounter() const { return counter; }
};
```

```
int main() {
    Example obj;
    obj.increment();
    cout << "Counter: " << obj.getCounter() << endl;
    return 0;
}
```

Lambda函数与表达式（匿名函数）

适用于需要在短时间内创建并使用一次的小型函数。

语法：

```
[capture](parameters) -> return_type { function_body; }
```

- `[capture]`：捕获列表，指定要捕获周围作用域中的外部变量。
  - `[=]`：按值捕获所有可用变量
  - `[&]`：按引用捕获所有可用变量
- `(parameters)`：参数列表，类似于普通函数的参数。
- `-> return_type`：返回类型，若编译器可以推断返回类型则可以省略。
- `{ function_body; }`：函数体，包含执行的代码。

```
int main() {
    auto lambda = [](int a, int b) -> int { return a + b; };
    cout << "Sum: " << lambda(3, 4) << endl;  // 输出: Sum: 7
    return 0;
}

// 按值捕获
int main() {
    int x = 10, y = 20;
    auto lambda = [x, y]() { return x + y; };
    cout << "Sum: " << lambda() << endl;  // 输出: Sum: 30
    return 0;
}

// 按引用捕获
int main() {
    int x = 10, y = 20;
    auto lambda = [&x, &y]() { x += y; };
    lambda();
    cout << "x = " << x << endl;  // 输出: x = 30
    return 0;
}

// 可变Lambda表达式
// 默认情况下，Lambda捕获的变量为常量，使用mutable关键字可以使Lambda捕获的变量可修改，但
```

```
不会影响外部变量
int main() {
    int n = 10;
    auto lambda = [n]() mutable { n += 5; cout << "n = " << n << endl; };
    lambda();  // 输出: n = 15
    cout << "Original n = " << n << endl;  // 输出: Original n = 10
    return 0;
}

//  Lambda表达式作为函数参数
int main() {
    int x = 10, y = 20;
    auto sum = [](int a, int b) { return a + b; };
    cout << "Sum: " << sum(x, y) << endl;  // 输出: Sum: 30
    return 0;
}
```

cmath头文件

```
#include <cmath>
```

函数	描述	示例	返回值
sqrt(x)	计算 x 的平方根	sqrt(16)	4
pow(base, exp)	计算 base 的 exp 次幂	pow(2, 3)	8
abs(x)	计算整数 x 的绝对值	abs(-5)	5
fabs(x)	计算浮点数 x 的绝对值	fabs(-3.14)	3.14
round(x)	四舍五入	round(2.5)	3
exp(x)	计算 e^x	exp(1)	2.71828
log(x)	计算自然对数（以 e 为底）	log(2.71828)	1
log10(x)	计算常用对数（以 10 为底）	log10(1000)	3
sin(x)	计算正弦值（x 为弧度）	sin(3.14159 / 2)	1
cos(x)	计算余弦值（x 为弧度）	cos(0)	1
tan(x)	计算正切值（x 为弧度）	tan(3.14159 / 4)	1
asin(x)	计算反正弦值，返回值为弧度	asin(1)	1.5708
acos(x)	计算反余弦值，返回值为弧度	acos(0)	1.5708
atan(x)	计算反正切值，返回值为弧度	atan(1)	0.7854
hypot(x, y)	计算 sqrt(x^2 + y^2)	hypot(3, 4)	5

随机数

```cpp
#include <iostream>
#include <cstdlib>   // 包含 rand() 和 srand()
#include <ctime>     // 包含 time()
using namespace std;

int main() {
    srand(time(0));  // 使用当前时间作为种子，保证每次运行的结果不同

    cout << rand() % 100 << " ";  // 生成 [0, 99] 范围的随机数

    return 0;
}
```

```cpp
#include <iostream>
#include <random>    // 包含随机数库
using namespace std;

int main() {
    // 创建随机数生成器，使用默认随机设备作为种子
    random_device rd;
    mt19937 gen(rd());  // Mersenne Twister 19937 生成器
    uniform_int_distribution<> dist(1, 100);  // [1, 100] 范围的均匀分布

    cout << dist(gen) << " ";

    return 0;
}
```

std::string

```cpp
#include<iostream>
#include<string>

int main() {
    // 声明并初始化空字符串
    std::string str1;

    // 声明字符串并赋值
    std::string str2 = "Hello String";

    // 声明字符串并使用字符串初始化字符串
    std::string str3 = str2;

    // 声明字符串并使用重复的字母初始化字符串
    std::string str4(5, 'A');
```

```
    // 字符数组和字符串的转换
    const char* chars1 = "Hello";
    std::string str5(chars1);  // str -> char*
    const char* chars2 = str5.c_str();  // char* -> str

    // 获取长度
    int length = str2.length();

    // 拼接
    std::string str6 = str2 + "and" + str3;
    std::string str7 = str2.append("and").append(str3); // 这两个等价

    // 查找
    int pos = str2.find("String");  // 返回值为字串的起始位置，不存在时为-1

    // 替换
    str2.replace(7, 6, "C++");  // 起始位置，长度，目标字符串

    // 截取
    str2 = str2.substr(0, 5);  // 起始位置，长度

    // 比较
    int result = str1.compare(str2);  // 比较的是unicode值
}
```

常用函数

最值

`#include<algorithm>`中的`min_element(st, ed)`返回地址 $[st, ed)$ 中最小值的地址，`max_element(st, ed)`同理。

`nth_element(first, pos, last)`将`[first, last)`区间内的第`pos`个元素放到正确的位置，`[first, pos)`区间内的元素都小于等于`pos`，`[pos, last)`区间内的元素都大于等于`pos`，可以类比快速排序。

大小写转换

`#include<cctype>`中的`islower(char ch)`和`isupper(char ch)`函数用于检查一个字母是否为小写/大写。

`tolower(char ch)`和`toupper(char ch)`将字母转换为小写/大写。

memset

`#include<cstring>`中的`memset(void* ptr, int value, size_t num)`函数可将ptr指向的内存块用value填充num个字节。

范围基 for 循环

```
for (auto &element : container) {
    // auto 会自动推断容器元素的类型
    // & 表示通过引用访问元素，避免复制
}
```

```
#include <iostream>
#include <string>

int main() {
    // 数组
    int arr[] = {10, 20, 30, 40, 50};
    // auto 推断为 int& 类型
    for (auto &num : arr) {
        printf("%d ", num);
    }

    std::string str = "Hello, World!";
    // auto 推断为 char& 类型
    for (auto &ch : str) {
        printf("%c ", ch);
    }

    return 0;
}
```

排序

使用位于`#include<algorithm>`中的`sort`函数（使用快速排序）对元素进行排序，时间复杂度 $O(nlog_2n)$

`sort(first, last)`，first为第一个元素的地址，last为最后一个元素的地址（**不包含**），默认为升序排序。

```
#include <iostream>
#include <algorithm>

int main() {
    int arr[] = {4, 2, 5, 1, 3};
    int n = sizeof(arr) / sizeof(arr[0]);

    std::sort(arr, arr + n);

    for (int i = 0; i < n; ++i) {
        std::cout << arr[i] << " ";  // 1 2 3 4 5
    }

    return 0;
}
```

可传入**比较函数**，实现降序排序：

```
#include <iostream>
#include <algorithm>

bool compare(int a, int b) {
    return a > b;  // 降序
    // 返回true则a在前
}

int main() {
    int arr[] = {4, 2, 5, 1, 3};
    int n = sizeof(arr) / sizeof(arr[0]);

    std::sort(arr, arr + n, compare);

    for (int i = 0; i < n; ++i) {
        std::cout << arr[i] << " ";  // 5 4 3 2 1
    }

    return 0;
}
```

也可以使用lambda表达式实现降序：

```
#include <iostream>
#include <algorithm>

int main() {
    int arr[] = {4, 2, 5, 1, 3};
    int n = sizeof(arr) / sizeof(arr[0]);

    std::sort(arr, arr + n, [](int a, int b) {
        return a > b;
    });

    // 输出排序后的数组
    for (int i = 0; i < n; ++i) {
        std::cout << arr[i] << " ";  // 5 4 3 2 1
    }

    return 0;
}
```

swap

`#include<utility>`中的`swap(T& a, T& b)`函数可以交换传入的两个同类型变量的引用。

STL 标准模板库

仅学习竞赛常用的部分

算法

容器

顺序容器 vector

与数组类似，是连续的顺序存储结构，但是 **长度可变**
数据量巨大时对比普通数组有优势，不容易爆内存

若数组长度可提前确定则应在构造时就指定初始大小，否则当内存耗尽后重新分配内存会浪费时间

构造：

```
// vector<类型> arr(长度, [初值])

// 一维数组
vector<int> arr = {1, 2, 3};     // int数组并指定初值
vector<int> arr(100);     // 初始长100的int数组
vector<int> arr(100, 1); // 初始长100的int数组，初值为1

// 二维数组
vector<vector<int>> matrix(n, vector<int> (m))
vector<int> arr[100];     // 初始100行，不指定列数的二维数组
vector<int> arr[100] {{100, 1}, ... ,{100, 1}}; // 使用列表初始化
```

使用：

```
arr.push_back(elem);  // 在末尾添加元素
arr.pop_back();       // 删除末尾元素
arr[int pos]          // 与数组相同，获取pos位置的元素
arr.size()            // 获取长度
arr.empty()           // 判断是否为空
arr.clear()           // 清空数组
arr.resize(int newlength)  // 修改vector长度
arr.resize(int newlength, elemtype defaultvalue)  // 修改vector长度并指定默认值
// 长度增加时，使用元素类型默认值或指定的默认值填充
// 长度缩短时，截断超出部分
```

容器适配器

- stack 栈
  - 构造：`stack<int> stk;`
  - 进栈：`stk.push(elem);`

    - 出栈：`stk.pop();`
    - 取栈顶：`stk.top();`
    - 不可以通过`stk[int pos]`访问内部元素，栈只能操作栈顶元素
- queue 队列
    - 构造：`queue<int> que;`
    - 进队：`que.push(elem);`
    - 出队：`que.pop();`
    - 取队首：`que.front();`
    - 取队尾：`que.back();`
    - 与栈同理，不可访问内部元素，只能操作队首队尾元素
- priority_queue 优先队列
    - 提供O(1)的最大元素查找，O(logn)的插入与提取
    - 保证每次进行插入删除后，优先级最高的元素总是在队首
    - 构造：`priority_queue<类型，容器，比较器> pque`
        - 容器默认为`vector<类型>`，比较器默认为`less<类型>`即降序排列
        - 设定比较器为`greater<类型>`可实现升序排列
    - 进队：``pque.push(elem);`
    - 出队：`pque.pop();`
    - 取队首：`pque.top();`
    - **仅队首可读**，队中元素和队尾均不可读，且所有元素**不可修改**
    - 若先后进队10，5，20，则出队顺序为20，10，5（less）

关联容器

- set 集合
    - 特点：一个元素仅可能在或不在set中，set中元素无顺序，不可重复，默认按升序排列
    - 由于元素无顺序，故无法用下标索引，仅能用迭代器遍历
    - 构造：`set<int> st;`
    - 使用for循环遍历：`for (auto &ele : st)`
    - 第一个元素：`st.begin();`
    - 最后一个元素的下一个位置：`st.end();`
    - 插入：`st.insert(elem);`
    - 删除：`st.erase(elem);`
    - 查找：`st.find(elem);`，若存在则返回指向该元素的迭代器，否则返回`st.end()`
    - 判断是否存在：`st.count(elem);`

```
#include <iostream>
#include <set>
using namespace std;

int main() {
    // 创建 set 容器
    set<int> mySet = {10, 20, 30, 40, 50};

    // 使用迭代器遍历 set
    for (set<int>::iterator it = mySet.begin(); it != mySet.end(); ++it) {
        // 创建int类型set的迭代器it
        // 赋初值mySet.begin()，每次循环都自增，直到等于mySet.end()
```

```
        cout << *it << " ";  // 输出: 10 20 30 40 50
    }
    cout << endl;

    // 使用auto自动推断迭代器类型
    for (auto it = mySet.begin(); it != mySet.end(); ++it) {
        std::cout << *it << " ";  // 输出: 10 20 30 40 50
    }
    cout << endl;

    // 使用范围for循环遍历set
    for (const auto& val : mySet) {
        std::cout << val << " ";  // 输出: 10 20 30 40 50
    }
    cout << endl;

    return 0;
}
```

- map
  - 一种有序的键值对结构，一个键只能出现一次
  - 构造：`map<键类型, 值类型 [, 比较器]> mp`，比较器默认为`less<类型>`
  - 增/查/改元素：`mp[key] = value;`
    - 若执行`mp[1]`，但mp中没有元素键为1，则会新增键值对，值为默认值0
  - 查元素（返回迭代器）：`mp.find(key);`
  - 删除：`mp.erase(key);`
  - 判断是否存在：`mp.count(key);`

```
// 使用迭代器遍历
for (map<int, int>::iterator it = mp.begin(); it != mp.end(); ++it)
    cout << it->first << ' ' << it->second << endl;

for (auto it = mp.begin(); it != mp.end(); ++it)
    cout << *it << " ";  // 输出: 10 20 30 40 50

// 使用pair访问
for (auto &pr : mp)
    cout << pr.first << ' ' << pr.second << endl;

// 使用键值对遍历
for (auto &[key, val] : mp)
    cout << key << ' ' << val << endl;
```

字符串

```
string s1;           // 空字符串
string s2 = "awa!";  // 赋值awa!
string s3(10, '6');  // 复制10个6，即6666666666
```

比较两个string是否相同可以直接使用==，也可以使用.compare(str)

尾接字符串尽量使用+=而不是+，因为+会创建新的字符串并赋值，而+=会原地操作

对 二元组

pair<第一个值类型，第二个值类型> pr

```
pair<int, int> p1;
pair<int, long long> p2;
pair<char, int> p3 = {'a', 1};

// 使用first和second取值
char key = p3.first;
int value = p3.second;
```

迭代器

对如树、集合这样非线性的数据结构，没有下标，因此使用 **迭代器** 进行遍历

vector<int>::iterator it

- 头迭代器：.begin()
- 尾迭代器：.end()
  - end指向的是最后一个元素的下一个位置，是无意义的值
- 前一个迭代器：prev(it)
- 后一个迭代器：next(it)