2022 香农先修班第一次课

C++入门

语法基础

这次课只介绍与算法相关的 C++ 知识，写算法用得很少的知识(如 `try-catch`, 类)不予介绍。

基本概念

C++ 是 C 的超集，这意味着所有 C 的语法都能直接用于 C++。

C++ 同 C 一样，都分为多个版本。一般而言越新好用的新语法越多。鉴于绝大多数比赛和平台都支持的 C++11，而更新的不一定支持，所以主要使用 C++11。具体如何配置自行百度。

> 推荐的开发环境首选 vscode(即visual studio code蓝色图标的)，自行了解。可以自行下载好用的插件。vscode 两大好处是边写边报错和自动格式化。

C++ 的文件后缀一般是 `.cpp`(还有别的后缀，但最常用是这个)。头文件后缀可以是 `.hpp`。

C++ 的优势是具有大量的 STL(标准模板库)，提供很多内置的库函数和数据结构，所以我们推荐使用 C++ 而不是 C。在算法竞赛里，对效率要求很严格，因为 Python 和 Java 是解释型语言，效率低，所以不推荐使用。其他语言通常并不是所有算法竞赛都支持。(至少 C++ 按理是每个算法竞赛都会支持的)

万能头文件

C++ 的一大优点是，可以只引用一个头文件，称为万能头文件，即：

```
#include<bits/stdc++.h>
```

如果你打开该文件的源码，会发现它的本质就是把大部分库都 `#include` 了一次。(源码请见这里)

> 优点显而易见，就是可以不用记忆各个函数在哪个头文件，也不用写一堆 include。
>
> 有一些潜在但通常无伤大雅的缺点：
>
> - 因为把所有库都调了一边，所以自带很多常量和函数名，容易重名。典型的重名有：`copy,sort,x1,y1,x0,y0,xn,yn,prev,size,merge` 等。
>
> 同时，因为需要加载大量头文件，效率有些微下降(可忽略不计)。
> - 万能头不是GNU C++标准头文件，并不是所有 OJ 都支持万能头(已知 POJ 不支持)。
> - 部分开发环境(如紫色的visual studio)没有该头文件，需要自己装。

命名空间

为了避免重名冲突，C++ 的库函数一般是不能直接调用的，而是定义了一个叫做命名空间的前缀，需要使用作用域解析运算符才能调用。默认命名空间名字叫 `std`(即 standard)。所以对一个函数，调用方法是 `std::函数()`。

例如，已知 C++ 有一个名为 `sort` 的函数，调用的代码举例：

```
int a[3]={3,2,1};
std::sort(a,a+3); //调用函数
printf("%d %d %d",a[0],a[1],a[2]);
```

如果需要大量调用库函数，显然这样会徒增很多码量。为了能够像 C 一样调用库函数，需要声明使用命名空间 `std`：

```
using namespace std;
```

连起来的完整代码为：

```
#include <bits/stdc++.h> //万能头
using namespace std; //命名空间
int a[3] = {3, 2, 1};
int main()
{
    sort(a, a + 3); //调用库函数
    printf("%d %d %d", a[0], a[1], a[2]);
    return 0;
}
```

> 关于其他命名空间和自定义命名空间，不讲，感兴趣自学。sort 函数预计会在后续课程详细讲解，这里从略。
>
> 部分算法选手喜欢写成 `signed main()` 或 `int32 main()`，其实是一样的意思。这样写是以防如果需要把全文 `int` 替换成 `long long`，即使用 `#define int long long` 时，因为不允许 `main` 函数的返回值是 `long long`，所以需要让 `main` 不被替换而采取的策略。 如果没有空间严格限制，可以推荐用 `long long` 取代全部 `int`，视个人喜好决定是否遵循该推荐。
>
> 习惯上，数组变量放全局。这是因为每个函数的内存上限约为 2MB。而全局无限制。如果不放全局的静态数组大于 2MB 就会 RE。而且全局能够自动各元素初始化为 0。而且事实上习惯每个数组都比所需大小开多几个元素。
>
> 对数组，有两种代码风格，一种是从下标 0 开始使用，另一种是从下标 1 开始使用。看个人喜好。

输入输出

我们知道，在 C 语言，输入和输出不同类型的变量需要使用不同的占位符。但是在 C++ 可以统一使用一种格式。

> C++ 可以完全用 C 的输入输出，下文只是另一种新的方法，不强制使用。

> 如果对输入输出整体不理解，可以推荐拓展阅读。并请注意在 OJ 输入和输出是可以同步进行的，如你可以读入第一行再输出一些东西再读入第二行再输出，不必全部读入再全部输出……最终只需要你输出的内容合起来与答案一致即可

输入

cin

使用`cin`函数输入一个变量，表达式是`cin>>变量名`。可以连写表示输入多个变量，如`cin>>x1>>x2;`，等效于`cin>>x1;cin>>x2;`。变量类型不同其等效表达式也不同：

- 各种整型与浮点型：等效于`scanf`
- 字符数组(char*)：等效于`scanf`

- 字符(char)：不等效于任何一个 C 语言函数。会读取第一个输入流的非空(非空白回车等)字符。

> 算法竞赛里无论 C/C++，除非对I/O流很熟悉，否则不建议读取 `char`，一般当成字符数组/字符串读入然后取首字符。

`cin>>变量名` 这个表达式本身会返回一个布尔值，代表当前是否读取到 EOF；为真表示遇到了 EOF。

使用举例：

```
#include <bits/stdc++.h>
using namespace std;
int x;
double y;
char s[5];
int main()
{
    cin >> x >> y >> s;
    printf("%.2lf%c", x + y, s[0]);
    return 0;
} /*input:5 0.8 x output:5.80x*/
```

```
#include <bits/stdc++.h>
using namespace std;
int v, s;
signed main()
{
    while (cin >> v)
    { //类比EOF!=scanf("%d",&v)
        s += v;
    }
    cout << s;
    return 0;
}
```

特别提一下，有一个函数为 `cin.ignore()`，作用是读走一个输入流字符，等效于 `getchar()`。

getline

如果要读取一整行内容，一种方法是使用 `cin.getline` 成员函数。一种语法如下：

```
cin.getline(字符数组变量名, 读取长度, 终止字符)
```

将终止字符设置为 `\n`，那么读取到的长度是从当前输入流到 `\n` 前(不含 `\n`)的全部内容。如：

```cpp
#include <bits/stdc++.h>
using namespace std;
char s[105];
int main()
{
    cin.getline(s, 100, '\n');
    printf("%d", strlen(s));
    return 0;
} /*input:hello world output:11*/
```

> 还有 `getline` 函数。下文叙述。
>
> 更多函数，例如 `peek`，用处不是特别大，感兴趣可自行了解。

顺便提一下，`gets` 函数是被 C11 和 C++11 等标准禁用了的，请使用 `fgets` 或 `cin.getline` 代替。

> 同样被高版本(不一定是11，但有的更高的会禁用)禁用的功能还有：`register` 和 `random_shuffle` 等，建议有使用这些语法的尽量改掉。

输出

使用 `cout` 函数输出一个表达式。格式为 `cout<<表达式`。注意 `<<` 本质是位运算，为避免歧义和优先级问题，所以含位运算、四则运算的表达式可能需要加括号。具体优先级表自查。

输出的语法与输入是类似的。一些细节：

- 输出整数、字符、字符数组跟最普通的 `printf` 一样
- 输出浮点数，默认六位**有效数字**(注意不是小数点后六位)，大于六位数用指数输出

```cpp
#include <bits/stdc++.h>
using namespace std;
double x, y;
int main()
{
    cin >> x >> y;
    cout << x << '+' << y << "=" << (x + y) << "\nI'm smart!";
    return 0;
} /*input:2333333 4.0 output(1st line):2.33333e+06+4=2.33334e+06*/
```

> 你也许听过 `endl` 可以“代替” `\n`，但强烈不推荐使用。

> 如果要输出特定位小数，如保留小数点后九位，建议使用 C 语言的 `printf`。C++ 的 `cout` 能做，但是相比之下更复杂。感兴趣自行百度。

读写加速

通常情况下，C 语言 `printf/scanf` 的极限约为每秒 10^6 个非数组变量。但未加优化 C++ 的 `cout/cin` 的极限才约为 2×10^5，为了加速到接近 C 语言速度，通常需要加上三行代码：

```cpp
ios::sync_with_stdio(false);//false即0,是C++的布尔值bool类型
cin.tie(0); //0即空指针，可以写成C++的nullptr或C语言的NULL
cout.tie(0);
```

如：

```cpp
#include <bits/stdc++.h>
using namespace std;
int x;
int main()
{
    ios::sync_with_stdio(false), cin.tie(0), cout.tie(0);
    cin >> x;
    cout << x * x;
    return 0;
}
```

使用了读写加速后，不能同时使用 `cin,scanf` 与 `cout,printf`，否则可能出错

> 想要比 10^6 更快，需要手写读入/输出函数，称为快读/快写，一般用不上，感兴趣自学。

传引用

> 我们知道，在 C 语言，有传址与传值之分，例如交换一个变量可以写成：
>
> ```cpp
> #include <stdio.h>
> void myswap(int *x, int *y)
> {
>  int t = *x;
>  *x = *y;
>  *y = t;
> }
> int main()
> {
>  int x = 580, y = 1437;
>  myswap(&x, &y);
>  printf("%d %d", x, y);
> }
> ```
>
> 显然，这样写太麻烦了。而在 C++，有对传值的简化写法。

使用 `变量类型& 传引用变量名` 定义一个传引用变量(称为左值传引用)。作用跟指针类似，但不需要指针语法，只需要按一般变量语法使用即可。

> 有右值传引用 `&&`，基本用不上，感兴趣自学。

赋值一个传引用变量，直接把变量赋给它即可，即 `传引用变量名 = 被引用变量名`。声明时必须赋值，且不能改变初始化再指向其他对象。

传引用可以视作一个变量的别名。即调用传引用等效于调用被引用变量本身。如：

```cpp
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int x = 114515;
    int &y = x;
    --y;
    printf("%d %d", y, x);//都是114514
    return 0;
}
```

因此可以在函数里，直接用传引用实现传址功效。如：

```cpp
#include <bits/stdc++.h>
using namespace std;
void myswap(int &x, int &y)
{
    int t = x;
    x = y;
    y = t;
}
int main()
{
    int a = 233, b = 666;
    myswap(a, b);
    cout << a << ' ' << b; // 666 233
    return 0;
}
```

> 事实上，C++ 内置了 `swap` 函数，可以直接调用来实现上述功能。

传引用的功能是代替指针。特别注意不能传引用一个数组。

使用传引用的目的：

- 修改值
- 对很大的数据对象(结构体)，提高运行速度，避免拷贝

> 对于认为不需要修改值的用途里，可以写成 `const 类型& 变量`。这在重载比较函数里很常用。

结构体

基本使用

与 C 语言的结构体不同，C++ 的结构体有一些新的特点。

首先，可以更简单的声明(不需要使用 typedef 等一堆东西)。格式：

```cpp
struct 结构体名字 {结构体定义};
```

可以在声明结构体时马上再定义该结构体的变量，也可以单独定义，格式分别是：

```
struct 结构体名字 {结构体定义} 变量名1;
结构体名字 变量名;
```

在定义时，可以用大括号法依次给成员赋值。即：

```
变量名 = {成员1值, 成员2值, ... 成员n值};
```

当然也可以像 C 语言那样在后续再赋值。如：

```
#include <bits/stdc++.h>
using namespace std;
struct node
{
    int x, y;
} a = {2, 3};
node b = {4, 5}, c;
void printnode(node x)
{ //对于比较大的结构体，可以node& x提高效率避免复制
    printf("(%d, %d) ", x.x, x.y);
}
int main()
{
    c.x = 6;
    c.y = 7;
    printnode(a);
    printnode(b);
    printnode(c);
    return 0;
}
```

方法

与 C 语言不同，C++ 的结构体可以还能定义成员函数(称为方法)。如果学过面向对象，可以认为 C++ 的结构体是默认全是公有成员的类，每个结构体变量是实例。

方法的定义和调用跟一般函数是类似的，只不过需要多用一次 `.` 直接成员运算符或 `->` 间接成员运算符(如果用了结构体指针，一般很少用)。

方法可以不使用成员运算符直接调用自己的成员属性。

如：

```
#include <bits/stdc++.h>
using namespace std;
struct node
{
    int x, y;
    void print()
    {
        printf("(%d, %d) ", x, y);
    }
    node add(node r) //不修改r的话建议写成const node& r
```

```
    {
        return {x + r.x, y + r.y};
    } //大括号是定义结构体的一种方法
} a = {2, 3}, b = {4, 5}, c = {6, 7};
int main()
{
    a.print();
    b.print();
    c.print();
    node d = a.add(c);
    d.print();
    (a.add(b)).print();
    return 0;
}
```

> 上文提到的 `cin.getline` 函数，实际上就是 `cin` 这个类的 `getline` 方法。
>
> 对比参数 `node x`，`node& x` 与 `const node& x`：
>
> 1. `node x` 需要复制一遍 x 作为参数，传值；x 可以是变量或常量、表达式
> 2. `node& x` x 一定变量，不能是常量或表达式；传址。
> 3. `const node& x` 传址，不能修改 x，只能读取。可以是变量、常量、表达式。
>
> 如：(上述代码为例)
>
> ```
> (a.add(b)).print(); //1,2,3可以(2,3传址)
> (((node){3, 4}).add({1, 2})).print(); //1,3可以(3传址)
> ```

重载运算符

对于基本运算符，如关系运算 `+-*/%`，比较运算 `!=,>`，位运算等，可以给结构体重载，使得结构体支持直接使用该运算。有两种方法，一种是在结构体内写方法：

```
返回值 operator 运算符(该运算符的右参数) //可能没有参数
{
    函数体
}
```

一种是定义函数：

```
返回值 operator 运算符(左参数, 右参数)
{
    函数体
}
```

对于二元运算符，如 `+`，一定有左右参数；对一元运算符如 `++`，没有右参数。

举例：(写法一：方法)

```
#include <bits/stdc++.h>
using namespace std;
struct node
```

```
{
    int x, y;
    void print()
    {
        printf("(%d, %d) ", x, y);
    }
    node operator+(const node &r)
    {
        return {x + r.x, y + r.y};
    }
} a = {2, 3};
int main()
{
    node c = a + a;
    c.print();
    (a + (node){10, 10}).print();
    ((node){100, 100} + (node){10, 10}).print();
    return 0;
}
```

(写法二：函数)

```
#include <bits/stdc++.h>
using namespace std;
struct node
{
    int x, y;
    void print()
    {
        printf("(%d, %d) ", x, y);
    }
} a = {2, 3};
node operator+(const node &l, const node &r)
{
    return {l.x + r.x, l.y + r.y};
}
int main()
{
    node c = a + a;
    c.print();
    (a + (node){10, 10}).print();
    ((node){100, 100} + (node){10, 10}).print();
    return 0;
}
```

> 事实上，`cin >>` 和 `cout <<` 的本质就是 `cin,cout` 两个类的重载函数。

> 重载比较函数，建议写成：
>
> ```
> bool operator < (const node& r) const
> {
>  函数体
> }
> ```

> 其中前者 `const` 表示右操作数不会被改变；后者表示左操作数。
>
> 或：

```
bool operator < (const node&l, const node& r)
{
 函数体
}
```

下面再介绍三个比较好用的特性：

auto

语义与 C 语言完全不一样。在 C++ 的语义代表智能判定表达式的类型。如：

```
auto x = 1;   //是int类型
auto y = 2LL; //是long long类型
auto z = 3.0; //是double类型
int a1 = 5, a2 = 6;
auto w = a1 + a2; // int
```

同理，可以判定结构体，如对上文程序，可以写：

```
auto c = a + a; //原文是node c = a + a;
```

好处是，遇到很多名字很长的类型名时，可以少写点东西。这在下文很有用。

> 注意：auto 不能用于判定不出类型的情形(有歧义)，如函数的参数。

for-each

对于一个可迭代的变量，如数组，可以用 `for-each` 表达式不取下标直接依次把每个元素值调用出来，格式：

```
for(类型 变量名 : 可迭代变量)
{
    表达式;
}
```

如：

```
int x[5] = {2, 4, 6, 8, 10};
for (int v : x)
{
    printf("%d ", v); //2 4 6 8 10
}
```

```
for (auto i : {-1, 0, 1, 2}) //即int i
{
    printf("%d\n", i);
}
```

> 缺点是运行速度略慢于一般的 for，但是写起来比较简洁。
>
> 在高版本(如C++17和更高)还可以做多元素迭代，如：
>
> ```
> #include <bits/stdc++.h>
> using namespace std;
> struct node
> {
>  int x, y;
> } a[] = {{1, 2}, {3, 4}, {5, 6}};
> int main()
> {
>  for (auto [x, y] : a)
>  {
>      printf("%d %d %d\n", x, y, x + y);
>  }
>  return 0;
> }
> ```

匿名函数

有一些函数，不需要递归，用得很少，或者很简单，有时往往不希望定义为全局的，导致阅读起来代码结构很乱。这时候可以定义局部函数，即匿名函数。如果把定义结果赋值给一个变量，就成了函数变量。格式：

```
[](参数列表){函数体}; //匿名函数
auto 函数变量名 = [](参数列表){函数体}; //函数变量
```

默认不允许在函数体使用全局变量，若需要允许，把 `[]` 改成 `[&]`。

调用函数变量与调用函数是一样的。

如：

```
#include <bits/stdc++.h>
using namespace std;
signed main()
{
    auto f = [](int x, int y)
    { return x + y; };
    printf("%d %d", f(1, 2), f(3, 4));
    return 0;
}
```

应用举例：对 `sort` 函数，重定义排序规则，将默认升序改为降序排序。`sort` 函数可以传入第三个参数，类型是函数变量，表示比较依据(返回 `true` 代表排前面)。

```cpp
#include <bits/stdc++.h>
using namespace std;
int a[] = {1, 100, 10};
signed main()
{
    sort(a, a + 3, [](int x, int y)
         { return x > y; });
    printf("%d %d %d", a[0], a[1], a[2]);
    return 0;
}
```

> 事实上，C++ 内置了这样的函数变量，需要逆序即使用 `greater<类型>()` 即可获得该函数变量。升序同理 `less<类型>()`。如上述代码改为：

```cpp
sort(a, a + 3, greater<int>());
```

> 事实上递归函数也可以写匿名，但是比较麻烦，而且是 C++14 和更高版本才支持的，例如：

```cpp
#include <bits/stdc++.h>
using namespace std;
int main()
{
 auto print = [](auto self, int v) -> void
 { //函数功能：输出正数,->void是返回值声明为void
     if (v >= 10)
     {
         self(self, v / 10);
     }
     putchar((v % 10) + '0');
 };
 print(print, 1437581);
 return 0;
}
```

> 顺便提一个重要的友情提示，每个函数(包括main函数)的允许空间大小为 2MB，所以对比较大的静态数组请设置为全局变量，否则可能会运行出错

模板类

上文定义了 `int` 的 node，如果需要定义其他类型的，又要重写一遍 node，非常的麻烦。为了具有更高的通用性，可以只定义一个 node，使得所有类型都通用，这种思想就是模板。例如，用模板类(这里准确来说是模板结构体)我们可以改造为：

```cpp
#include <bits/stdc++.h>
using namespace std;
template <typename t>
struct node
{
    t x, y;
    void print()
    { //因为不知道类型，所以不能printf，占位符未知
        cout << '(' << x << ' ' << y << ") ";
```

```
    } //事实上cout<<就是一个模板方法
    node<t> operator+(node<t> r)
    {
        return {x + r.x, y + r.y};
    }
};
int main()
{
    node<int> x = {1, 2}, y = {3, 4};
    (x + y).print();
    node<double> z = {0.5, 0.6};
    (z + node<double>{0.1, 0.2}).print();
    return 0;
}
```

可以看到，我们只定义了一个 node，却可以用很多个类型的数据。

语法简要解释：

- `template <typename t>` 表示下面的结构体为模板，不确定类型的成员的类型抽象为 `t`。(即类型本身是变量 `t`。)
- `t x,y;` 是定义了两个成员属性，其类型是不确定的 `t`。
- `node<t>` 表示该 `node` 结构体所使用的类型值是 `t`。
- `node<int>` 声明了一个具体的 `node` 结构体变量，即 `t=int`。

对 C++ 入门来说，只需要掌握模板的使用即可，不需要掌握定义。即 `main` 函数内的部分要求掌握，`node` 的声明能理解即可。在下文 STL 会出现大量模板的使用。

STL

STL 即 standard template library，标准模板库。顾名思义，定义了很多模板类及其相关的函数的一系列库的即可。

也就是说，STL 所提供的全部类都是模板类。

> 下文会出现复杂度的描述。因时间有限，本节课不讲复杂度，可在后续课程学完复杂度后倒回来再看看这部分内容。

vector

定义和使用

直译是向量(即线性代数的单行矩阵)，即一维数组。是动态的。即不定长数组。

常用定义方法有：

```
vector<数据类型>变量名;
vector<数据类型>变量名{元素1, 元素2, ..., 元素n};
vector<数据类型>变量名 (长度);
vector<数据类型>变量名 (长度, 初始值);
vector<数据类型>变量名 = 变量名2; //复制构造
vector<数据类型>变量名(变量名2); //同上
vector<数据类型>变量名(数组首地址, 数组尾地址);//复制[首,尾)
```

可以用 `for-each` 来简单地输出一个 vector 的所有元素。也可以用 `[]` 运算符，将其当做数组来使用。如：

```cpp
#include <bits/stdc++.h>
using namespace std;
vector<int> v1{1, 1, 4, 5, 1, 4};
vector<int> v2(5, 1);
vector<int> v3 = v1;
vector<int> v4(v1);
int a[] = {1, 4, 3, 7};
vector<int> v5(a, a + 4);
void print(vector<int> &v) //不复制，提高效率
{
    for (int x : v)
    {
        printf("%d ", x);
    }
    printf("\n");
}
signed main()
{
    v3[0] = 1919810;
    v4[1] = 666;
    print(v1);
    print(v2);
    print(v3);
    print(v4);
    print(v5);
    return 0;
}
```

常用方法

- `size()` 返回 `size_t` 类型(类似 `unsigned` 整型)当前有几个元素。$O(1)$

  因为其是 `unsigned` 的，所以谨慎使用 `size()-1` 以免 0-1 得到错误。

- `resize(int size, value=0)` 改变大小并重新初始化。

  即有两个方法，一个是 `resize(int size)`，一个是 `resize(int size, value)`。前者默认 `value=0`。

- `push_back(value)` 插入一个元素到最末，size 加一。均摊 $O(1)$

  > emplace_back是push_back的优化版，区别如下:(给看得懂的人看)
  >
  > emplace_back() 在容器尾部添加一个元素，这个元素原地构造，不需要触发拷贝构造和转移构造。而且调用形式更加简洁，直接根据参数初始化临时对象的成员。(当然int之类的也能用)是C++11的新特性。

- `pop_back()` 删除数组最后的元素， size 减一。$O(1)$

  警告：请勿在空 vector 执行该操作，否则可能会 RE。下文的 `front, back` 同理

- `clear()` 清空数组，使 size 为 0。$O(1)$

- `front()` 获得首元素的传引用。

- `back()` 获得末元素的传引用。
- `begin()` 获得首元素的迭代器。

  迭代器作用类似指针，可以进行 ++ 和 -- 等来移动，可以 * 来取值。
- `end()` 获得末元素的迭代器。

  > 有 `rbegin,rend` 迭代器，感兴趣自学。
- `insert(迭代器 pos, value)`，在当前迭代器位置插入元素，当前位置及以后位置往后移，size 加一。$\mathbf{O(n)}$

  警告：不要使得迭代器越界，否则会 RE，下文 `erase` 同理。

  > 还有更多 insert 的重载方法，感兴趣自学。
- `erase(迭代器 pos)`，删掉当前迭代器位置元素，当前之后的元素往前移，size 减一。$\mathbf{O(n)}$
- `empty()` 返回布尔值，代表 vector 是否为空。$O(1)$

使用举例：

```cpp
#include <bits/stdc++.h>
using namespace std;
vector<int> v;
void print()
{ //用上文的for-each也行
    if (v.empty())
    {
        cout << "empty\n";
        return;
    }
    cout << "size=" << v.size() << ": ";
    for (auto i = v.begin(); i != v.end(); ++i)
    {
        cout << *i << ' ';
    }
    cout << '\n';
}
signed main()
{
    v.push_back(1);
    print();
    ++v.front();
    cout << v.back() << '\n';
    v.insert(v.begin(), 1);
    print();
    v.insert(v.begin() + 1, 10);
    print();
    v.erase(v.begin() + 1);
    print();
    v.pop_back();
    print();
    v.clear();
    print();
    return 0;
}
```

迭代器指向的下标，不会随着增删而移位。(通常而言，使用迭代器时，不建议同时进行增删操作；且增删后建议新开迭代器而不是用增删前的，下文的其他 STL 同理)

如：

```cpp
#include <bits/stdc++.h>
using namespace std;
vector<int> v{1, 3, 5, 7, 9};
int main()
{
    auto it = v.begin();
    v.erase(v.begin() + 1);
    ++it;
    printf("%d\n", *it);
    v.insert(v.begin() + 1, 6);
    printf("%d\n", *it);
    return 0;
}
```

嵌套

可以造一维 vector 数组，那么第一维是静态的，第二维是 vector 本身是动态的。如：

```cpp
#include <bits/stdc++.h>
using namespace std;
vector<int> v[10];
int main()
{
    v[0].push_back(1);
    cout << v[0][0] << '\n';
    v[1].resize(5);
    v[1].back()--;
    cout << v[1][4];
    return 0;
}
```

然而，这样的话无论怎么搞，只有最右维是动态的。如想每个维度都是动态的，需要使用嵌套。

实现多维动态数组，可以使用 vector 套 vector 的方法，做到每一维都是动态的。例如二维嵌套为：`vector<vector<基本数据类型>>`。

使用构造方法 `(长度, value)`，其 `value` 是一个低维 vector，即 `vector<基本类型>(长度)`。或者用 `resize`。如：

```cpp
#include <bits/stdc++.h>
using namespace std;
//二维动态数组，初始两维长度分别是0,0
vector<vector<int>> v0, v2;
//二维动态数组，初始两维长度分别是10,20;值都是0
vector<vector<int>> v1(10, vector<int>(20));
int n, m;
int main()
{
```

```
    //动态设置初始大小
    cin >> n >> m;
    v2.resize(n, vector<int>(m));
    cout << v1.size() << ' ' << v1[0].size() << '\n';
    v1[0].push_back(1);
    cout << v1[0].size() << ' ' << v1[1].size() << '\n';
    return 0;
}
```

可以发现第二维的长度是不一的，这比较容易理解。

相似的，如果想要造三维动态数组，可以用下面两种方法：

```
vector<vector<vector<int>>> v(5, vector<vector<int>>(10, vector<int>(15,1)));
//5,10,15是三个维度的初始长度,元素值全部设为1
vector<vector<vector<int>>> v2;
v2.resize(5, vector<vector<int>>(10, vector<int>(15, 1)));
```

更高维依次类推。

pair

二元对，即两个变量组成的一个数据类型。定义方法：

```
pair<首数据类型, 尾数据类型> 变量名;
pair<首数据类型, 尾数据类型> 变量名 (首, 尾);
pair<首数据类型, 尾数据类型> 变量名 = {首, 尾};
pair<首数据类型, 尾数据类型> 变量名 = make_pair(首, 尾);
```

成员变量为 `first` 和 `second`。

默认重载了比较函数，两个 pair 比较大小，以首数据类型为第一关键字，以尾数据类型为第二关键字进行升序比较。

使用举例：(当然可以不用 vector，单独使用也是可以的)

```
#include <bits/stdc++.h>
using namespace std;
vector<pair<int, double>> v = {{1, 9}, {3, 2.4}, {3, 2.2}, {2, 0}};
int main()
{
    sort(v.begin(), v.end()); //使用pair比较函数进行排序
    for (auto x : v)
    {
        printf("(%d %.1lf)\n", x.first, x.second);
    }
    return 0;
}
```

tuple

三元对。直接使用 `{}` 构造。取出方法是 `tie(变量a, 变量b, 变量c)=tuple变量`。如：

```cpp
#include <bits/stdc++.h>
using namespace std;
int main()
{
    tuple<int, int, char> t = {1, 3, 'c'};
    int a, b;
    char c;
    tie(a, b, c) = t;
    printf("%d %d %c", a, b, c);
    return 0;
}
```

> C++17 等高版本可以用 `auto[a,b,c]=t`。即：
>
> ```cpp
> tuple<int, int, char> t = {1, 3, 'c'};
> auto [a, b, c] = t;
> printf("%d %d %c", a, b, c);
> ```

stack

栈。符合后进先出(FILO,first in last out)原则，可以认为是每次只能访问、插入和删除尾部(称为栈顶)的动态数组。可以类比摞成一叠的盘子，每次只能操作顶部。

常用方法(都是 $O(1)$)：

- `size()`。同理
- `push(value)`。压栈/进栈/入栈。将一个元素放到栈顶
- `pop()`。弹栈/出栈。将栈顶元素删除

  请勿在栈空时执行该方法，否则会 RE。下文 `top` 同理
- `top()`。输出栈顶元素
- `empty()`。同理

> STL 的栈常用于常用于后续课程的滑动窗口、双指针、单调栈等算法。

queue

队列。符合先进先出(FIFO, first in first out)原则。可以类比一般的排队。只能够取队列首部，删除队列首部和将元素添加到队列尾部。是特殊的动态数组。

常用方法与栈类似，且也都是 $O(1)$：

- `size()`
- `push(value)`。入队。放到队尾。
- `pop()`。将队首出队。
- `front()`。取队首，注意不是 top。

- `back()`。取队尾。显然 `pop,front,back` 执行前都应保证非队空。
- `empty()`

> STL 的队列(及下文优先级队列)常用于后续课程的 BFS 等算法。

priority_queue

基本使用

优先级队列，即大根堆(最大堆)。是一种按照值大小进行排序的有序队列，并只保证值最大的一定在队首。

与队列常用方法不同是：

- 使用 `front` 而不是 `top`。
- 没有 `back` 方法。
- 设 $n = size()$，则 `push, pop` 都是 $O(\log_2 n)$。其他不变。

> 具体原理可以参考排序算法的堆排序，后续课程可能介绍。

如：

```cpp
#include <bits/stdc++.h>
using namespace std;
priority_queue<int> q;
int main()
{
    q.push(1);
    q.push(3);
    q.push(5);
    q.push(3);
    printf("%lld\n", q.top());
    q.pop();
    printf("%lld\n", q.top());
    q.pop();
    printf("%lld\n", q.top());
    q.pop();
    printf("%lld\n", q.top());
    return 0;
}
```

结构体

对结构体，需要重载 `<` 运算符(注意不是 >)，如：

```cpp
#include <bits/stdc++.h>
using namespace std;
struct node
{
    int v, i;
    bool operator<(const node &r) const { return v < r.v; }
```

```
};
priority_queue<node> q;
int main()
{
    q.push({10, 1});
    q.push({1, 2});
    printf("%d", q.top().i);
    return 0;
}
```

> 使用 `priority_queue<pair<int,int>> q` 可以类似实现上述内容，不需要重载，因为 node 与二元组含义类似。

逆序

如果想要实现小根堆，即每次队首都是最小值，有如下方法：

1. 每次 `push` x 时入队相反数 $-x$，每次 `top` 取出时再取相反数即 $-(-x)$
2. 重新定义模板，设为：

   ```
   priority_queue<类型, vector<类型>, greater<类型>>
   ```

   如果类型是自定义类/结构体，需要事先重载 `>` 运算符(注意不是 <)
3. 将类型设为结构体，重载 `>` 运算符，使其表现出小于的含义

如：(第二点)

```
#include <bits/stdc++.h>
using namespace std;
struct node
{
    int v, i;
    bool operator>(const node &r) const { return v > r.v; }
};
priority_queue<node, vector<node>, greater<node>> q;
int main()
{
    q.push({10, 1});
    q.push({1, 2});
    printf("%d", q.top().i);
    return 0;
}
```

与下文内容等价：(第三点)

```
#include <bits/stdc++.h>
using namespace std;
struct node
{
    int v, i;
    bool operator<(const node &r) const { return v > r.v; }
};
```

```cpp
priority_queue<node> q;
int main()
{
    q.push({10, 1});
    q.push({1, 2});
    printf("%d", q.top().i);
    return 0;
}
```

set

基本使用

集合。即满足元素互异的动态数组。set 元素是升序排序的。有下面几个较常用的构造方法：①空集；②复制自数组；③复制自 vector；④复制自 set；⑤大括号。如：

```cpp
#include <bits/stdc++.h>
using namespace std;
int a[] = {2, 2, 3, 3, 1};
set<int> s1(a, a + 5);
vector<int> v{2, 3, 2, 5, 4};
set<int> s2(v.begin(), v.end());
set<int> s3(s2);
set<int> s4 = {1, 9, 1, 9, 8, 1, 0};
int main()
{
    s3.erase(2);
    printf("%d %d %d\n", s1.size(), s2.size(), s3.size());
    return 0;
}
```

常用方法：

- `size()`，$O(1)$
- `clear()`，$O(1)$
- `insert(value)`，尝试插入一个值，若已存在则忽略，$O(\log_2 n)$
- `erase(value)`，尝试删除一个值，若不存在则忽略(不会报错)，$O(\log_2 n)$
- `begin()` 和 `end()` 取首尾迭代器，$O(1)$
- `find(value)`，查找并返回对应值的迭代器，若不存在则返回 `end` 迭代器，$O(\log_2 n)$
- `lower_bound(value)`，查找并返回满足大于等于对应值的最小元素迭代器，不存在返回 `end`，$O(\log_2 n)$
- `upper_bound(value)`，查找并返回满足大于对应值的最小元素迭代器，不存在返回 `end`，$O(\log_2 n)$

如：

```cpp
#include <bits/stdc++.h>
using namespace std;
set<int> s = {1, 9, 1, 9, 8, 1, 0};
int main()
{
    auto it = s.find(5);
```

```
    printf("%d\n", it == s.end()); //判断是否在s里,与end比较
    auto it2 = s.find(1);
    printf("%d\n", *it2);
    s.insert(4);
    ++it2;
    printf("%d\n", *it2);
    s.erase(it2);
    auto it3= s.lower_bound(4);
    printf("%d\n", *it3);
    auto it4 = s.upper_bound(8);
    printf("%d\n", *it4);
    for (auto v : s)
    {
        printf("%d ", v);
    }
    return 0;
}
```

> 有 STL 标准函数并集 set_union、交集 set_intersection 和差集 set_difference，感兴趣自学。
>
> set 的原理是红黑树(一种平衡树)，所以复杂度是 $O(\log_2 n)$；下文 map 同理。
>
> 注意：使用 STL 函数 `lower_bound, upper_bound` 作用于 set 和 map 是 $O(n)$，必须使用成员方法才能保证 $O(\log_2 n)$

结构体与逆序

使用于 set 的结构体，必须重载 `<` 运算符。如：

```
#include <bits/stdc++.h>
using namespace std;
struct node
{
    int i;
    pair<int, int> v;
    bool operator<(const node &r) const { return i < r.i; }
};
set<node> s;
int main()
{
    s.insert({1, {2, 3}});
    s.insert({0, {4, 3}});
    auto pr = s.begin()->v;
    printf("%d %d", pr.first, pr.second);
    return 0;
}
```

逆序的话，一种方法是使用 `set<类型, greater<类型>>`(其他方法与优先级队列类似)，如：

```cpp
#include <bits/stdc++.h>
using namespace std;
using pr = pair<int, int>; //等效于#define pr pair<int,int>是一种偷懒简写
set<pr, greater<pr>> s = {{0, 0}, {0, 1}, {1, 0}, {1, 1}};
int main()
{
    for (auto v : s)
    {
        printf("%d %d\n", v.first, v.second);
    }
    return 0;
}
```

unordered_set

> 习惯上可以简称为 `unset`(语法上不能)。下文 `unmap` 同理。并不是很常用，但有时候可以使用。

是无序的集合。无序的优点是性能更高(即上文所有 $O(\log_2 n)$ 在这里都是 $O(1)$)(但常数大，故只有在较大数据下如 10^5 或以上，才能显现出来，小数据不如 set)，实现原理是哈希函数(`unmap` 同理)。

> 哈希函数会在后续课程讲解，这里不介绍。

对结构体，必须重载 `==` 运算符的数据类型才能使用，且必须重载 `()` 运算符代表哈希函数，传入参数是该结构体(重载 `()` 的可以是其他结构体，也可以是自身)。此时格式为 `unordered_set<类型，重载了()的类型>`，如：

```cpp
#include <bits/stdc++.h>
using namespace std;
unordered_set<int> s = {1, 1, 4, 5, 1, 4};
struct node
{
    int x, y, z;
    bool operator==(const node &r) const
    {
        return x == r.x && y == r.y && z == r.z;
    }
    size_t operator()(const node &r) const
    {
        return r.x * 1e9 + r.y * 1e5 + r.z;
    }
};
unordered_set<node, node> s2;
int main()
{
    printf("%d\n", s.size());
    s2.insert({1, 2, 3});
    s2.insert({1, 2, 3});
    s2.insert({1, 2, 4});
    printf("%d\n", s2.size());
    return 0;
}
```

> 此外，还有允许重复的多重集 `multiset` 和同理的 `multimap`，以及多重无序集，很少用，感兴趣自学。
>
> 注意 pair 类型是无法存 unordered_set 的，所以需要给它重载一下或手写 pair。

map

映射，字典。有键和值两个元素。可以理解键是广义数组下标(即从整数拓展到一切数据类型)，值是元素值。所以可以使用 `[]` 运算符。有序，按照键升序排序。

定义：

```
map<键数据类型, 值数据类型> 变量名;
```

常用方法：

- `size()`，$O(1)$
- `clear()`，$O(1)$
- `insert(pair)`，插入键值对，若键已存在则忽略(而不是覆盖)，否则插入；$O(\log_2 n)$
- `erase(key)`，删除，若找得键就删，否则忽略；$O(\log_2 n)$
- `begin()` 和 `end()` 得到迭代器，指向的是 pair；$O(1)$
- `find(key)`，查找这个键对应的 pair，不存在返回 `end` 迭代器；$O(\log_2 n)$
- `lower_bound(key)`，大于等于该键的迭代器；$O(\log_2 n)$
- `upper_bound(key)`，大于该键的迭代器；$O(\log_2 n)$
- `[key]`，取该键对应的值(或赋值)，若不存在马上新建并设值为 0；$O(\log_2 n)$

`for-each` 遍历时得到的是 pair。如：

```
#include <bits/stdc++.h>
using namespace std;
map<double, int> m = {{1.1, 100}, {2.2, 10}, {3.3, 1}};
int main()
{
    m.insert({4.4, 0});
    m.insert({1.1, 666});
    for (auto v : m)
    {
        printf("%.2lf %d\n", v.first, v.second);
    }
    m.erase(114514);
    printf("%d\n", m.size());
    m.erase(1.1);
    printf("%d\n", m.find(1.1) != m.end());
    auto it = m.begin();
    printf("%.2lf %d\n", it->first, it->second);
    printf("%d\n", m.lower_bound(2.2)->second);
    printf("%d\n", m.upper_bound(2.2)->second);
    printf("%d\n", m[2.2]);
    printf("%d\n", m[2.3]);
    m[2.4] = 666;
    printf("%d\n", m[2.4]);
    return 0;
```

```
}
```

逆序同理，改一下加 `greater<键类型>` 即可，如：

```cpp
#include <bits/stdc++.h>
using namespace std;
map<int, vector<int>, greater<int>> m;
int main()
{
    m[1437] = {1, 1, 4, 5, 1, 4};
    m[580] = {};
    for (auto pr : m)
    {
        printf("%d:", pr.first);
        for (auto v : pr.second)
        {
            printf(" %d", v);
        }
        printf("\n");
    }
    return 0;
}
```

结构体同理。只需要键重载了 `<` 运算符即可。如：(事实上很少键会用结构体)

```cpp
#include <bits/stdc++.h>
using namespace std;
struct node1
{
    int i, v;
    bool operator<(const node1 &r) const { return v < r.v; }
};
struct node2
{
    int x, y, z, w;
};
map<node1, node2> m = {{{1, 2}, {1, 2, 3, 4}}, {{2, 1}, {5, 6, 7, 8}}};
int main()
{
    for (auto i : m)
    {
        auto k = i.first;
        auto v = i.second;
        printf("%d %d -> %d %d %d %d\n", k.i, k.v, v.x, v.y, v.z, v.w);
    }
    return 0;
}
```

`unmap` 可以类推得之。可自行尝试，相信不难。

bitset

可以认为是静态布尔值数组，但是比一般的布尔值数组快，一般认为快 32 倍，因为原理大约是用一个 int 的 32 位当 32 个 bool 来用。`bitset` 在后续学到动态规划等地方可能常用，用作优化。如果是 64 位计算机，就快 64 倍。

定义：

```
bitset<长度> 变量名; //一开始全false
bitset<长度> 变量名(整型变量); //将整型转二进制存入(从低到高)
bitset<长度> 变量名(只含01的字符串);
```

如：

```
#include <bits/stdc++.h>
using namespace std;
bitset<10> b1(5), b2("1101");
int main()
{
    printf("%d %d %d %d\n", b1[3] & 1, b1[2] & 1, b1[1] & 1, b1[0] & 1);
    printf("%d %d %d %d\n", b2[3] & 1, b2[2] & 1, b2[1] & 1, b2[0] & 1);
    return 0;
}
```

可以直接当布尔值数组来用，也有常用方法：一般是 $O(\frac{n}{w}), w$ 是计算机位数

- 使用位运算符与一个 `bitset` 直接进行位运算
- `count()` 求 1 的个数
- `any()` 是否存在 1
- `none()` 是否不存在 1
- `set()` 全部设为 1
- `reset()` 全部设为 0
- `flip()` 全部按位取反

如：

```
#include <bits/stdc++.h>
using namespace std;
bitset<10000> b1("1110"), b2("1011");
int main()
{
    printf("%d\n", (b1 & b2).count());
    printf("%d\n", (b1 | b2).flip().count());
    printf("%d\n", (b1 ^ b2)[1] & 1);
    return 0;
}
```

> 补充知识：基本位运算(自学，重要)： 与、或、异或、左移与右移
>
> 注意要点：
>
> - 优先级： `~; +, -; <<, >>; ==, !=; &; ^; |; &&; ||; ?:`
> - 移位结果为 long long 时应该是 `1LL << k`

- 右移位，等同于round(x/2.0)，负数的移位结果不会大于-1

常见应用：

- 取正数 `x` 的从左往右(从零数)第 `i` 位：`(x>>i)&1`
- 对某个正数 `x` 从左往右(从零数)第 `k` 位修改取反：`x^=(1<<k)`
- `c&15` 或 `c^'0'` 优化 数字字符转数值(手写快读常用)
- 取某个数的最低 1 所在位(lowbit 操作，后续学树状数组用)：`x&-x`

内建函数：

- 注：对 `unsigned long long` 每个函数名后面加上 `ll` (传入的是什么类型不影响结果, 影响的是函数名)

1.__builtin_popcount(unsigned int n)

该函数时判断n的二进制中有多少个1

```
int n = 15; //二进制为1111
cout<<__builtin_popcount(n)<<endl;//输出4
```

2.__builtin_parity(unsigned int n)
该函数是判断n的二进制中1的个数的奇偶性

```
int n = 15;//二进制为1111
int m = 7;//111
cout<<__builtin_parity(n)<<endl;//偶数个，输出0
cout<<__builtin_parity(m)<<endl;//奇数个，输出1
```

3.__builtin_ffs(unsigned int n)
该函数判断n的二进制末尾最后一个1的位置，从一开始

```
int n = 1;//1
int m = 8;//1000
cout<<__builtin_ffs(n)<<endl;//输出1
cout<<__builtin_ffs(m)<<endl;//输出4
```

4.__builtin_ctz(unsigned int n)
该函数判断n的二进制末尾后面0的个数，当n为0时，和n的类型有关

```
int n = 1;//1
int m = 8;//1000
cout<<__builtin_ctzll(n)<<endl;//输出0
cout<<__builtin_ctz(m)<<endl;//输出3
```

5. __builtin_clz (unsigned int x)
返回前导的0的个数。

```
int n = 1;  //1
int m = 8;  //1000
cout<< 32 - __builtin_clz(n) <<endl;    //输出1
cout<< 64 - __builtin_clzll(m) <<endl;  //输出4
```

> 应用：`31 - __builtin_clz(n)` 等效于 $\lfloor \log_2 n \rfloor$
>
> 其他：异或的性质
>
> - 交换律、结合律、消去律，有单位元 0，自己与自己运算得单位元
> - $a \oplus b \leq a + b = (a|b) + (a\&b)$
>
> 前半句：因异或是不进位的加法；后半句：因 $a\&b$ 是进位部分

string

可变长字符串。默认使用 char 模板，不需要额外指定。只推荐使用一般赋值来初始化。常用方法：

- `+`。字符串拼接。$O(n+m)$

  区别：`+=`，也是拼接，但 $O(m)$

- `<`，`>`，`<=`，`>=`，`!=`，`==`。字符串字典序比较。$O(\min(n,m))$
- `=`。字符串复制，$O(m)$
- `[]`。取单个下标的字符的传引用，可以修改。$O(1)$
- `substr(int start[,int len])` 取下标范围 $[start, start+len)$ 的子串

  如果 $start+len$ 越界，取到结尾为止。若 start 越界报错(`erase` 同理)。不改变原字符串。$O(len)$
- `back()` 取最后一个字符的传引用。$O(1)$
- `insert(int pos, string s)` 在下标 `pos` 处插入字符串 s，原下标 `pos` 和之后的往后推。改变原字符串。越界会报错。$O(n)$
- `erase(int pos, int len)`，删除下标范围 $[start, start+len)$ 的子串，原 $start+len$ 和以后的子串往前挪。$O(n)$
- `find(char/string s, start=0)`，在以 $start$ 下标开头的后缀找是否出现过子串 s，是返回出现的首字符下标，不出现返回 -1。$O(nm)$

  > 还有 `find_first_of`，`find_last_of`，`find_first_not_of`，`find_last_not_of`，顾名思义，感兴趣自学

- `c_str()` 取 C 风格字符串。$O(n)$

  反过来，C 风格字符串转成 C++ 的 string 直接用 `=` 即可。

- `begin()`，`end()` 取首尾迭代器，$O(1)$

> 虽然有 `push_back` 函数，但强烈不建议使用，该函数可能引发乱码

字符串只能用 `cin` 输入与 `cout` 输出。如：

```cpp
#include <bits/stdc++.h>
using namespace std;
string a = "baicha", b = "guodong", c;
int main()
{
    c = a + "_xingyue";
    cout << a << ' ' << b << ' ' << c << '\n';
    cout << (a == c) << ' ' << (a < b) << ' ' << (a >= b) << '\n';
```

```
    cout << a.substr(3) << ' ' << a.substr(3, 100) << b.substr(0, 3) << '\n';
    b.insert(3, "zi");
    cout << b << '\n';
    b.erase(3, 2);
    cout << b << '\n';
    b.back() = 'G';
    cout << b << '\n';
    cout << a.find("cha") << ' ' << a.find('a') << ' ' << a.find('a', 2) << '\n';
    cout << (int)b.find('O') << '\n'; //区分大小写
    printf("%s", c.c_str());
    return 0;
}
```

```
string s = "abc";
s[1] = 'B';
cout << s << s[0];
```

常用字符串函数：(都是 $O(n)$)

- `getline(cin, string)`，读整行(与 `cin.getline` 读 C 风格字符串区分)
- `to_string(any)` 将其他类型转化为 `string`。 注：char 会视为 int。
- `stoi(), stol(), stof(), stod()` 将字符串转化为别的类型(分别是 `int, long long, float, double`)
- `count(首迭代器, 尾迭代器, char)` 统计字符出现次数
- `replace(首迭代器, 尾迭代器, char source, char dest)` 全部字符替换

如：

```
#include <bits/stdc++.h>
using namespace std;
string s, x, y;
int main()
{
    getline(cin, s);
    cout << count(s.begin(), s.end(), 'a') << '\n';
    x = to_string(92 + 8);
    printf("%d\n", stoi(x + "86"));
    y = "aaAA";
    replace(y.begin(), y.end(), 'a', 'c');
    cout << y;
    return 0;
}
```

> string 可以进行一系列正则表达式操作，比较少用，感兴趣自学

stringstream

流。一般常用于读取一行(一个字符串)里的内容，即把一个字符串当成要 `cin` 的对象来处理。

新建流：

```
stringstream 流变量名(字符串变量);
```

从流里读出一个变量(类比 `cin >> 变量名`)：

```
流变量名 >> 变量名;
```

如：

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    string s;
    getline(cin, s);
    stringstream ss(s);
    int v, t = 0;
    while (ss >> v)
    {
        t += v;
    }
    cout << t;
    return 0;
}
```

常用函数

分为编译器函数和库函数。

编译器函数，顾名思义，这些函数不在任何库里，而是编译器自带的。又称内置函数。特征是通常以下划线开头。

> 上文提到的位运算函数都是编译器函数。

此外，还有：常用的是 `__gcd(a, b)`，显然就是求两个整数的最大公因数。如：

```
#include <bits/stdc++.h>
using namespace std;
signed main()
{
    printf("%d\n", __gcd((int)(1e9 + 7), 998244353)); //两个常见质数
    printf("%d\n", __gcd(30, 20));
    printf("%d\n", __gcd(0, 5)); //(0,x)=x
    return 0;
}
```

不建议对一正一负使用 `__gcd`，结果正负号无规律。若同为负，则有 $\gcd(x,y)=-\gcd(-x,-y)$。$O(\log\max(a,b))$

常用库函数有：

- `max(v1, v2)` 返回较大值；多个值就 `max({v1, v2, ..., vn})`；同理有 `min` 函数。$O(|v|)$ (即参数数目)
- `sort(首迭代器, 尾迭代器[,比较函数])` 排序，$O(n\log n)$，原理是内省排序(快排+堆排+插排)
- `unique(首迭代器, 尾迭代器)`。对升序序列去重，返回去重后不重部分长度，$O(n)$
- `reverse(首, 尾)`。转置一个序列，$O(n)$
- `inplace_merge(l, c, r[, 比较函数])`，合并一个序列的两个连续升序部分 $[l,c),[c,r)$ 为一个升序序列 $[l,r)$

  `merge(首1, 尾1, 首2, 尾2, 目标迭代器[, 比较函数])`，合并两个升序序列到目标迭代器上，均 $O(n)$

  > 在后续课程的归并排序中可以使用到

- `lower_bound(首, 尾, 值)`，找升序序列首个大于等于值的迭代器所在，查无返回尾迭代器，$O(\log_2 n)$

  `upper_bound(首, 尾, 值)`，找升序序列首个大于值的迭代器所在，查无返回尾迭代器，$O(\log_2 n)$

  `lower_bound(首, 尾, 值, greater<类型>())`，找降序序列首个小于等于值的迭代器所在，查无返回尾迭代器，$O(\log_2 n)$

  `upper_bound(首, 尾, 值, greater<类型>())`，找降序序列首个小于值的迭代器所在，查无返回尾迭代器，$O(\log_2 n)$

  > 在后续课程的二分算法广泛使用；请注意用这些函数直接操作 set/map 的复杂度是 $O(n)$ 的，而使用 set/map 的同名方法才是 $O(\log_2 n)$ 的

如：

```cpp
#include <bits/stdc++.h>
using namespace std;
int a[] = {1, 4, 3, 7, 5, 8, 1};
void print(int *a, int n)
{
    for (int i = 0; i < n; ++i)
    {
        printf("%d ", a[i]);
    }
    putchar('\n');
}
int b[] = {1, 3, 5, 70, 2, 4, 6, 8, 10, 12};
int c1[] = {1, 3, 5}, c2[] = {2, 4}, c3[10];
int main()
{
    reverse(a, a + 7);
    print(a, 7);
    sort(a, a + 7);
    print(a, 7);
```

```
    int n2 = unique(a, a + 7) - a;
    print(a, n2);
    auto pos = lower_bound(a, a + n2, 2);
    printf("%d\n", *pos);
    inplace_merge(b, b + 4, b + 10);
    print(b, 10);
    merge(c1, c1 + 3, c2, c2 + 2, c3);
    print(c3, 5);
    return 0;
}
```

> 还有一些不常用的，如 `nth_element, advance, max_element`。感兴趣自行学习。

> 生成随机数，可以自行去了解 `mt19937` 数据类型等，如：

```
#include <bits/stdc++.h>
using namespace std;
#define sc(x) scanf("%lld", &x)
typedef long long ll;
random_device divc;
mt19937 mt(divc());
uniform_int_distribution<ll> range1(1, 10000000);
ll a, b;
signed main() // SCNUOJ1001
{
 sc(a), sc(b);
 while (true)
 {
     ll c = range1(mt);
     if (a + b == c)
     {
         printf("%lld", c);
         return 0;
     }
 }
 return 0;
}
```

```
#include <bits/stdc++.h>
using namespace std;
#define sc(x) scanf("%lld", &x)
typedef long long ll;
ll v[5050], n = 10, a, b;
signed main()
{
 random_device rd;
 mt19937 mt(rd());
 for (ll i = 1; i <= n; ++i)
 {
     v[i] = i;
 }
 shuffle(v + 1, v + 1 + n, mt);
 for (ll i = 1; i <= n; ++i)
 {
```

```
17      printf("%lld ", v[i]);
18   }
19   return 0;
20 }
```