# 常见编程技巧

PS：除了内存池和常量数组，还搬了一些其他的小技巧。

## 内存池

当我们需要动态分配内存的时候，频繁使用 **new/malloc** 会占用大量的时间和空间，甚至生成大量的内存碎片从而降低程序的性能，可能会使原本正确的程序 **TLE/MLE**。

这时候我们就需要使用到「内存池」这种技巧：在真正使用内存之前，先申请分配一定大小的内存作为备用，当需要动态分配时则直接从备用内存中分配一块即可。

当然在大多数 **OI** 题当中，我们可以预先算出需要使用到的最大内存并一次性申请分配。

如申请动态分配**32**位有符号整数数组的代码：

```
inline int* newarr(int sz) {
  static int pool[maxn], *allocp = pool;
  return allocp += sz, allocp - sz;
}
```

线段树动态开点的代码：

```
inline Node* newnode() {
  static Node pool[maxn << 1], *allocp = pool - 1;
  return ++allocp;
}
```

## 常量数组

善用常量数组往往能简化代码。定义常量数组时无须指明大小，编译器会计算。 下面是两道例题

$WERTYU$：$UVa10082$

```
#include<stdio.h>
char s[] = "`1234567890-=QWERTYUIOP[]\\ASDFGHJKL;'ZXCVBNM,./";
int main(){
    int i, c;
    while((c = getchar()) != EOF){
        for (i=1; s[i] && s[i]!=c;i++);
        if(s[i]) putchar(s[i-1]);
        else putchar(c);
    }
    return 0;
}
```

回文词：$UVa401$：

输入一个字符串，判断它是否为回文串以及镜像串。输入字符串保证不含数字0。（使用常量数组）

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
const char* rev = "A   3  HIL JM O   2TUVWXY51SE Z  8 ";
const char* msg[] = {"not a palindrome", "a regular palindrome", "a mirrored
string", "a mirrored palindrome"};

char r(char ch) {
  if(isalpha(ch)) return rev[ch - 'A'];
  return rev[ch - '0' + 25];
}

int main() {
  char s[30];
  while(scanf("%s", s) == 1) {
    int len = strlen(s);
    int p = 1, m = 1;
    for(int i = 0; i < (len+1)/2; i++) {
      if(s[i] != s[len-1-i]) p = 0;
      if(r(s[i]) != s[len-1-i]) m = 0;
    }
    printf("%s -- is %s.\n\n", s, msg[m*2+p]);
  }
  return 0;
}
```

# 对拍

有的时候我们写了一份代码，但是不知道它是不是正确的。这时候就可以用对拍的方法来进行检验或调试。

什么是对拍呢？具体而言，就是通过对比两个程序的输出来检验程序的正确性。你可以将自己程序的输出与其他程序（打的暴力或者其他 dalao 的标程）的输出进行对比，从而判断自己的程序是否正确。

当然，对拍过程要多次进行，我们需要通过批处理的方法来实现对拍的自动化。

具体而言，我们需要一个数据生成器，两个要进行对拍的程序。

每次运行一次 数据生成器，将生成的数据写入输入文件，通过重定向的方法使两个程序读入数据，并将输出写入指定文件，利用 **Windows** 下的 **fc** 命令比对文件（**Linux** 下为 **diff** 命令），从而检验程序的正确性。

如果发现程序出错，可以直接利用刚刚生成的数据进行调试啦。

对拍程序的大致框架如下：

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  // For Windows
```

```
  //对拍时不开文件输入输出
  //当然，这段程序也可以改写成批处理的形式
  while (1) {
    system("gen > test.in");  //数据生成器将生成数据写入输入文件
    system("test1.exe < test.in > a.out");  //获取程序1输出
    system("test2.exe < test.in > b.out");  //获取程序2输出
    if (system("fc a.out b.out")) {
      //该行语句比对输入输出
      // fc返回0时表示输出一致，否则表示有不同处
      system("pause");  //方便查看不同处
      return 0;
      //该输入数据已经存放在test.in文件中，可以直接利用进行调试
    }
  }
}
```

# 善用标识符进行调试

我们在本地测试的时候，往往要加入一些调试语句。要提交到 OJ 的时候，就要把他们全部删除，有些麻烦。

我们可以通过定义标识符的方式来进行本地调试。

大致的程序框架是这样的：

```
#define DEBUG
#ifdef DEBUG
// do something
#endif
// or
#ifndef DEBUG
// do something
#endif
```

**#ifdef** 会检查程序中是否有通过 **#define** 定义的对应标识符，如果有定义，就会执行下面的内容，**#ifndef** 恰恰相反，会在没有定义相应标识符的情况下执行后面的语句。

我们提交程序的时候，只需要将 **#define DEBUG** 一行注释掉即可。

当然，我们也可以不在程序中定义标识符，而是通过 **-DDEBUG** 的编译选项在编译的时候定义 **DEBUG** 标识符。这样就可以在提交的时候不用修改程序了。

不少 **OJ** 都开启了 **-DONLINE_JUDGE** 这一编译选项，善用这一特性可以节约不少时间。

# 循环宏定义

我们写代码时，像下面这样的循环代码写得会非常多：

```
for (int i = 0; i < N; i++) {
```

```
}
```

为了简化这样的循环代码，我们可以使用宏定义：

```
#define f(x, y, z) for (int x = (y), __ = (z); x < __; ++x)
```

这样写循环代码时，就可以简化成 **f(i, 0, N)** 。例如：

```
// a is a STL container
f(i, 0, a.size()) { ... }
```

另外推荐一个比较有用的宏定义：

```
#define _rep(i, a, b) for (int i = (a); i <= (b); ++i)
```

# 重载运算符

重载运算符是通过对运算符的重新定义，使得其支持特定数据类型的运算操作。

有的时候，我们构造了一种新的数据类型（高精度数，矩阵），当然可以将数据类型的运算操作用函数的形式写出来。但采用重载运算符的方式，可以让程序更为自然。

当然，重载运算符在语法上也有一些要求：

1. 重载运算符的一般形式为返回值类型 **operator** 运算符(参数,…)

2.在结构体内部重载运算符时，括号内的参数会少一个（事实上，另外一个参数是this指针，即指向当前参数的指针，也就是说，两元运算符只需要1个参数。（在结构体外部定义时，两元运算符还是需要2个参数）

其他要求就和普通函数的要求差不多了。

举一个简单的例子：

```
#include <stdio.h>
struct pair_num//一个二元组类型
{
 int x,y;
 pair_num operator +(pair_num a)const //不加const会CE
 {
  pair_num res;
  res.x=x+a.x;//x事实上是this.x
  res.y=y+a.y;
  return res;
 }
 pair_num operator -(pair_num a)const
 {
  pair_num res;
  res.x=x-a.x;
  res.y=y-a.y;
  return res;
```

```
 }
 bool operator <(pair_num a)const //sort,set等数据结构需要使用小于号
 {
  return x<a.x||(x==a.x&&y<a.y);
 }
}a,b,res;
pair_num operator *(pair_num a,pair_num b)//在结构体外部定义时，不要加const
{
 pair_num res;
 res.x=a.x*b.x;
 res.y=a.y*b.y;
 return res;
}
int main()
{
 scanf("%d%d",&a.x,&a.y);
 scanf("%d%d",&b.x,&b.y);
 res=a+b;
 printf("%d %d\n",res.x,res.y);
 res=a-b;
 printf("%d %d\n",res.x,res.y);
 res=a*b;
 printf("%d %d\n",res.x,res.y);
 return 0;
}
```