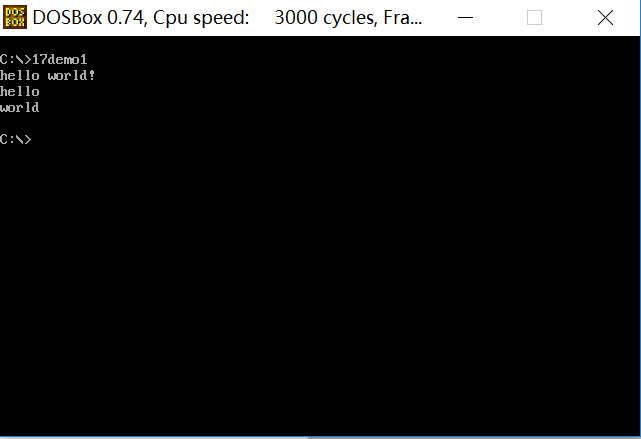
**c语言综合研究十七报告**

1. **研究过程展示**

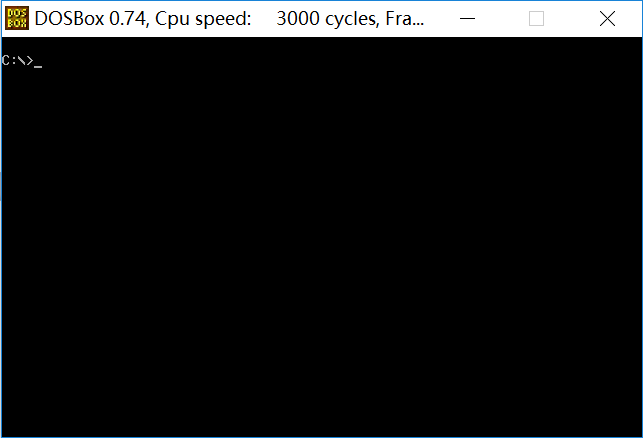
**1>.程序17demo1.c部分：**



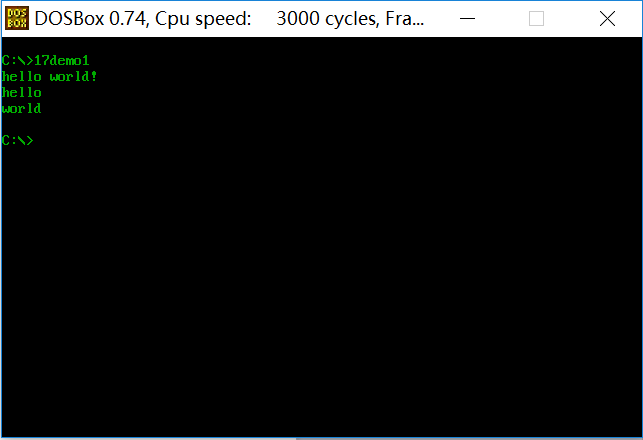
N & n:



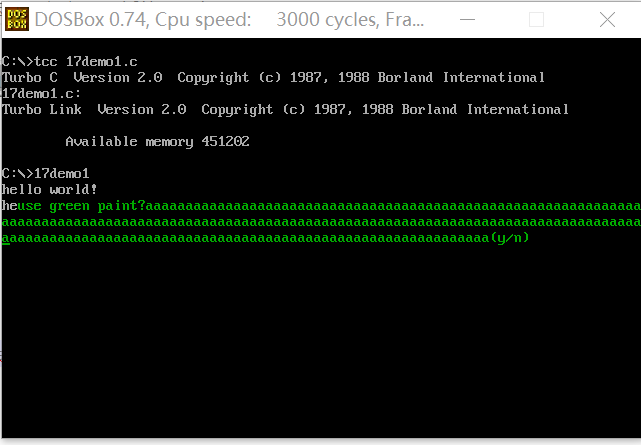
Y &y:



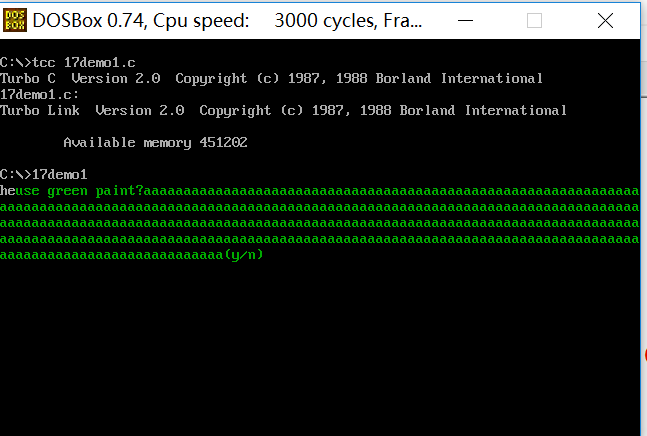
Y & n:



给个长度大于一行的字符串：



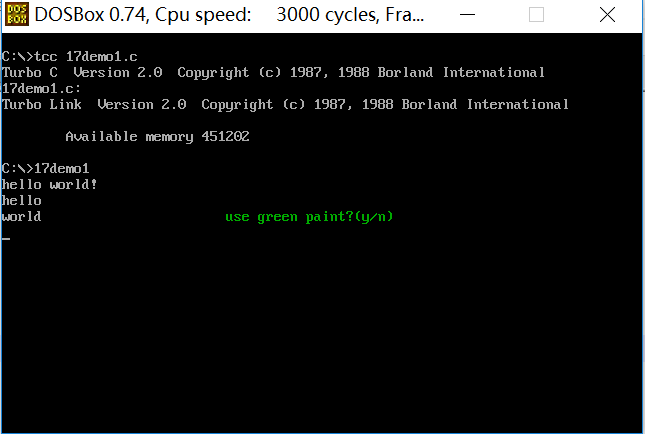
再长一点：



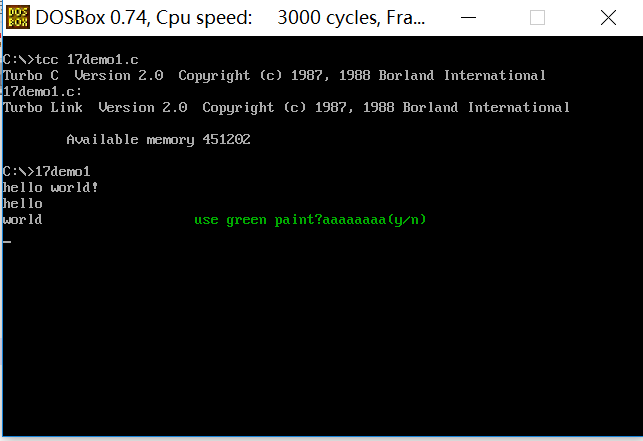
可以看到，字符串开头处相比前一个又提升了一行，但是都是从第3列开始，而且整体表现的效果就是“相对于屏幕上下方向居中”：前一个总共占3行，第二行，即中间一行占据屏幕正中间一行；第二个总共占5行，第3行，即中间一行还是占据屏幕正中间一行。

但是之前小于一行长度的字符串，我们看到字符串开头并不是第3行，修改一下，加个另外一个长度小于屏幕一行的字符串和原版对比：

原版：



对比版：



比较可以得出，字符串开头显示的列的位置变了，但是行的位置没有变。但是两个字符串相对于屏幕左右方向而言是居中的。又因为本来就是只是一行，而且也是设定好了中间一行，所以实际效果其实是相对于屏幕上下方向居中并且左右方向居中的。

因为我们要在屏幕中央显示字符串，这样会覆盖掉屏幕上原来的信息，就像前面输入过长的字符串参数的时候，显示的同时把原有的“hello world”等覆盖掉了一样。要想程序结束之后恢复他们，所以我们要提前将要显示的区域的原有内容保存起来，在我们的字符串显示之后，再将原有的屏幕内容恢复。那么这些信息保存在哪里呢？保存在另一段内存中，因为传入的字符串长度是不确定的，所以要保存的屏幕信息的大小也是不确定的，我们无法先开辟一个固定的空间来保存它，只能动态地开辟空间来保存。所以需要使用malloc函数来动态地开辟空间，并将屏幕上的信息保存到这段内存中。这里传入的字符串长度为n，但是屏幕上一个字符是用2个字节来显示的，一个存储字符内容，一个存储颜色属性，所以我们在转移时转移的字节数应该是字符串长度的两倍。

**2>.程序17demo2.c部分：**

程序：

void f1();

void f2();

void ask(char\* chp,void (\*fp)());

main()

{

printf("hello world!\n");

printf("hello\n");

printf("world\n");

ask("use green paint?(y/n)",f1);

ask("clear scr?(y/n)",f2);

}

void f1()

{

int a;

char far\* p = (char far\*)0xb8000000;

for(a = 0 ; a < 2000; a++)

{

p[a\*2 + 1] = 2;

}

}

void f2()

{

system("cls");

}

void ask(char\* str,void (\*fp)())

{

int n;

int a;

int row;

int col;

char far\* scr = (char far\*)0xb8000000;

char\* buf;

char ch;

n = strlen(str);

if(n/80)

{

row = (25 - (n/80 + 1))/2;

col = 3;

}

else

{

row = 12;

col = (80 - (n%80))/2;

}

buf = (char\*)malloc(n\*2);

for(a = 0;a < n\*2;a++)

{

buf[a] = scr[ (row - 1)\*160 +(col - 1)\*2 + a ];

}

for(a = 0;a <n;a++)

{

scr[ (row - 1)\*160 + (col - 1 + a)\*2 ] = str[a];

scr[ (row - 1)\*160 + (col - 1 + a)\*2 + 1 ] = 2;

}

while(8)

{

ch = getch();

if( ch == 'y' || ch == 'Y' || ch == 'n' || ch == 'N' )

break;

}

for(a = 0; a < n\*2; a++)

{

scr[ (row - 1)\*160 + (col - 1)\*2 + a ] = buf[a];

}

if( ch == 'y' || ch == 'Y' )

{

fp();

}

}

如果用户输入的是’y’，’Y’，则调用子函数，之后返回，如果用户输入的是’n’，’N’，则直接返回。

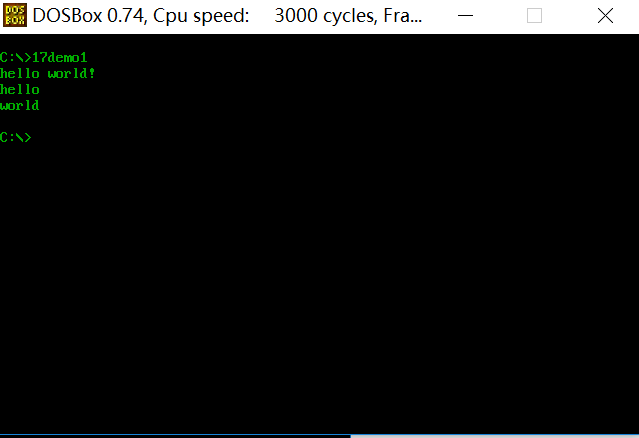
我们将函数ask的代码加入到cs .lib中，这样我们要实现完整程序，只需要在c文件里写好子函数，再在main函数里调用就行了。

**3>.分析部分**

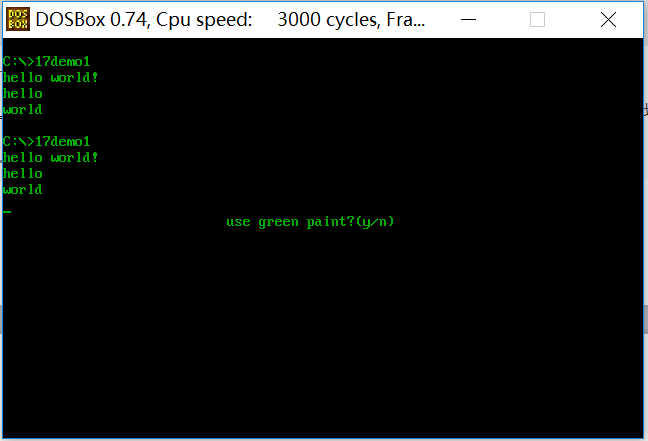
但是以上两种程序都有一个地方不符合用户需求，我在连续调用程序的时候，比如第一次调用，选择“y & n”选项，也就是绿屏并且不清空；然后紧接着第二回调用，选择“n & n”选项，也就是不绿屏并且不清空。此处用户已经选择了不绿屏，但是显示效果呢？是这样的：

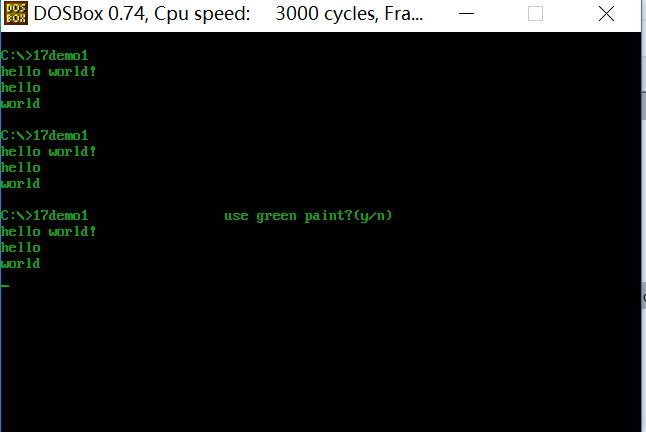
第一次调用，“y & n”:





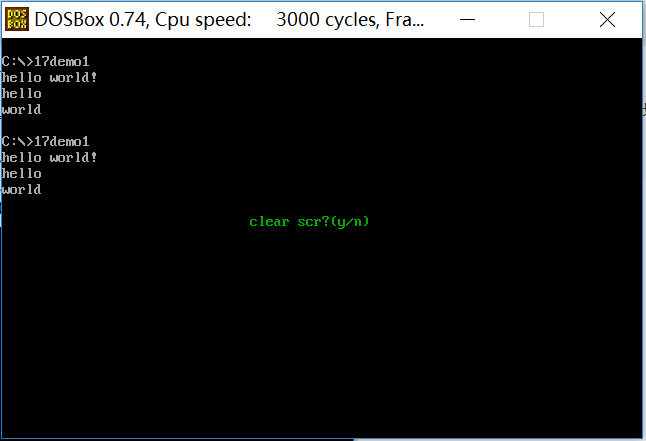
紧接着第二次调用，“n & n”:

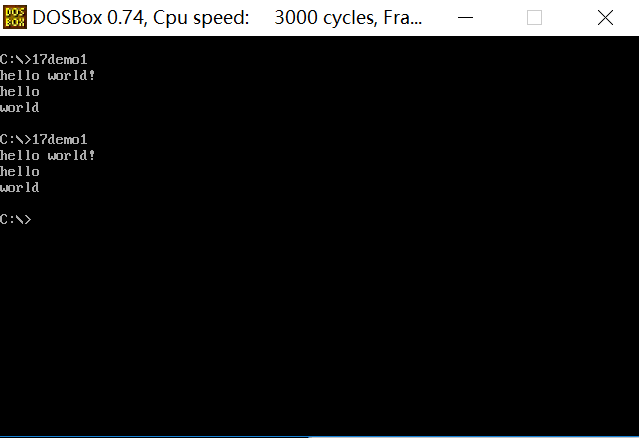




可以看到第二次调用的不绿屏选项明显受到了第一次调用的绿屏选项的干扰而无法实现。

针对这个现象，应该添加对其处理的语句，修改后的效果：





程序部分：

int fa(char\* str )

{

int n;

int a;

int row;

int col;

char far\* scr = (char far\*)0xb8000000;

char\* buf;

char ch;

n = strlen(str);

if(n/80)

{

row = (25 - (n/80 + 1))/2;

col = 3;

}

else

{

row = 12;

col = (80 - (n%80))/2;

}

buf = (char\*)malloc(n\*2);

for(a = 0;a < n\*2;a++)

{

buf[a] = scr[ (row - 1)\*160 +(col - 1)\*2 + a ];

}

for(a = 0;a <n;a++)

{

scr[ (row - 1)\*160 + (col - 1 + a)\*2 ] = str[a];

scr[ (row - 1)\*160 + (col - 1 + a)\*2 + 1 ] = 2;

}

while(8)

{

ch = getch();

if( ch == 'y' || ch == 'Y' || ch == 'n' || ch == 'N' )

break;

}

for(a = 0; a < n\*2; a++)

{

scr[ (row - 1)\*160 + (col - 1)\*2 + a ] = buf[a];

}

return (ch == 'y'||ch == 'Y');

}

main()

{

int a;

char far\* p = (char far\*)0xb8000000;

printf("hello world!\n");

printf("hello\n");

printf("world\n");

a = fa("use green paint?(y/n)");

if(a)

{

for(a = 0; a < 2000; a++)

{

p[a\*2 + 1] = 2;

}

}

else

{

for(a = 0; a < 2000; a++)

{

p[a\*2 + 1] = 7;

}

}

a = fa("clear scr?(y/n)");

if(a)

{

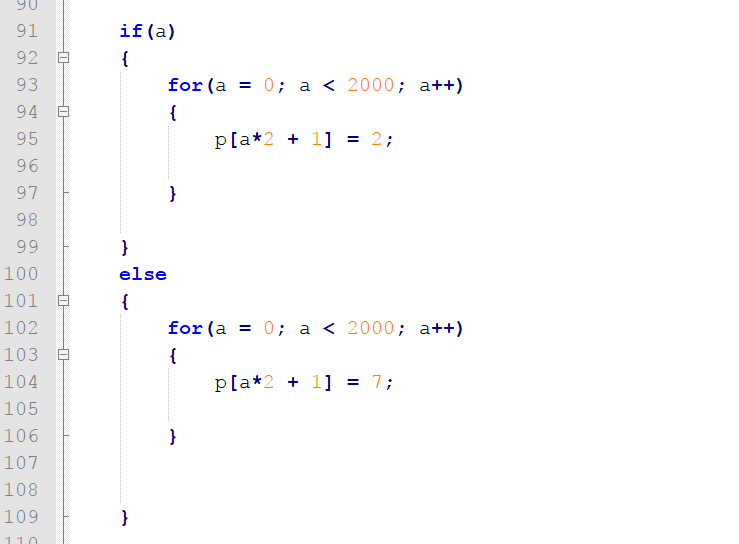
system("cls");

}

}

也就是在第一部分的一个地方修改了一下：

Main函数中：



同理，第二部分的程序在也可以经过修改弥补这一点。

fa函数是用一个函数实现所有的功能，但是这样一个函数里面的代码太多了，会导致程序的维护性和可读性降低，所以ask在fa的基础上用子函数实现特定功能，这样修改和读起来都更加方便快捷。所以ask函数要比fa函数更好，因为它可以增加子函数对程序的功能进行扩展，而且函数的可读性和维护性更好。ask函数的设计思想是将函数的功能尽可能地细化，尽量做到每一个函数只实现一个功能。

ask函数的设计，很容易让人想到“分治法”，将一个问题分而治之，可以有效地处理大规模问题。同时，可以看到，main函数中只剩下printf()函数和ask()函数的调用，函数之间的分层比较明显。

1. **已思考研究并已解决问题汇总**
2. **已思考研究并未解决问题汇总**
3. **研究感想（心得体会）**

刚看了几个TED关于科技方面的演讲，最近又开了好几个新坑，学的东西又要多了，有点混乱，但是就像本次研究的一部分核心思想一样，有序对于程序的重要性看来比我想的还要重要，对啊，多浅显的道理，二分法就让效率提高了多少。不过感觉自己对这里的敏感度太不够了。以后尽量试试一个功能，一个函数这样的编写。