

## 第九章 C++实用技巧与模版库

在本章中将介绍各种各样实用的提高我们编程效率的小技巧和模版库。其中绝大多数依靠 C++ 自带类和函数实现。

### 第一节 排序算法

排序算法为竞赛中最常用的算法之一，我们可以利用 C++ 自带的库函数进行排序。

使用排序算法必须包含 `algorithm` 头文件

自带排序算法的一般形式为：

//将数组 `arr` 的下标为 `m` 的元素到下标为 `n-1` 的元素进行从小到大排序

```
sort(arr+m,arr+n);
```

`sort(arr+m,arr+n,comp);` //与 `sort(arr+m,arr+n)` 相比，这个写法可以自己定义排序的规则

//其中，`comp` 为自定义的函数

对于 `sort(arr+m,arr+n)` 我们举个简单的例子：

//这个程序实现从键盘读入 10 个数，然后从小到大输出的功能

```
#include<iostream>
```

```
#include<algorithm>
```

```
using namespace std;
```

```
int a[10];
```

```
int main()
```

```
{
```

```
    for (int i=0;i<10;++i) cin>>a[i];
```

```
    sort(a+0,a+10);
```

```
    for (int i=0;i<10;++i) cout<<a[i]<<' ';
```

```
    cout<<endl;
```

```
    return 0;
```

```
}
```

当然，有时我们需要从大到小的进行排序。那么我们可以用 `sort(arr+n,arr+m,comp)` 进行排序。

不过，在调用 `sort(arr+n,arr+m,comp)` 之前我们需要自己写个 `comp` 函数。

从大到小排序的 `comp` 函数可以这样写：

```
int my_comp(const int & a,const int & b)
```

```
{
```

```
    return a>b;           //在两元素相同时一定要返回 0 或者 false
```

```
} //comp 函数的名字是由我们自己决定的，例如可以叫 cat_cat,dog_dog 等等
```

//程序实现从键盘读入 10 个数，然后从大到小输出的功能

```
#include<iostream>
```

```
#include<algorithm>
```

```

using namespace std;
int a[10];
int my_comp(const int & a,const int & b)
{
    return a>b;    //如果 a>b 则返回 1 ， 否则返回 0
}
int main()
{
    for (int i=0;i<10;++i) cin>>a[i];
    sort(a+0,a+10,my_comp);
    for (int i=0;i<10;++i) cout<<a[i]<<' ';
    cout<<endl;
    return 0;
}

```

在更多情况下，我们不仅对一个特征进行排序，而是多个特征。例如将学生的成绩进行排序，当然用上面的做法是行不通的。这时，我们就想到了结构体这种数据类型。当我们采用 `sort()` 函数的默认规则排序结构体时，`sort()` 默认结构体中的第一个成员为第一关键字，第二个成员为第二关键字，……，第  $N$  个元素为第  $N$  关键字，然后从小到大排序。

例如我们要将学生的成绩从大到小排序，当成绩相同时，根据姓名字典序小的优先的规则排进行序。显然我们无法采用默认规则进行排序。

这时我们可以定义这样的 `comp`:

```

int score_comp(const student & a,const student & b)
{
    if (a.score>b.score) return 1;
    if (a.score<b.score) return 0;
    if (a.name<b.name) return 1;
    return 0;
}

```

**例 8.1** 每当考试结束，老师总要对学生的成绩进行排序，以便研究学生学习情况。老师发现学生数目庞大，所以找来了会编程的你，并且它给你了全部同学的分，希望你能按如下规则排序：1.分数高的排在前面；2.如果分数相同，就把名字字典序低的放在前面。

#### 【输入格式】

第一行一个整数  $n$

接下来  $n$  行每行一个学生名字  $name_i$  和一个分数  $score_i$ 。

#### 【输出格式】

每行一个名字和一个分数

#### 【输入样例】

```

3
Xiaoxiao 396
Yingmo 405
Diyucailang 399

```

#### 【输出样例】

Yingmo 405

Diyucailang 399

Xiaoxiao 396

#### 【参考程序】

```
#include<iostream>
#include<algorithm>
#include<string>
using namespace std;
struct student{
    int score;
    string name;
}a[100];
int n;

int score_comp(const student & a,const student & b)
{
    if (a.score>b.score) return 1;
    if (a.score<b.score) return 0;
    if (a.name<b.name) return 1;
    return 0;
}

int main()
{
    cin>>n;
    for (int i=0;i<n;++i)
    {
        cin>>a[i].name;
        cin>>a[i].score;
    }
    sort(a+0,a+n,score_comp);
    for (int i=0;i<n;++i)
        cout<<a[i].name<<' '<<a[i].score<<endl;
    return 0;
}
```

## 第二节 重载运算符

在之前的高精度运算的章节，我们学习过使用高精度算法进行高精度数的计算。当我们多次使用高精度时，我们会把它写成函数。或许你会疑问，我们难道不能向像 `int a=3,b=4,c; c=a+b;` 一样进行运算吗？答案是肯定的。接下来我们将介绍如何利用重载运算符进行高精计算。

为方便学习，我们将假设我们所进行运算的数字为非负整数。

首先我们需要定义一个结构，使得我们的高精数可以存储。

```
const int MAXN=4000;    //高精度数的长度
struct BIGNUM {
    int len,s[MAXN];
    BIGNUM () {len=1;memset(s,0,sizeof(s)); }
};
```

结构体里面有一个 BIGNUM()的函数，它称为“构造函数 (Constructor)”。构造函数是 C++中特有的，作用就是进行初始化。每当我们定义一个 BIGNUM 类型的变量时，这个变量就会被初始化。例如定义 BIGNUM x 时，我们将会得 x.len==1,x.s 中的所有元素均为 0。

我们知道，结构体之间是可以相互赋值的。但我们如何能用“=”将 字符串数字赋给 BIGNUM 的变量？很简单，只要在**结构体里面**写如下运算符重载：

```
BIGNUM operator = (const char* num) {          //operator 是关键字
    len=strlen(num);
    for (int i=0;i<len;++i) s[i]=num[len-i-1]-'0';
    return *this;
}
```

当我们可以写 例如 x=”77088013145207708801314520” 的时候，我们肯定还想将 1314520 这种较小的非字符串数字赋值给 x。这时需要在**结构体里面**写：

```
BIGNUM operator = (const int num) {
    char a[MAXN];
    sprintf(a,"%d",num);
    *this = a;
    return *this;
}
```

肯定会有人看到这里时非常兴奋，忍不住敲一遍代码，一探究竟。当他敲完上面代码并打了 BIGNUM x=1;后，他一定会发现，编译失败。这是因为初始化和赋值是不同的概念，编译器处理这两者的方式也不同，所以我们需要在**结构体里面**加上：

```
BIGNUM (int num) { *this=num; }
BIGNUM (const char * num) { *this=num; }
```

当我们解决了结构体的数据定义，赋值，初始化，我们需要将它们输出，我们可以使用重载“<<”运算符进行输出,注意这个重载需要写在**结构体外面**：

```
ostream& operator << (ostream &out,const BIGNUM& x) {
    for (int i=x.len-1;i>=0;--i)
        cout<<x.s[i];
    return out;
}
```

重载完“<<”，顺便重载下“>>”，和重载“<<”时一样，要写在**结构体外面**：

```
istream& operator >> (istream &in,BIGNUM &x) {
    char num[MAXN];
    in>>num;
```

```

x=num;
return in;
}

```

我们已经可以实现高精度数的存储，赋值，初始化，输入，输出了。现在一定迫不及待的想知道如何实现高精度的运算了吧。由于篇幅有限，本文仅展示”+”、”+=”重载。这个也是写在**结构体里面**：

```

BIGNUM operator + (const BIGNUM & a) {
    BIGNUM c;
    c.len=max(len,a.len)+1;           //默认两个数字相加有进位
    for (int i=0,x=0;i<c.len;++i) {
        c.s[i]=s[i]+a.s[i]+x;
        x=c.s[i]/10;
        c.s[i]=c.s[i]%10;
    }
    if (c.s[c.len-1]==0) --c.len;      //如果没有进位长度就减 1
    return c;
}

BIGNUM operator += (const BIGNUM & a) {
    *this = *this+a;
    return *this;
}

```

对于数字，我们不仅会对它们进行计算，有时我们还会对它们进行比较，当然这个也要进行运算符重载，依然是在**结构体里面**补充相关代码：

我们先重载下 “<” 然后利用 “<” 重载出其他关系运算符

```

bool operator < (const BIGNUM & x) const {
    if (len != x.len) return len<x.len;    //注意不能有前导零
    for (int i=len-1;i>=0;--i) {
        if (s[i] != x.s[i]) return s[i]<x.s[i];
    }
    return false;                          //全部都相等，说明小于不成立
}

```

现在用”<”来表示其他的的关系运算符：

```

bool operator > (const BIGNUM & x) const { return x<*this; }
bool operator <= (const BIGNUM & x) const { return !(x<*this); }
bool operator >= (const BIGNUM & x) const { return !(*this<x); }
bool operator == (const BIGNUM & x) const { return !(x<*this||*this<x); }
bool operator != (const BIGNUM & x) const { return x<*this||*this<x; }

```

**例 8.2** 运用重载的例子: 给出 n 和 m, 求出  $1!+2!+3!+\dots+n!$  是否大于 m. 大于 m 的话输出”yes”, 否则输出”no”. 其中  $0<n<101, m<10^500$ .

**【输入格式】**

只有一行，包括一个  $n$  和一个  $m$

**【输出格式】**

如果  $1!+2!+3!+\dots+n!$  大于  $m$ ，则输出“yes”，否则输出“no”

**【输入样例】**

33 8954945705218228090637347680100940313

**【输出样例】**

no

**【参考程序】**

```
#include<cstdio>
#include<cstring>
#include<iostream>
#include<cmath>
using namespace std;
const int MAXN=4000;
struct BIGNUM {
    int len,s[MAXN];
    BIGNUM () {memset(s,0,sizeof(s)); len=1;}
    BIGNUM operator = (const char* num) {
        len=strlen(num);
        for (int i=0;i<len;++i) s[i]=num[len-i-1]-'0';
        return *this;
    }
    BIGNUM operator = (const int num) {
        char a[MAXN];
        sprintf(a,"%d",num);
        *this = a;
        return *this;
    }
    BIGNUM (const int num) { *this=num; }
    BIGNUM (const char * num) { *this=num; }
    BIGNUM operator + (const BIGNUM & a) {
        BIGNUM c;
        c.len=max(len,a.len)+1;
        for (int i=0,x=0;i<c.len;++i) {
            c.s[i]=s[i]+a.s[i]+x;
            x=c.s[i]/10;
            c.s[i]=c.s[i]%10;
        }
        if (c.s[c.len-1]==0) --c.len;
        return c;
    }
    BIGNUM operator += (const BIGNUM & a) {
```

```

        *this = *this+a;
    return *this;
}

BIGNUM operator * (const BIGNUM & x) {
    BIGNUM c;
    c.len=len+x.len;
    for (int i=0;i<len;++i) {
        for (int j=0;j<x.len;++j) {
            c.s[i+j]+=s[i]*x.s[j];
            c.s[i+j+1]+=c.s[i+j]/10;
            c.s[i+j]%=10;
        }
    }
    if (c.s[c.len-1]==0) --c.len;
    return c;
}

BIGNUM operator *= (const BIGNUM & a) {
    *this = *this * a;
    return *this;
}

bool operator < (const BIGNUM & x) const {
    if (len != x.len) return len<x.len;
    for (int i=len-1;i>=0;--i) {
        if (s[i] != x.s[i]) return s[i]<x.s[i];
    }
    return false;
}

bool operator > (const BIGNUM & x) const { return x<*this; }
bool operator <= (const BIGNUM & x) const { return !(x<*this); }
bool operator >= (const BIGNUM & x) const { return !(*this<x); }
bool operator == (const BIGNUM & x) const { return !(x<*this||*this<x); }
bool operator != (const BIGNUM & x) const { return x<*this||*this<x; }

};

ostream& operator << (ostream &out,const BIGNUM& x) {
    for (int i=x.len-1;i>=0;--i)
        cout<<x.s[i];
    return out;
}

istream& operator >> (istream &in,BIGNUM &x) {
    char num[MAXN];
    in>>num;
    x=num;
}

```

```

        return in;
    }

    BIGNUM f[5001];

int main()
{
    int n;
    BIGNUM m,sum=0,num=1;
    cin>>n>>m;
    for (int i=1;i<=n;++i) {
        num*=i;
        sum+=num;
    }
    if (sum>m) cout<<"yes"<<endl;
    else cout<<"no"<<endl;
    return 0;
}

```

### 第三节 字符串(string)

在运用 C++ 字符数组进行字符串操作时，我们习惯性会把 字符串 s1 的值赋值给字符串 s2 写成 s2=s1，这毋庸置疑是错的。不过，接下来介绍的一种数据类型——字符串类型(string)，它允许写 s2=s1 的赋值方式。当然，字符串类型与字符数组类型是完全不同的类型，两者不要弄混了。

在使用 string 类型时，我们必须包含 string 头文件。

string 类型的定义与初始化

```

string s1;                //定义一个字符串 s1，并初始化为空
string s2(s1);            //用 s1 初始化 s2
string s3("value");       //将 s3 初始化为"value"
string s4(n,'c');         //将 s4 初始化为字符'c'的 n 个副本

```

string 类型的读写

string 类型的读写就像其它类型的读写一样，使用 cin,cout。不过值得注意的是，使用 cin 读入 string 类型，就像用 scanf();读字符数组一样，忽略开头的(制表符、换行符、空格)，当再次碰到空字符就停止（并不会读取空字符）。

读入未知数目的 string 对象：

```

#include<string>
#include<iostream>
using namespace std;
int main() {
    string s;
    int tot=0;

```





```
string s2(“world\n”);
s1+=s2 //等价于 s1=s1+s2, 此时 s1=“hello world\n”
```

字符串 “+” 操作符的左右操作数必须有一个为字符串类型。

```
string s1= "hello";
string s2="world";
string s3=s1+", "; //合法
string s4="hello "+"world"; //非法
string s5=s1+", "+"world"; //合法, 根据从左至右的结合法则
//s1+", "变成字符串类型, 然后再与"world"相连。
string s6="hello"+"", "+ s2; //非法, 因为"hello"与", "都是字符数组类型
```

字符串的下标操作:

```
string str="2+2=4";
cout<<str[1]<<endl; //将输出'+'
str[1]='*'; //现在 str[1]变成了'*'
```

其它一些常用操作:

s.insert(pos,s2)	在 s 下标为 pos 的元素前插入 string 类型 s2
s.substr(pos,len)	返回一个 string 类型, 它包含 s 中下标为 pos 起的 len 个字符
s.erase(pos,len)	删除 s 中下标为 pos 开始的 len 个字符
s.replace(pos,len,s2)	删除 s 中下标为 pos 的 len 个字符, 并在下标为 pos 处插入 s2
s.find(s2,pos)	在 s 中以 pos 位置起查找 s2 第一次出现的位置, 若查找不到返回 string::npos
s.c_str()	返回一个与 s 面值相同的 C 风格的字符串临时指针

将字符串类型的变量转化为数字:

```
//错误方法
#include<cstdio>
#include<string>
#include<iostream>
using namespace std;
int main() {
    string s1="1234",s2="4321";
    cout<<s1+s2<<endl;
    return 0;
}
```

正如前面所讲 s1+s2 是将两个字符串类型相连

```
//正确方法
#include<cstdio>
#include<string>
#include<iostream>
using namespace std;
int main() {
```

```

string s1="1234",s2="4321";
int a,b;
sscanf(s1.c_str(),"%d",&a);           //sscanf()的作用是从字符数组中读入
sscanf(s2.c_str(),"%d",&b);
cout<<a+b<<endl;
return 0;
}

```

同样也可以利用 `sprintf()` 将数据输入到 `sprintf()`。 `sscanf()/sprintf()` 的用法与 `scanf()/printf` 相同，只是在参数中第一个加个字符数组。

## 第四节 FIFO 队列和优先队列

在实际应用中，我们将会大量使用到 FIFO 队列和优先队列，本节将介绍这两个 C++ 自带算法。当然，使用他们必须包含 `queue` 头文件。

定义 FIFO 队列

`queue<类型名> 变量名`

`queue<int> que` //定义 que 为一个 int 类型的 FIFO 队列

`queue<char> a` //定义 a 为一个 char 类型的 FIFO 队列

`queue<data> c` //定义 c 为一个 data 类型的 FIFO 队列

//其中 data 为自定义的数据类型，可以为结构体

定义简单的优先队列

`priority_queue <int> heap;` //定义 heap 为一个 int 类型的优先队列

`priority_queue <double> k;` //定义 k 为一个 double 类型的优先队列

这两种定义方式都是大根堆，想要使其变成小根堆，可以将每个数据都乘以 -1

定义结构体的优先队列

```

struct data{
    int x;
    bool operator < (const data & a)const { //const 一定要写
        return a.x<x; //等于小根堆
    }
};

```

`priority_queue <data>q;` //q 优先队列的优先规则由 data 重载小于号决定

FIFO 队列和优先队列的操作

<code>q.empty()</code>	若队列为空，则返回 true, 否则返回 false
<code>q.size()</code>	返回队列中元素的个数
<code>q.pop()</code>	删除队首元素，但不返回其值
<code>q.front()</code>	返回队首元素的值，但不删除该元素（仅适用于 FIFO 队列）

q.back()	返回队尾元素的值，但不删除该元素（仅适用于 FIFO 队列）
q.top()	返回具有最高优先级的元素的值，但不删除该元素（仅适用于优先队列）
q.push()	对 queue，在队尾压入一个新元素 对于 priority_queue,在基于优先级的适当位置插入新元素

利用 C++自带优先队列的合并果子代码：

```
#include<cstdio>
#include<iostream>
#include<queue>
using namespace std;
priority_queue <int> que;
int main()
{
    int n;
    scanf("%d",&n);
    for (int i=0,x;i<n;++i) {
        scanf("%d",&x);
        que.push(-x);
    }
    int ans=0;
    for (int i=1,tmp;i<n;++i) {
        tmp=que.top();
        ans-=que.top();
        que.pop();
        tmp+=que.top();
        ans-=que.top();
        que.pop();
        que.push(tmp);
    }
    cout<<ans<<endl;
    return 0;
}
```

## 第五节 动态数组

为了节省空间，有时我们会使用动态数组 `vector`。

定义动态数组

`vector<类型名>变量名`

`vector<int>que` //定义 `que` 为一个 `int` 类型的动态数组

`vector<char> a` //定义 `a` 为一个 `char` 类型的动态数组

`vector<data> c` //其中 `data` 为自定义的数据类型，可以为结构体

`vector` 动态数组的操作

<code>a[i]</code>	返回动态数组中的第 <code>i</code> 个元素
<code>a.empty()</code>	若动态数组为空，则返回 <code>true</code> , 否则返回 <code>false</code>
<code>a.size()</code>	返回动态数组中元素的个数
<code>a.resize()</code>	修改动态数组大小
<code>a.push_back()</code>	向动态数组尾部插入一个元素
<code>a.pop_back()</code>	删除动态数组尾部的一个元素
<code>a.begin()</code>	返回指向 <code>vector</code> 头部的迭代器（指针）
<code>a.end()</code>	返回指向 <code>vector</code> 尾部元素的后一个元素的迭代器（指针）

利用 C++ 动态数组实现的排序代码：

```
#include <iostream>
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
    vector<int> a;//动态数组
    int n;
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
    {
        int tmp;
        scanf("%d",&tmp);
        a.push_back(tmp);
    }
```

```

sort(a.begin(),a.end());
for(int i=0; i<a.size(); i++)printf("%d ",a[i]);
printf("\n");
return 0;
}

```

## 第六节 关联式容器 map

在实际应用中，我们可以使用 map 容器来作为一个有序的映射表，可以将其看做是一个下标可以是任何类型的数组。对 map 单次操作的时间复杂度为  $O(\lg n)$ 。

定义 map

map<类型 1,类型 2>变量名;

map<string,int> ma; //定义 ma 为一个从 string 到 int 的一个映射

访问 map 中的元素

map<string,int> ma; //定义 ma

ma["abc"]=2; //将字符串"abc"映射到整数"2"上

cout<<ma["abc"]<<endl; //屏幕上将输出整数 2

同时，map 中的类型可以是自己定义的结构体，此时结构体中应该有重载小于符号。

map 的操作：

operator[]	访问 map 中的元素，若该元素不存在，将创建一个新元素并将该元素映射到类型 2 的初始值上（对于 int 类型，初始值为 0）
ma.begin()	返回 map 中第一个元素的迭代器（指针）
ma.end()	返回 map 中最后一个元素的后一个元素的迭代器（指针）
ma.size()	返回 map 中元素个数
ma.count(element)	若元素 element 存在于 map 中返回 1，否则返回 0
ma.clear()	初始化 map
ma.lower_bound()	返回键值大于等于给定元素的第一个位置

注意：一旦 map 中的一个元素被访问，不论它之前是否已经被赋值，它都将被视为已经存在，例如：

```
if(ma["abc"]) /*do something*/;
```

```
if(ma.count("abc"))cout<<"yes"<<endl;
```

```
else cout<<"no"<<endl; //本段程序运行后屏幕上将输出"yes"
```

示例代码：

```
#include <iostream>
```

```
#include <map>
```

```
#include <string>
```

```
using namespace std;
```

```
map<string,int> ma;
```

```
int main()
```

```
{
```

```
    ma["apple"]=1;
```

```
    ma["banana"]=2;
```

```
    ma["lemon"]=3;
```

```
    cout<<ma.size()<<endl; //输出 map 的大小 3
```

```
    cout<<ma["apple"]<<endl; //输出 1
```

```
    if(ma.count("pear"))cout<<"pear"<<endl;
```

```
else cout<<"no pear"<<endl;          //输出 "no pear"  
cout<<ma.lower_bound("cow")->first<<endl;      //输出 "lemon"  
cout<<ma.lower_bound("cow")->second<<endl;  
                                         //输出 ma["lemon"], 即输出 3  
  
return 0;
```

}