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。