C++ Advanced Features

xzy 2024.7.16

序

这玩意我觉得没什么好讲的，所以大家听着玩。

运算符重载 (since C++ 98)

假如有一个结构体：

```
struct Vector2D {
    double x, y;
};
```

我们需要计算两个向量的数量积，可以定义一个函数：

```
double dot(Vector2D a, Vector2D b) {
    return a.x * b.x + a.y * b.y;
}
```

但是这样子计算数量积很麻烦，需要用 `dot(a, b)` 而不是自然的 `a * b`，有什么方法吗？

运算符重载 (since C++ 98)

为了让我们自定义的类型也支持 C++ 的标准运算符，我们可以使用运算符重载，例如上面的例子，我们可以改写成这样：

```
double operator*(Vector2D a, Vector2D b){
    return a.x * b.x + a.y * b.y;
}
```

其中，`operator` 是一个关键字，表示我们正在定义一个运算符。`*` 就是我们想要定义的运算符。运算符重载的语法和函数语法基本一致，但无需写出函数名。

运算符重载 (since C++ 98)

我们同样可以重载一个元素的位移运算符，来支持 `cin/cout/cerr` 等输入输出流。例如：

```
istream& operator>>(istream& in, Vector2D &vct){
    in >> vct.x >> vct.y;
    return in;
}

ostream& operator<<(ostream& out, const Vector2D &vct){
    out << vct.x << ' ' << vct.y;
    return out;
}
```

函数的重载 (since C++ 98)

对于函数的重载，我们来看一例子，假如我们不但希望计算两个二维向量的数量积，而且希望计算三维向量的数量积，按照一般的写法，可以写成这样：

```
struct Vector2D { double x, y; };
struct Vector3D { double x, y, z; };
double dot2d(Vector2D a, Vector2D b){ return a.x * b.x + a.y * b.y; }
double dot3d(Vector3D a, Vector3D b){ return a.x * b.x + a.y * b.y + a.z * b.z; }
```

但是这样写太麻烦了，可不可以统一使用 `dot` 函数呢？我们可以使用函数的重载来解决这个问题。

函数的重载 (since C++ 98)

C++ 允许我们定义多个同名函数，但是这些函数的参数类型不同，这被称为函数的重载。

对于上面的例子，可以改写成这样：

```
...省略结构体定义...
double dot(Vector2D a, Vector2D b){ return a.x * b.x + a.y * b.y; }
double dot(Vector3D a, Vector3D b){ return a.x * b.x + a.y * b.y + a.z * b.z; }
```

注意，函数的重载的参数类型必须不同，否则编译器就不知道应该调用哪个函数，这个现象叫做不明确（ambiguous）。

PS：Testlib.h 禁止用户使用 `rand` 函数，就是用了函数重载的方法，让调用 `rand` 函数成为一种不明确的行为。

函数的声明 (since C++ 98)

假如我们有一个函数：

$$f(x) = \begin{cases} 0 & x = 1 \\ f(\frac{x}{2}) & 2 \mid x \\ f(3x+1) & \text{otherwise} \end{cases}$$

一种实现如下：

```
int f_odd(int x) { return x == 1 ? 1 : f_even(3 * x + 1); }
int f_even(int x) { return x == 1 ? 1 : ((x / 2) & 1 ? f_odd(x / 2) : f_even(x / 2)); }
```

（我知道这样实现非常奇怪，但是为了演示，就这样写了）

函数的声明 (since C++ 98)

上面的实现中，函数 `f_odd` 中调用了 `f_even`，但是 `f_even` 还没有定义，所以编译器会报错。而 `f_even` 又因为相同的原因，不能提到 `f_odd` 的定义之前。为了解决这个问题，C++ 引入了函数的声明。我们可以这样写：

```
int f_even(int x);
```

没有写函数体的函数定义，我们称之为函数声明。函数声明告诉编译器，这个函数存在，但是没有定义，所以编译器可以继续编译。如果该文件中没有函数定义，编译器就会在静态库中寻找，如果找不到，就会报错。

上面的例子可以改成这样：

```
int f_even(int x);
int f_odd(int x) { return x == 1 ? 1 : f_even(3 * x + 1); }
int f_even(int x) { return x == 1 ? 1 : ((x / 2) & 1 ? f_odd(x / 2) : f_even(x / 2)); }
```

函数的声明 (since C++ 98)

类似地，C++ 还引入了不完全类型。用于解决类型定义的冲突。

限于篇幅，我们在这里不会讨论。感兴趣的同学请自行搜索。

bitset (since C++ 98)

如果你没有使用万能头文件，使用 bitset 时，需要包含 `<bitset>` 头文件。

bitset 是一个存储定长 01 序列的容器。它相比于一般的 `bool[]` 有更高的效率。

定义 bitset 非常容易：

```
bitset<10086> my_bitset;
```

上面的代码定义了一个长度为 10086 的 bitset。bitset 与 vector 等容器不同，bitset 的大小是固定的，不能改变。

bitset (since C++ 98)

下面我们将介绍 bitset 的成员函数：

- `bitset::size()`：返回 bitset 的长度。

```
my_bitset.size() // 10086
```

- `bitset::operator[](size_t i)`：获取 bitset 的第 i 位的值。

```
my_bitset[0] // 初始时为 0
my_bitset[0] = 1 // 设置第 0 位为 1
my_bitset[0] // 1
```

- `bitset::any()` 判断 bitset 中是否存在 1。

```
my_bitset.any() // true
```

bitset (since C++ 98)

- `bitset::all()` 判断 bitset 中是否不存在 0。

```
my_bitset.all() // false
```

- `bitset::flip()` 反转 bitset 的所有位。

```
my_bitset.flip();
my_bitset[0] // 0
my_bitset[1] // 1
```

- `bitset::reset()` 将 bitset 的所有位设置为 0。

```
my_bitset.reset();
my_bitset[1] // 0
```

此外，bitset 还支持一些运算符。如按位与/或/异或/取反/移位等。

bitset (since C++ 98)

bitset 为什么那么快呢?

因为 bitset 会按照计算机的位宽（64）进行优化。具体来说，它相当于一个 `unsigned long long` 数组，每个元素存储若干位。这样子批量操作的常数就会大大降低，空间占用也会大大减少。

使用 `bitset` 时，尽量保证访问是连续的，这样可以提高缓存命中率。如果不保证这一点，可能会导致负优化。

PS：`vector<bool>` 实际上类似一个可变长的 bitset，但是 bug 比较多，不推荐使用。

引用 (since C++ 98)

引用可以代替一部分指针。

比如说在 C 语言中，我们要写一个交换两个整数的时候，需要这么写：

```
void swap(int *a, int *b){
    int tmp = *a;
    *a = *b; *b = tmp;
}
```

使用指针常常会很麻烦，且会引入类似 wild pointer 的问题。为此，C++ 引入了引用。

引用 (since C++ 98)

改写后的代码如下：

```
void swap(int &a, int &b){
    int tmp = a;
    a = b; b = tmp;
}
```

在定义引用变量或引用参数时，需要加上 `&` 。比如：

```
void swap(int &a, int &b);
int &c = x;
```

引用类型必须和所赋的变量类型一致。

引用 (since C++ 98)

引用也可以作用到变量上，例如：

```
int x = 1;
int &y = x;
cout << x << ' ' << y << endl;// 1 1
x = 2;
cout << x << ' ' << y << endl;// 2 2
```

引用同时还涉及到值类别的概念，由于我觉得这玩意 useless，故不展开。

auto (since C++ 11)

auto 是 C++11 新增的关键字，用于自动推导变量的类型。

例如，在写颜色段均摊的时候，经常会遇到这样的场景：

```
for(multiset<int>::iterator it = s.begin(); it != s.end(); it++){
    // do something
}
```

`multiset<int>::iterator` 不但拼写耗时，还容易出错。我们可以使用 `auto` 来解决这个问题。

```
for(auto it = s.begin(); it != s.end(); it++){
    // do something
}
```

auto (since C++ 11)

auto 可以推到出一个值或变量的类型。但不能推断函数参数的类型（需要用到后面提到的 template）。

auto 也可以用于返回值的推导，不过需要保证返回值的类型是唯一的。（since C++ 14）

Range-based for loop (since C++ 11)

C++11 新增了 range-based for loop，可以替代一些循环。

在 C++ 98 时，假如我们需要遍历一个 `vector<int>` 的所有元素，会这样写：

```
vector<int> vct;
// 写法 1
for(unsigned i = 0; i < vct.size(); i++){
    int now = vct[i];
}
// 写法 2
for(vector<int>::iterator it = vct.begin(); it != vct.end(); it++){
    int now = *it;
}
```

这都太麻烦了，有什么方法可以简化呢?

Range-based for loop (since C++ 11)

在 C++ 11 中，可以这样写：

```
for(int now : vct){ }
```

无需定义任何变量，直接使用 `now`。

循环变量也可以是引用，这样子可以修改对应位置的值，如：

```
for(int &now : vct) { now *= 2; }
```

等价于

```
for(int i = 0; i < vct.size(); i++){ vct[i] *= 2; }
```

Lambda (since C++ 11)

Lambda 是 C++11 新增的语法，用于定义匿名函数。

假如需要将一个长度为 n 的数组按照绝对值从大到小排序，在 JavaScript 中，我们可以这样写：

```
arr.sort((a, b) => Math.abs(b) - Math.abs(a));
```

而 C++ 98 中，却只能这样写：

```
bool cmp(int a, int b) { return abs(b) < abs(a); }
sort(arr.begin(), arr.end(), cmp);
```

在 C++ 11 中，也支持了定义函数的表达式，也就是所谓的 Lambda 表达式。

Lambda (since C++ 11)

Lambda 表达式可以这样写：

```
[](int a, int b) { return abs(b) < abs(a); }
```

其中，`[]` 表示 capture list。如果为空，表示不能访问任何局部变量。`[&]` 表示 capture list 中可以访问所有局部变量，并且以引用形式捕获（即，可以改变变量的值）。`[=]` 表示 capture list 中可以访问所有局部变量，并且以值形式捕获（即，无法直接修改变量的值）。

其余部分和普通函数定义大体相同。另外，Lambda 表达式的参数是可以用 `auto` 推导的。

Lambda 表达式是一个表达式，返回一个函数，因此也可以保存到一个变量中，变量类型可以用 `auto` 推导，或者使用 `function` 模板类 `function<bool(int,int)>`，在此不详细介绍。

Lambda (since C++ 11)

利用上述知识，可以将一开始的示例代码改写成：

```
sort(arr.begin(), arr.end(), [](int a, int b) { return abs(b) < abs(a); });
```

是不是丝滑多了……

constexpr (since C++ 11, C++ 14 增强)

constexpr 是 C++11 新增的关键字，用于修饰函数、变量、类型等，使得它们可以在编译期进行求值。

在 C++ 中，下面的代码会真的执行 1000 次循环来计算 $1+2+3+...+1000 \pmod{998244353}$：

```
int calc(int x) {
    int sum = 0;
    for (int i = 1; i <= x; i++) {
        sum = (sum + i) % 998244353;
    }
    return sum;
}

int val = calc(1000);
```

constexpr (since C++ 11, C++ 14 增强)

现在我们加上 constexpr 试一试：

```
constexpr int calc(int x) {
    int sum = 0;
    for (int i = 1; i <= x; i++) {
        sum = (sum + i) % 998244353;
    }
    return sum;
}

int val = calc(1000);
```

constexpr (since C++ 11, C++ 14 增强)

把这两份代码放到 Compiler Explorer 上去测试。第一份代码，第二份代码

发现第二份代码编译后汇编语言就变成了：

```
val:
        .long   500500
```

相当于：

```
int val = 500500
```

也就是说，编译器在编译期就完成了 `calc` 函数的计算。这得益于 constexpr 关键字。

constexpr (since C++ 11, C++ 14 增强)

constexpr 可以修饰变量：

```
constexpr double CaiXuKun = 5.0 / 2;
```

等价于

```
constexpr double CaiXuKun = 2.5;
```

constexpr 也可以用于修饰函数，如上面的例子。用于表示这个函数的值编译期就可以进行计算。

constexpr 修饰的变量与 const 类似，均不能修改。

许多标准库函数都无法再编译期进行计算。具体请自行尝试。

constexpr (since C++ 11, C++ 14 增强)

constexpr 与 const 不同，主要在于：

- const 只限定了变量只读（readonly），而 constexpr 需要限定变量是可以编译器计算的常量（constant）。
- const 修饰类型（例如 `int` 与 `const int` 是两个类型），而 constexpr 修饰变量（例如 `int` 与 `constexpr int` 都是 `int`）。

const & mutable (since C++ 98)

请看下面的例子：

```
#include <bits/stdc++.h>
using namespace std;
struct Name {
    string str;
    Name(string str) : str(str) {}
    void greet() { cout << "Hello " << str << '\n'; }
};
const Name name("ytxy");
signed main(){ name.greet(); }
```

const & mutable (since C++ 98)

上面的代码会报错：

```
<source>: In function 'int main()':
<source>:9:26: error: passing 'const Name' as 'this' argument discards qualifiers [-fpermissive]
    9 | signed main(){ name.greet(); }
      |                ~~~~~~~~~~^~
<source>:6:10: note:   in call to 'void Name::greet()'
    6 |     void greet() { cout << "Hello " << str << '\n'; }
      |          ^~~~~
Compiler returned: 1
```

原因是编译器比较蠢，不知道 `greet` 函数不会修改 `name` 的值，不会违反 `const` 的限制。

为了解决这个问题，C++ 引入了 const 关键字修饰函数的语法。

const & mutable (since C++ 98)

我们把代码改成这样：

```
#include <bits/stdc++.h>
using namespace std;
struct Name {
    string str;
    Name(string str) : str(str) {}
    void greet() const { cout << "Hello " << str << '\n'; }
};
const Name name("ytxy");
signed main(){ name.greet(); }
```

相当于通知编译器，`Name::greet` 函数不会改变对象的值，不会违反 `const` 的性质。

被 `const` 修饰的函数无法修改成员函数，也无法调用未被 `const` 修饰的函数，除了某些例外。

const & mutable (since C++ 98)

在实现颜色段均摊的时候，`set` 迭代器储存的元素被定义为 `const` 变量，如何修改值呢？为了解决这个问题，C++ 配套引入了 mutable 关键字，用于修饰变量。

被 mutable 修饰的变量，始终处于可变状态，不受 `const` 约束。

const & mutable (since C++ 98)

例如：

```
#include <bits/stdc++.h>
using namespace std;
struct Name {
    string str;
    int cnt;
    Name(string str) : str(str) { cnt = 0; }
    void greet() const { cout << "Hello " << str << '\n'; cnt++; }
};
const Name name("ytxy");
signed main(){ name.greet(); }
```

会报错，原因是 `cnt++` 违反了 const 的规则，这个时候我们可以使用 `mutable`。

const & mutable (since C++ 98)

将代码修改为：

```
#include <bits/stdc++.h>
using namespace std;
struct Name {
    string str;
    mutable int cnt;
    Name(string str) : str(str) { cnt = 0; }
    void greet() const { cout << "Hello " << str << '\n'; cnt++; }
};
const Name name("ytxy");
signed main(){ name.greet(); }
```

编译就通过了。

namespace (since C++ 98)

假如一道题既需要写线段树，又需要写树状数组，那么代码可能长这样：

```
// ...省略若干代码...
void update(int ql, int qr, int v, int i, int l, int r);
// ...省略若干代码...
void update(int p, int v);
// ...省略若干代码...
```

那么这样 `update` 就重名了，虽然根据函数重载的特性，不会产生编译错误，但是难免会导致代码混乱。为了解决重名问题，C++ 引入了 namespace。

namespace (since C++ 98)

使用 namespace 可以创建一个命名空间，命名空间里面可以塞一些定义。在命名空间内的代码，优先访问本命名空间内的东西。如：

```
string name = "Schools";
namespace YZYL {
    string name = "YZYL";
    void promotion(){
        cout << "Hello everybody, We're " << name << endl;
    }
}
namespace HNFMS {
    string name = "HNFMS";
    void promotion(){
        cout << "Hello everybody, We're " << name << "one of the" << ::name << endl;
        cout << "We're not " << YZYL::name << "!" << endl;
    }
}
```

namespace (since C++ 98)

我们使用 `命名空间名::成员` 访问命名空间的成员。使用 `::成员` 访问全局变量或函数。

命名空间允许嵌套。同时也支持取别名，如：

```
namespace A {
    namespace B {
        int value = 1;
    }
}
namespace C = A::B;
int val = C::value;
int val2 = A::B::value;
val == val2; // true
```

这里的 `C` 就是 `A::B` 的别名。

另外可以使用 `using namespace A` 来将命名空间 A 的所有成员引入到全局命名空间。

仿函数 (since C++ 98，C++ 11 增强)

在 `priority_queue` 中，如果需制定堆的类型为小根堆，那么可以这样写：

```
priority_queue<int, vector<int>, greater<int>> pq;
```

你是否想过，`greater<int>` 到底是什么？为什么我自己写的函数就不能传入模板参数呢？

另外，为什么从大到小排序的代码，`greater<int>` 后需要加上一对括号呢？

```
sort(arr.begin(), arr.end(), greater<int>());
```

仿函数 (since C++ 98，C++ 11 增强)

仿函数其实是一个对象。不过它重载了函数调用运算符 `()`，所以它可以像函数一样被调用。可以用下面的方法定义一个仿函数结构体：

```
struct MyGreater {
    bool operator()(int a, int b) { return a > b; }
};
```

`MyGreater` 就类似于 `greater<int>`。然后使用 sort 的时候就可以这样用了：

```
sort(arr.begin(), arr.end(), MyGreater());
```

`MyGreater()` 表示创建一个对象，`MyGreater` 对象就是仿函数。

仿函数 (since C++ 98，C++ 11 增强)

仿函数类型是一个类型，按照 C++ 标准库的习惯，经常充当模板参数，由标准库代为创建一个对象。例如：

```
priortiy_queue<int, vector<int>, MyGreater> pq;
```

这个时候就是将 `MyGreater` 仿函数类型传入到 `priority_queue` 的模板参数中。表示我们使用 `MyGreater` 仿函数作为比较函数改变原有的比较规则。

仿函数 (since C++ 98，C++ 11 增强)

为了配合仿函数，C++ 11 引入了 `function` 模板类，用于表示函数对象，主要作为函数的参数。

`function` 模板类的形式为 `function<返回值类型(参数类型1, 参数类型2, ...)>`，例如 `MyGreater()` 就可以被 `function<bool(int, int)>` 接受。

下面是一个例子：

```
template<class T>
T get_min(T a, T b, function<bool(T, T)> cmp = less<T>()){
    return cmp(a, b) ? a : b;
}

get_min(1, 2, greater<T>()); // 2
get_min(1, 2, less<T>()); // 1
```

GCC 编译器扩展

GCC/G++ 编译器支持一些扩展。

- `__int128` / `__uint128` 类型：128 位整数类型。范围分别为 $[-2^{127}, 2^{127}-1]$ 与 $[0, 2^{128}-1]$。

`__int128` / `__uint128` 虽然看似好用，但是需要注意：

- `__int128` / `__unit128` 并没有原生的 I/O 支持。需要自己手写输入输出函数。
- 许多内置函数都不支持 `__int128` / `__uint128`。

GCC 编译器扩展

这时候应当给大家看一看史诗级灾难片《联合省选 2024 上的 `abs(__int128)`》。

我在 HNOI 2024 就因为这个问题掉坑了。

GCC 编译器扩展

- `__builtin_clz` / `__builtin_ctz`：计算整数的二进制表示中前导/后导 0 的个数。
- `__builtin_ffs` 计算二进制表示的最低非 0 位的下标（从 1 开始）。
- `__builtin_popcount`：计算整数的二进制表示中 1 的个数。

在这些函数名后，加上 `ll` 后缀，表示传入 `long long` 类型整数，否则最好传入 `int` 类型整数。否则可能会得到意料之外的结果。

这几个函数调用内核级指令，因此速度嘎嘎快，建议合理使用。

GCC 编译器扩展

示例：

```
int x = 10;
__builtin_popcount(x); // 2
long long y = 5;
__builtin_popcountll(y); // 2
__builtin_clzll(y); // 61
```

GCC 编译器扩展

- `__lg(x)` $O(1)$ 的时间复杂度内求得 $\lfloor \log_2 x \rfloor$。
- `__gcd(x, y)` $O(\log n)$ 求得最大公因数。

常数较小。

标准库扩展 1——pbds 平衡树

pbds 提供了许多内置高级数据结构。比较有用的是平衡树 `tree`、堆 `priority_queue` 与哈希表 `gp_hash_table` 与 `cc_hash_table`。

使用标准库扩展，需要引入 `<bits/extc++.h>` 头文件（不包含在 `<bits/stdc++.h>` 头文件中）。

建议用这个类型模板：

```
template<class T, class Compare = less<T> >
using tree = __gnu_pbds::tree<T,  __gnu_pbds::null_type, Compare, __gnu_pbds::rb_tree_tag, __gnu_pbds::tree_order_statistics_node_update>
```

其中，第一个模板参数表示数据类型，第二个模板参数表示映射类型（一般不用特别了解），第三个模板参数表示比较规则的仿函数类型（俗名小于号），第四个模板参数表示平衡树类型（在上面的例子中就是红黑树），第五个模板参数表示节点更新策略，如果不特别指定，则无法查询排名有关信息。

标准库扩展 1——pbds 平衡树

`tree` 的常用成员如下：

- `insert(x)`：插入元素 `x`。
- `erase(x)`：删除元素 `x`。
- `order_of_key(x)`：返回元素 `x` 的排名（从 0 开始）。
- `find_by_order(k)`：返回排名为 `k` 的元素（从 0 开始）。
- `join(x)`：将 x 平衡树合并到当前平衡树，合并后 x 清空。
- `split(x, t)`：将小于等于 x 的元素保留在当前平衡树，将剩余的元素移动到 t 中。
- `empty()` / `size()`：判断是否为空，获取元素个数。
- `lower_bound(x)` / `upper_bound(x)`：与 STL 中的同名函数差不多，不过返回的是迭代器。（这个貌似 useless）

标准库扩展 2——pbds 堆

建议用这个类型模板：

```
template<class T, class Compare = less<T> >
using priority_queue = __gnu_pbds::priority_queue<T, Compare, __gnu_pbds::pairing_heap_tag>;
```

其中，第一个和第二个模板参数比较好理解，第三个模板参数表示堆的类型，`pairing_heap_tag` 表示配对堆，这个是最快的。

标准库扩展 2——pbds 堆

- `push(x)` / `pop()` / `empty()` / `size()` / `top()`：与 STL 的 `priority_queue` 中的同名成员差不多。
- `join(x)` 把 x 合并到当前堆中，然后将 x 清空。

注意 `join` 的时候要注意顺序之类的，所以一般要搭配并查集使用。

标准库扩展 3——pbds 哈希表

pbds 中提供了 `__gnu_pbds::gp_hash_table` 与 `__gnu_pbds::cc_hash_table`。一个是线性探测哈希表，一个是开链哈希表。感觉前者速度更快。

用法和 `unordered_map` 类似，不详细展开。但是这两个玩意比 `unordered_map` 快很多。

标准库扩展 4——rope

rope 是一个类似 `vector` 的容器，但是由于内部是可持久化文艺平衡树，所以无论是随机访问还是插入删除，时间复杂度都是 $O(\log n)$。此外还支持分裂和复制。

可以用下面的代码创建一个 `int` 类型的 `rope`：

```
__gnu_cxx::rope<int> r;
```

标准库扩展 4——rope

- `push_back` / `size` / `empty`：与 STL 中 `vector` 的同名成员差不多。
- `insert(pos, x)` 插入元素 `x` 到 `pos` 位置前，其中 `pos` 是整数。
- `erase(pos)` 删除 `pos` 位置的元素。其中 `pos` 是整数。
- `substr(pos, len)` 截取`pos` 位置起的 `len` 个元素。其中 `pos` 是整数，`len` 是整数。

这玩意网上资料很少，所以这张幻灯片可能有误，请亲自实践。

并且据报告，`rope.erase` 存在 bug，请谨慎使用。

STL 进阶 1——迭代器

迭代器是 C++ 容器的一个概念，用来定位容器中的元素，类似指针，但是比指针更加可靠。

下面的容器以 `vector<int> vct` 为例。

可以用 `vector<int>::iterator` 取得迭代器类型。可以用 `vct.begin()` 取得正向迭代器的第一个，`vct.end()` 取得正向迭代器的终止哨兵（注意不是最后一个）。可以将 `begin` 改为 `rbegin`，`end` 改为 `rend`，这样取得的是逆向迭代器。

迭代器支持自增操作和自减操作，表示将迭代器往后/往前移动一位。

可以使用 `next(it, n = 1)` 取得迭代器 it 后 n 个位置的迭代器（不会改变原迭代器）；
可以使用 `prev(it, n = 1)` 取得迭代器 it 前 n 个位置的迭代器（不会改变原迭代器）。
（Since C++ 11）如果 it 支持随机访问，那么时间复杂度为 $O(1)$，否则为 $O(n)$。

STL 进阶 1——迭代器

由于迭代器是模仿指针设计的，所以可以使用 `*it` 取得迭代器 it 所指元素，也可以使用 `it->member` 取得迭代器 it 所指元素的成员。

STL 进阶 2——序列式容器

`vector` / `array` / `deque` / `list` / `forward_list` 都是序列式容器。

其中 `array` 相当于静态数组，在 OI 中几乎不会用到，故在此不讨论。`list` / `forward_list` 无法支持 OI 中常见的链表结构，故在此亦不讨论。

因此我们重点讲一下 `vector` / `deque`。这两个容器想必大家已经有一定的了解，所以在这里只讲解一下一些大家可能不太清楚的部分。

顺便提一句，STL 容器的 `clear` 不是释放空间，而是将大小置为 0，因此后续插入不需要扩容。如果真的希望释放空间，可以使用 `shrink_to_fit` 成员函数。

STL 进阶 2——序列式容器

- `vector` 相当于一个可变长数组（动态扩容，每次扩容的时候多扩一点空间，然后将整个数组复制到扩容后的部分），因此插入删除的时间复杂度都是 $O(n)$。
- `vector` 支持一个成员函数 `swap(x)` 可以与 `x` 交换，时间复杂度是 $O(1)$。
- 不要使用 `vector<bool>`，原因已经提及了。
- 据说使用 `emplace_***` 比 `push_***` 效率更高。实际上也是如此，但是差别不大。（Since C++ 11）
- `deque` 的常数比较大，建议实用数组模拟。

STL 进阶 3——关联式容器

`set` / `multiset` / `map` / `multimap` 都是关联式容器。

这些东西内部都是使用红黑树来实现的。需要时可以使用，常数大小中等。

如果需要在这些数据结构上二分，请使用成员函数 `lower_bound` 和 `upper_bound`，如果使用 STL 函数，则时间复杂度会退化为 $O(n)$。

这四个容器均原生不支持按照排名查询元素。如果使用 STL 函数 `nth_element`，则时间复杂度是 $O(n)$。

`set` 可以传入比较函数，如 `set<int, vector<int>, MyGreater>`，注意中间的 `vector<int>` 必不可少。

`map<K,V>` 和 `multimap<K,V>` 在 Range-based for loop 中，循环变量是 `pair<K,V>`。

STL 进阶 4——无序关联式容器

`unordered_set` / `unordered_multiset` / `unordered_map` / `unordered_multimap` 都是无序关联式容器。

这些容器内部都是使用哈希表来实现的。需要时可以使用，常数大小中等。但是谨慎使用，可能会被精心设计的数据将时间复杂度卡到 $O(n)$。

无序关联式容器是没有顺序的。

STL 进阶 5——容器适配器

`stack` / `queue` / `priority_queue` 都是容器适配器。

这些东西我相信大家已经很熟悉了。不过要注意前两个东西速度不敢恭维，所以建议手写栈和队列。

不过由于正式考试是开了 O2 优化的，所以其实速度差别也不大。

优先队列一般不用担心常数问题，可以放心使用。

注意 `queue` 的取队首的成员函数是 `front` 不是 `top`。

STL 函数 1——algorithm

- `fill(begin, end, val)` 将 `[begin, end)` 之间的元素全部赋值为 val。

关于 `fill` 的内部实现，我阅读了一下 STL 源码，发现：

- `fill` 函数对 `vector<bool>` 有特殊的优化。此时 `fill` 类似于 `memset`。
- `fill` 函数对于 `char`、`signed char`、`unsigned char` 数组有特殊优化。同样像 `memset`。
- `fill` 对于其他情况，真的是运行一遍循环赋值的。

```
template<typename _ForwardIterator, typename _Tp>
inline typename __enable_if<!__is_scalar<_Tp>::__value, void>::__type
__fill_a(_ForwardIterator __first, _ForwardIterator __last, const _Tp& __value){
    for (; __first != __last; ++__first) *__first = __value;
}
```

STL 函数 1——algorithm

- `min_element(begin, end)` / `max_element(begin, end)` 找到 `[begin, end)` 中的最小/最大元素。

给一下 `min_element` 的实现：

```
template<typename _ForwardIterator, typename _Compare>
constexpr _ForwardIterator __min_element(_ForwardIterator __first, _ForwardIterator __last, _Compare __comp){
    if (__first == __last) return __first;
    _ForwardIterator __result = __first;
    while (++__first != __last)
        if (__comp(__first, __result)) __result = __first;
    return __result;
}
```

可以看出确实是 $O(n)$ 的，并且几乎没有额外常数开销。

STL 函数 1——algorithm

- `copy(begin, end, out)` 将 `[begin, end)` 中的元素拷贝到首迭代器为 `out` 的容器中，返回 `out` 的尾迭代器。

时间复杂度是 $O(n)$。实现：

```
template<typename _II, typename _OI>
static _OI __copy_m(_II __first, _II __last, _OI __result){
        for (; __first != __last; ++__result, (void)++__first) *__result = *__first;
        return __result;
}
```

STL 函数 1——algorithm

- `stable_sort(begin, end, [cmp])` 将 `[begin, end)` 中的元素排序，排序规则由 `cmp` 决定（默认为 `<`）。

这个排序内部是归并排序，需要额外的空间，但是满足稳定排序的性质。

STL 函数 1——algorithm

- `nth_element(begin, end, k, [cmp])` 将 `[begin, end)` 中小于 `k` 的放在前面，大于 `k` 的放在后面。

时间复杂度和空间复杂度是 $O(n)$。

STL 函数 1——algorithm

- `next_permutation(begin, end)` / `prev_permutation(begin, end)` 找到下一个/上一个排列（其实不一定是排列，只要是满足字典序规则的就可以）。
- `lower_bound(begin, end, val, [cmp])` 找到 `[begin, end)` 中第一个大于等于 `val` 的元素。
- `upper_bound(begin, end, val, [cmp])` 找到 `[begin, end)` 中第一个大于 `val` 的元素。
- `sort(begin, end, [cmp])` 对 `[begin, end)` 中的元素进行排序。
- `reverse(begin, end)` 将 `[begin, end)` 中的元素翻转。
- `unique(begin, end)` 将 `[begin, end)` 中的元素去重。

STL 函数 1——algorithm

- `shuffle(begin, end)` 将 `[begin, end)` 中的元素随机打乱。

注意 `random_shuffle` 已经于 C++ 14 弃用，这玩意并不能保证均匀随机性。

`shuffle` 与 `random_shuffle` 时间复杂度均为 $O(n)$。

STL 函数 1——algorithm

- `merge(begin1, end1, begin2, end2, out)` 将两个有序容器合并成一个有序容器，输出到首迭代器为 `out` 的容器中。时间复杂度是 $O(n)$。

实现如下：

```
template <typename _InputIterator1, typename _InputIterator2,typename _OutputIterator, typename _Compare>
_OutputIterator
__merge(_InputIterator1 __first1, _InputIterator1 __last1,
        _InputIterator2 __first2, _InputIterator2 __last2,
        _OutputIterator __result, _Compare __comp) {
    while (__first1 != __last1 && __first2 != __last2) {
        if (__comp(__first2, __first1)) {
            *__result = *__first2;
            ++__first2;
        } else {
            *__result = *__first1;
            ++__first1;
        }
        ++__result;
    }
    return std::copy(__first2, __last2,
                     std::copy(__first1, __last1, __result));
}
```

STL 函数 1——algorithm

- `inplace_merge(begin, mid, end)` 将 `[begin, mid)` 和 `[mid, end)` 合并成一个有序容器存入 `begin` 中。这玩意是原地的，时间复杂度是 $O(n)$，跑得飞快。
- `count(begin, end, val)` 统计 `[begin, end)` 中等于 `val` 的元素个数。
- `replace(begin, end, val, new_val)` 将 `[begin, end)` 中等于 `val` 的元素替换成 `new_val`。
- `rotate(begin, mid, end)` 将 `[begin, end)` 中的元素旋转，使得原本在 `mid` 位置的元素移动到 `begin` 位置。

STL 函数 1——algorithm

- `count_if(begin, end, func)` 统计 `[begin, end)` 中满足 `fun(c)` 为真的元素个数。
- `copy_if(begin, end, out, func)` 将 `[begin, end)` 中满足 `func(c)` 为真的元素拷贝到首迭代器为 `out` 的容器中。

STL 函数 2——numeric

- `accumulate(begin, end, init, [op])` 将 `[begin, end)` 中的元素累加（初始值为 `init` ，运算符为 `op` ）。
- `partial_sum(begin, end, out, [op])` 将 `[begin, end)` 中的元素求前缀和，将结果拷贝到首迭代器为 `out` 的容器中。
- `inner_product(begin1, end1, begin2, init, [op1], [op2])` 将 `[begin1, end1)` 和 `[begin2,end2)` 中的元素对应一一相乘（ `op2` ），最后累加（ `op1` ）起来。
- `adjacent_difference(begin, end, out, [op])` 将 `[begin, end)` 中的元素求前缀差分，将结果拷贝到首迭代器为 `out` 的容器中。 `op` 表示减法运算。

以上 `op` 均为一个函数或仿函数对象。

STL 函数 2——numeric

- `iota(begin, end, val)` 将 `[begin, end)` 中的元素依次赋值为 `val`，`val + 1`，`val + 2`……

random

rand 函数并不适合生成随机数，因为这玩意不但有循环节，而且值域还很小，且不保证均匀随机性。

在 C++ 中，可以使用 `mt19937` 生成 int 范围的随机数，`mt19937_64` 生成 long long 范围的随机数。

比如：

```
mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
rnd() // 可能得到 114514
```

其中 `chrono::steady_clock::now().time_since_epoch().count()` 是一个纳秒级别的时间戳，建议用于替代 `time(NULL)`。

random

对于生成 $[l, r]$ 范围的随机数，使用取模勉强可以，但是不够均匀。如果要生成严格均匀的随机数，可以使用 `uniform_int_distribution`。

```
mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
uniform_int_distribution<int> dis(1, 2);
dis(rnd); // 可能得到 1
```

类似的，还有 `uniform_real_distribution`。表示$[l, r]$ 的浮点数。

此外还有其他的分布方法，不过在 OI 中，这些方法基本用不到。

模板黑魔法 1——模板初步

模板可以理解成，你有一套类型或变量或函数的模具，你可以往模具里面留空，然后使用的时候把空缺的东西填进去，然后就可以得到你想要的东西。使用模板可以节省开发效率。

考虑到可能有些同学可能没有接触过模板，所以在下一页幻灯片中我们给一个模板的例子。

模板黑魔法 1——模板初步

```cpp
template<class T, int N>
class Array {
    T data[N];
    T& operator[](int i) { return data[i]; }
}; // 模板类

template<class T>
typename T::value_type sum(T& a) {
    typename T::value_type res = a[0];
    for (unsigned i = 1; i < a.size(); i++) res += a[i];
    return res;
} // 模板函数，typename 用于表示后面的标识符是一个合法的类型，要不然编译器会报错

template<class T>
constexpr bool is_int = false;
template<>
constexpr bool is_int<int> = true;// 变量模板
```

模板黑魔法 1——模板初步

一般来说，使用模板需要指定所有模板参数。但是 C++ 引入了模板参数推导，可以推断出一些模板参数。

例如：

```
vector<int> a;
sum<vector<int> >(a);
```

可以直接写成：

```
vector<int> a;
sum(a);
```

模板黑魔法 1——模板初步

利用变量模板，我们可以进行一些类型的判断，例如：

```
template<class T>
constexpr bool is_string = false;

template<>
constexpr bool is_string<string> = true;
```

这一操作被称为模板的特化，即对于模板的特殊情况给予特殊的处理。

模板黑魔法 1——模板初步

变量模板不一定需要为 `bool` 类型，所以可以用它进行一些编译期的计算。

请大家试一试用模板计算 2^{10}（通过一个模板计算 2^x）

模板黑魔法 1——模板初步

下面是一个可能的例子：

```
template<int N>
constexpr int pow2 = pow2<N-1> * 2;

template<>
constexpr int pow2<0> = 1;

int answer = pow2<10>;
```

可以看到编译后的汇编为：

```
answer:
    .long       1024
```

模板黑魔法 2——Type Traits

Type Traits 是 C++ 11 引入的一些模板，用于获取类型信息。

使用 type traits 需要引入 `<type_traits>` 头文件。

模板黑魔法 2——Type Traits

判断类型：

- `is_void<T>`，`is_integral<T>`，`is_floating_point<T>` 判断 `T` 是否为 `void`、整型、浮点型。
- `is_pointer<T>`，`is_reference<T>`，`is_array<T>` 判断 `T` 是否为指针、引用、数组。
- `is_const<T>` 判断 `T` 是否为常量。

模板黑魔法 2——Type Traits

下面是一个使用判断类型的示例：

```
is_void<void>::value;// true
is_integral<int>::value;// true
is_floating_point<double>::value;// true
is_pointer<int*>::value;// true
is_reference<int&>::value;// true
is_array<int[]>::value;// true
is_const<const int>::value;// true
```

模板黑魔法 2——Type Traits

在这里我们顺便介绍一个关键字 `decltype`。这个关键字可以获取表达式的类型。例如：

```
is_integeral<decltype(1+2)>::value;// true
```

`declval<T>` 位于 `<utility>` 中，可以获取一个 `T` 类型的临时对象（不会真的调用构造函数，类型为 `T&&`），可以配合 `decltype` 获取类型。如：

```
is_same<decltype(declval<int>() + 1), int>::value;// true
```

模板黑魔法 2——Type Traits

- `is_same<T1, T2>` 判断 `T1` 和 `T2` 是否为同一类型。
- `enable_if<B, T>` 如果条件 `B` 为真，则返回 `T` 类型，否则返回一个特殊的空类型 `enable_if_t<false>`，这个类型无法用于创建对象或函数返回类型，因此当 `B` 为假时，任何试图使用 `enable_if<B, T>` 的地方都会因类型替换失败而被编译器排除。这个现象称为 SFINAE（Substitution Failure Is Not An Error）。需要用 `enable_if<B, T>::type` 来获取结果。

此外 type_traits 还可以进行类型转换（如 `remove_pointer<T>`、`remove_reference<T>`、`remove_const<T>`、`add_pointer<T>`、`add_reference<T>`、`add_const<T>` 等），由于这些东西在 OI 上很少使用，所以就不一一介绍了。

模板黑魔法 2——Type Traits

例题：请编写一个函数 `constexpr int check(T x)`：

- 如果 x 为整型（如 int、long long），返回 0。
- 如果 x 为浮点型（如 double、long double），返回 1。
- 如果 x 为字符串（string 对象），返回 2。
- 如果 x 为指针或引用，返回 3。
- 如果均不满足，则编译错误。

模板黑魔法 2——Type Traits

下面是一个可能的实现：

```
template<class T>
constexpr typename enable_if<is_integral<T>::value, int>::type check(T x){ return 1; }
template<class T>
constexpr typename enable_if<is_floating_point<T>::value, int>::type check(T x){ return 2; }
template<class T>
constexpr typename enable_if<is_same<T, string>::value, int>::type check(T x){ return 3; }
template<class T>
constexpr bool is_pr = false;
template<class T>
constexpr bool is_pr<T*> = true;
template<class T>
constexpr bool is_pr<T&> = true;
template<class T>
constexpr typename enable_if<is_pr<T>,int>::type check(T x){ return 4; }
```

其实 is_pr 也可以用 bool 运算逻辑或来代替。

模板黑魔法 2——Type Traits

例题：请编写一个函数 `constexpr int check(T x)`，当 `T` 支持运算 `T + T` 时返回 0，否则编译错误。

模板黑魔法 2——Type Traits

下面是一个可能的实现：

```
template<class U>
constexpr bool is_addable = is_same<decltype(declval<U>() + declval<U>(), 1), int>::value;
template<class T>
constexpr typename enable_if<is_addable<T>, int>::type check(T x) { return 0; }
```

模板黑魔法 3——可变长参数模板与类型别名

这个东西比较好理解，放一个示例代码就行了：

```
template<typename T>
constexpr auto sum(T v){ return v; }
template<typename T,typename... Args>
constexpr auto sum(T v, Args... args){
    return v + sum(args...);
}
int x = sum(1, 2, 3);
int y = sum(1, 2.0, 3);
```

模板黑魔法 3——可变长参数模板与类型别名

在 C 语言中，可以使用 `typedef` 来定义类型别名。在 C++ 中，可以使用 `using` 来定义类型别名。

例如：

```
using int_t = int;
using vi = vector<int>;
```

区别是 `using` 可以带模板：

```
template<typename T>
using vt = vector<T>;

vt<int> v;// 等价于 vector<int> v
```

模板黑魔法 4——模板的偏特化

类模板和变量模板支持偏特化。函数模板不支持。

偏特化区别于特化，不需要给定所有的模板参数，只需要模板参数满足一定的条件就构成偏特化。

匹配时会匹配最特殊的偏特化，如果有多个，则会编译错误。

下面是一个简单的例子：

```
template<class K, class V>
constexpr bool check = false;

template<class K>
constexpr bool check<K, int> = true;

template<class V>
constexpr bool check<int, V> = true;
```

模板黑魔法 4——模板的偏特化

上述例子中，如果 K,V 中有一个是 int，那么会让 check<K,V> 为真，否则为假。

但是假如 K,V 都是 int，那么 check<K,V> 不知道该使用哪一个偏特化，因此会产生错误。

可以添加一个完全特化：

```
...省略上述代码...
template<>
constexpr bool check<int,int> = true;
```

或者干脆不使用模板偏特化，直接用 `enable_if` 完成也行。

模板黑魔法 5——模板元编程

例题：请编写一个类型别名 `swap_kv<T>`，接受一个 `T=Tp<K,V>`，得到 `Tp<V,K>`。

其中 `Tp` 是某一个类。

模板黑魔法 5——模板元编程

下面是一个可能的实现：

```
template<class T>
struct swap_kv_cls{using type = T;};
template<template<class,class>class Tp, class A, class B>
struct swap_kv_cls<Tp<A,B> >{using type = Tp<B,A>;};
template<class T>
using swap_kv = typename swap_kv_cls<T>::type;
```

注意别名不能特化，所以只能用一个结构体完成。

模板黑魔法 5——模板元编程

例题：请定义一个 `array<int, 11>`，第 i 项元素为 2^i（i 从 0 开始）。

要求数组每一项的值在编译期计算。

hint: `array` 支持类内部静态初始化，例如：

```
struct A{
    static constexpr int a[10] = {1,2,3,4,5,6,7,8,9,10};
};// 错误
struct B{
    static constexpr array<int, 10> a = {1,2,3,4,5,6,7,8,9,10};
};// 正确
```

不允许使用 `constexpr` 函数。

模板黑魔法 5——模板元编程

下面是一个可能的实现：

```
template<int N>
constexpr int pow2 = pow2<N-1> * 2;

template<>
constexpr int pow2<0> = 1;

template<int pos = 0, int ...val>
struct cls : cls<pos + 1, val..., pow2<pos> > {};

template<int ...val>
struct cls<11, val...>{
    static constexpr array<int, 11> vals = {val...};
};

constexpr auto vals = cls<>::vals;
```

模板黑魔法 5——模板元编程

模板元编程还有更多的应用，不过在此限于篇幅，不详细展开。有兴趣的同学请自行研究。

Thanks for listening!