输入输出

```cpp
#include <bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f
#define endl '\n'

signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);    // 关闭同步流
    cout << fixed << setprecision(10);           // 保留小数点后10位

    int a, b;
    cin >> a >> b;
    cout << a + b << endl;

    cout << 3.14 << endl;

    int arr[3] = {1, 2, 3};
    for (int i = 0; i < 3; i++)
        cout << arr[i] << " \n"[i == 2];      // 中间用空格隔开, 最后一个数换行

    cout.flush();    // 刷新缓冲区
    return (0 ^ 0);
}
```

`getline` `stringstream`

```cpp
string s;
while (getline(cin, s)) {
    stringstream ss(s);
    string tmp;
    while (ss >> tmp) {
        cout << tmp << ' ';
    }
    cout << endl;
}
/*
1 2 3 4
5 6 7 8 9
*/
```

```
string s;
while (getline(cin, s)) {
    stringstream ss(s);
    string tmp;
    while (getline(ss, tmp, ',')) {
        cout << tmp << ' ';
    }
    cout << endl;
}
/*
1,2,3,4
5,6,7,8,9
*/
```

讲义写的比较简略, 结合这个博客看看

《黑马》——C++核心编程_Netfishless的博客-CSDN博客

《黑马》——C++提高编程Netfishless的博客-CSDN博客黑马c++提高

引用

- 给变量起别名
- 必须初始化，初始化后不能改变
- 引用做函数参数，简化指针修改实参，避免拷贝
- 引用作为函数返回值，不要返回局部变量的引用

引用的本质

```
int a = 10;
int& ref = a;    // 自动转换为 int* const ref = &a; 指针常量是指针指向不可改，也说明为什么引用不可更改
ref = 20;        // 内部发现ref是引用，自动帮我们转换为: *ref = 20;
```

常量引用

```
// 防止误操作
struct Node {
    int a, b, c;
};
void func(const Node& B) {
    /* ... */
}
```

函数

函数默认参数

```
void build(int u, int l = 1, r = 10000) {
    if(l == r)
        return;
    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1, mid + 1, r);
}
build(1);
```

函数重载

- 同一作用于下
- 函数名称相同
- 函数参数**类型不同** 或者 **个数不同** 或者 **顺序不同**
- 函数的返回值不可以作为函数重载的条件

引用作为重载条件

```
void func(int& a) {
    cout << "func (int &a) 调用 " << endl;
}

void func(const int& a) {
    cout << "func (const int &a) 调用 " << endl;
}
signed main() {
    int a = 10;
    func(a);     // func (int &a) 调用
    func(10);    // func (const int &a) 调用

    return (0 ^ 0);
}
```

类和对象

- 成员变量
- 成员函数
- 构造函数

模板

模板就是建立**通用的模具**，大大**提高复用性**

函数模板

```cpp
template <class T>
T add(T a, T b) {
    return a + b;
}
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    cout << add(3, 4) << endl;
    cout << add(3.14, 2.718) << endl;

    cout << add<int>(1, 2) << endl;   // 显式指定

    return (0 ^ 0);
}
```

template — 声明创建模板

class — 表面其后面的符号是一种数据类型，可以用typename代替

T — 通用的数据类型，名称可以替换

类模板

- 类模板没有自动类型推导的使用方式（c++17后增加了类模板类型推导）
- 类模板在模板参数列表中可以有默认参数

```cpp
template <class T>
struct Point {
    T x, y;
    Point(T x, T y) : x(x), y(y) {}
};
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    Point<int> a(2, 4);
    Point<double> b(3.14, 2.718);

    Point c(3, 6);

    return (0 ^ 0);
}
```

一些库函数

`swap()`

`min()` `max()`

`abs()`

`sqrt()`

`sin()` `cos()` `tan()` `asin()` `acos()` `atan()` `atan2()`

`max_element()` `min_elenment()`　返回的是迭代器

`prev_permutation()` `next_permutation()`　全遍历$O(n!)$，单次最坏$O(n)$

`__gcd()`　c++17后可直接使用 `gcd()` `lcm()`

`__lg()`

`reverse()` 翻转

`rotate(beg, newbeg, end);`

`srand(time(0))` `random_shuffle()` 打乱

`memset()` 以字节为单位赋值

`fill()` 赋值为同一个值

`iota()` 赋值为连续的值

`find()` `find_if()`

`count()` `count_if()`

`accumulate()` 求和

`nth_element` $O(n)$求第k小

```
bool cmp(int a, int b) {
    return a > b;
}
int main() {
    int a[9] = {4, 7, 6, 9, 1, 8, 2, 3, 5};
    int b[9] = {4, 7, 6, 9, 1, 8, 2, 3, 5};
    int c[9] = {4, 7, 6, 9, 1, 8, 2, 3, 5};
    nth_element(a, a + 2, a + 9);
    //将下标为2，也就是第3个数放在正确的位置
    //也就是求的是第3小
    cout << "第3小是：" << a[2] << endl;

    //那么求第3大，就是求第9-3+1小,即第7小
    //也就是将下标为6的第7个数，放在正确的位置
    nth_element(b, b + 6, b + 9);
    cout << "第3大是：" << b[6] << endl;

    nth_element(c, c + 2, c + 9, cmp);
    // nth_element(c, c+2, c+9, greater<int>());
    cout << "第3大是：" << c[2] << endl;
}
```

`set`操作

```
set<int> a, b;      vector<int> c;
set_union(a.begin(), a.end(), b.begin(), b.end(), back_inserter(c));        //并集
set_intersection(a.begin(), a.end(), b.begin(), b.end(), back_inserter(c)); //交集
set_difference(a.begin(), a.end(), b.begin(), b.end(), back_inserter(c));   //差集 (A - AB)
```

运算符重载及*sort*

```cpp
struct Node {
    int u, v, w;
    bool operator<(const Node& B) const {
        return w < B.w;
    }
};
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    vector<Node> a(10)
    for (int i = 0; i < 10; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        a[i] = {u, v, w};
    }
    sort(a.begin(), a,end());

    // 另一种写法
    sort(a.begin(), a.end(), [&](const Node& A, const Node& B) {
        return A.w < B.w;
    });

    return (0 ^ 0);
}
```

离散化

`unique`

```cpp
vector<int> v = {2, 2, 1, 2, 5, 4, 6, 3, 2, 3, 4};
sort(v.begin(), v.end());
auto it = unique(v.begin(), v.end());     // 并不会把重复的删掉，只是放到了后面
cout << int(it - v.begin()) << endl;
for (auto& x : v)
    cout << x << ' ';
```

```cpp
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);

    vector<int> a = {432435, 32423, 54353489, 234234, 6546343, 32423, 54353489};
    auto v = a;
    sort(v.begin(), v.end());       // 32423 32423 234234 432435 6546343 54353489 54353489
    v.erase(unique(v.begin(), v.end()), v.end());  // 32423 234234 432435 6546343 54353489
    auto get = [&](int x) {
        return lower_bound(v.begin(), v.end(), x) - v.begin();
    };
    for (int i = 0; i < a.size(); i++)
        a[i] = get(a[i]);
    for (int i = 0; i < a.size(); i++)
        cout << a[i] << " \n"[i == a.size() - 1];
```

```
    for (int i = 0; i < a.size(); i++)
        cout << v[a[i]] << " \n"[i == a.size() - 1];

    return (0 ^ 0);
}
```

迭代器

迭代器不是指针，是类模板，表现得像指针。它只是模拟了指针的一些功能，通过重载了指针的一些操作符`->` `*` `++` `--`等封装了指针，是一个“可遍历STL容器内全部或部分元素”的对象，本质是封装了原生指针。

迭代器返回的是对象的引用。

`容器名::iterator`

string

`构造函数`

`赋值`

`拼接`

`find()`

`string::npos` 无符号数

`比较`

`提取`

`插入和删除`

`substr()`

`c_str()`

`to_string()` 数字转换为`string`

`stoi()` `stod()` `string`转化为`int` / `double`

vecotr

`构造函数`

`赋值`

`容量和大小`

`插入和删除`

`数据存取`

`内存`

```
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    vector<int> v = {1, 2, 3};
    cout << &v << ' ' << &v[0] << ' ' << v.data() << endl;

    int a = 1;
    cout << &a << endl;
}
```

动态扩容

```
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    vector<int> v;
    for (int i = 1; i <= 30; i++) {
        v.push_back(i);
        cout << i << ' ' << v.capacity() << endl;    // g++扩容因子为2, 即每次乘2
        cout << v.data() << endl;
    }

    vector<int> a(300);
    cout << a.capacity() << endl;
    a.clear();
    cout << a.capacity() << endl;

    return (0 ^ 0);
}
```

`预留空间reserve`

array

`可排序`

deque

`构造`

`大小操作`

`头部和尾部插入和删除`

stack

需要用时可用*vector*替代

queue

`构造`

`数据存取`

`大小操作`

priority_queue

自定义排序

```cpp
priority_queue<int, vector<int>, greater<>> q;    // 小根堆

// 或者自己重载小于号
struct Node {
    int x;
    bool operator<(const Node& B) const {
        return x > B.x;   // 注意
    }
}
priority_queue<Node> q;
```

set/multiset

- 所有元素都会在插入时自动被排序
- set不允许容器中有重复的元素
- multiset允许容器中有重复的元素

仿函数

```cpp
// less的定义
template<typename _Tp> struct less : public binary_function<_Tp, _Tp, bool> {
      bool operator()(const _Tp& __x, const _Tp& __y) const
      { return __x < __y; }
};

// set 的申明
template<typename _Key, typename _Compare = std::less<_Key>,typename _Alloc =
std::allocator<_Key>> class set;
```

使用*set*自带的：`lower_bound()` `upper_bound()`

自定义排序

```cpp
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    auto cmp = [&](const int& a, const int& b) {
        return a > b;
    };
    set<int, decltype(cmp)> st(cmp);
    st.insert(1);
    st.insert(2);
    cout << *st.begin() << endl;

    return (0 ^ 0);
}
```

pair

构造 `make_pair(x, y)`

可排序 先比较第一个元素, 再比较第二个

map/multimap

- map中所有元素都是pair
- pair中第一个元素为key（键值），起到索引作用，第二个元素为value（实值）
- 所有元素都会根据元素的键值自动排序

自定义排序

```cpp
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    auto cmp = [&](const int& a, const int& b) {
        return a > b;
    };
    map<int, int, decltype(cmp)> mp(cmp);
    mp[1] = 10;
    mp[2] = 20;
    cout << mp.begin()->second << endl;

    return (0 ^ 0);
}
```

unordered

底层为哈希表，本身常数较大，且会被卡，不推荐使用

bitset

```cpp
/* 构造 */
bitset<4> b;               // 定义一个4位的bitset
b = 10;                    // 将其赋值为10, 即1010
bitset<8> b1(100);         // 用十进制整数构造
bitset<10> b2("1010101");  // 用01字符串构造
/* bitset可直接进行位运算操作 */

/* 与其他类型的转换 */
bitset<10> bit("1010001");
string s = bit.to_string();              // 0001010001
unsigned long x = bit.to_ulong();        // 81
unsigned long long y = bit.to_ullong();  // 81

/* 像数组一样访问修改 */
bitset<8> b("1010");
cout << b << endl;  // 00001010
b[2] = 1;
cout << b << endl;  // 00001110

/* 提供了一些函数函数 */
bitset<20> b("1010101");
```

```
22
23 b.flip();   // 翻转所有位
24 b.flip(i);  //翻转 b[i]
25
26 b.set();   // 所有位置1
27 b.set(i);  // b[i] = 1;
28
29 b.reset();   // 所有位置0
30 b.reset(i);  // b[i] = 0;
31
32 cout << b.size() << endl;          // bitset的大小, 20
33 cout << b.count() << endl;         // 1的个数
34 cout << b.any() << endl;           // 是否有1, true
35 cout << b._Find_first() << endl;  // 返回lowbit
36 cout << b.all() << endl;           // 是否全为1, false
```

$\_\_int128\_t$

达到10^{38}，需要自己写读写操作

```
1  using i128 = __int128_t;
2  istream& operator>>(istream& is, i128& n) {
3      n = 0;
4      string s;
5      is >> s;
6      for (auto c : s) {
7          n = 10 * n + c - '0';
8      }
9      return is;
10 }
11 ostream& operator<<(ostream& os, i128 n) {
12     if (n == 0) {
13         return os << 0;
14     }
15     string s;
16     while (n > 0) {
17         s += '0' + n % 10;
18         n /= 10;
19     }
20     reverse(s.begin(), s.end());
21     return os << s;
22 }
```

随机数

产生高质量随机数

```
1  using ll = long long;
2  mt19937 mt(random_device{}());
3  uniform_int_distribution<int> range_i(1, 10);                    // [1, 10]int
4  uniform_int_distribution<ll> range_ll(1, 1000000000000000000);  // [1,
   1000000000000000000] long long
5  uniform_real_distribution<double> range_d(1, 10);                // [1, 10]double
6  int rnd_i() {
```

```
    return range_i(mt);
}
ll rnd_ll() {
    return range_ll(mt);
}
double rnd_d() {
    return range_d(mt);
}
signed main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    for (int i = 1; i <= 20; i++) {
        cout << rnd_i() << ' ' << rnd_ll() << ' ' << rnd_d() << endl;
    }

    return (0 ^ 0);
}
```

测试程序运行时间

```
auto t1 = std::chrono::steady_clock::now();    // 起始时间点
// run code 你的代码
auto t2 = std::chrono::steady_clock::now();    // 终止时间点

//秒
double dr_s = std::chrono::duration<double>(t2 - t1).count();
//毫秒级
double dr_ms = std::chrono::duration<double, std::milli>(t2 - t1).count();
//微妙级
double dr_us = std::chrono::duration<double, std::micro>(t2 - t1).count();
//纳秒级
double dr_ns = std::chrono::duration<double, std::nano>(t2 - t1).count();
```