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
}
```

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 $O(nlog_2 n)$

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;
}
```

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。