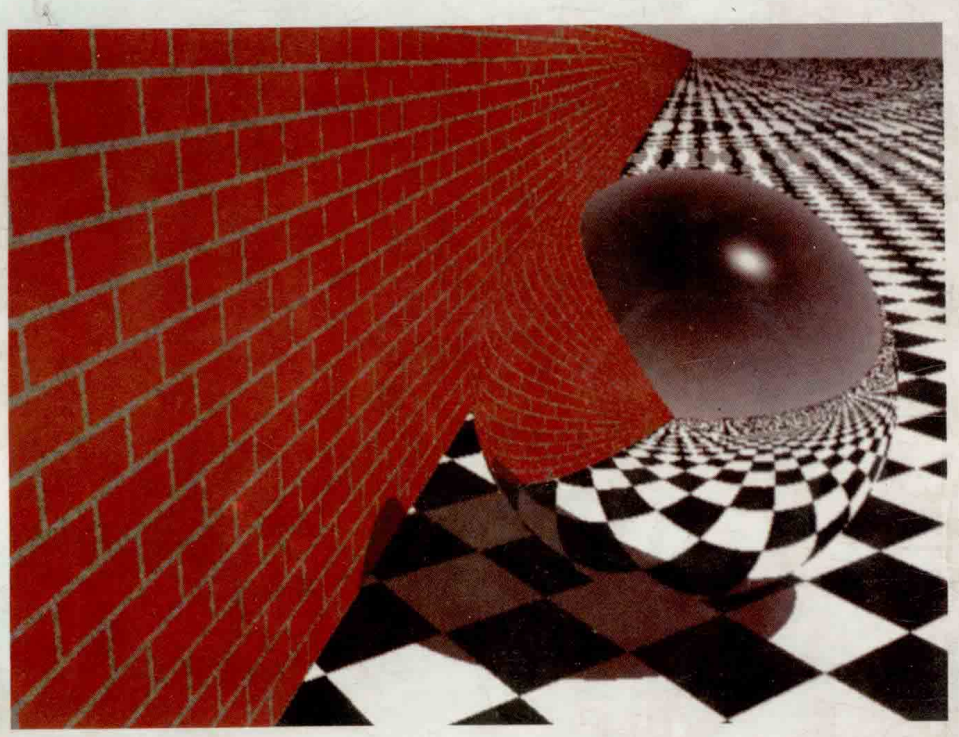
计算机C/C++语言系列丛书

希望

学苑出版社

Turbo C
Borland C++



李伟生 李春葆 编著

Turbo C/Borland C++
程序设计94例

Turbo C/Borland C++

程序设计94例

李伟生
李春葆 编著

熊可宜 审校

学苑出版社

(京)新登字 151 号

内 容 简 介

本书为在编制规模较大的软件时处理内存管理、外设使用、界面设计、中断服务、流操作这些问题提供了示例代码。本书含有大量例子，全部用 Turbo C 或 Borland C++写成，通用性强，便于移植。

本书是高级程序员的良师益友。

欲购本书的用户，请直接与北京海淀 8721 信箱书刊部联系，邮政编码：100080，电话：2562329。

计算机 C/C++语言系列丛书

Turbo C/Borland C++程序设计 94 例

编　　著：李伟生　李春葆
审　　校：熊可宜
责任编辑：甄国宪
出版发行：学苑出版社　　**邮政编码**：100036
社　　址：北京市海淀区万寿路西街 11 号
印　　刷：施园印刷厂
开　　本：787×1092　1/16
印　　张：23.375　　**字　　数**：554 千字
印　　数：1～3000 册
版　　次：1994 年 8 月北京第 1 版第 1 次
ISBN7－5077－0875－6/TP・24
本册定价：25.00 元

前　言

总的来说，C 语言是一种通用性很强、便于移植的计算机编程语言。但是，任何一类计算机工作平台都有它独有的硬件特性，因而在其上建立的计算机语言也有着其独特性。所以，仅仅学过一般 C 语言的程序员是无法在一个具体的环境中开发系统软件的。在编制规模较大的软件时，内存管理、外设使用、界面设计、中断服务和流操作等往往是令人头痛的事。所以如何用 C 语言来处理这一类问题自然就成了编程热点。本书是为处理这些问题提供了示例代码，这些代码揭示了许多秘密。

本书中所有例子都是用 Turbo C 或者 Borland C++ 写成的。程序大致涉及以下几个方面：内存管理、图形设备、鼠标器使用、DOS 使用、中断处理、外设检测、C++ 编程、Windows 与 Turbo Vision 开发等。

本书的每一节都是独立的。每一节都介绍一种特殊的技巧。相信这本书会成为高级程序员的良师益友。

编　者

目　录

1 确定运行栈空间的尺寸(Ⅰ)

在近模式,栈驻留在数据段。它从DGROUP的结尾处开始且向内存中SS及DS处增长。变量brklvl总是标准堆的顶。如果栈检查开关打开,则栈可以增长到SP抵达_brklvl,此时产生栈溢出。如果栈检查开关未开,则栈增长到SP抵达_brklvl后将覆盖堆。

在远模式,栈开始于内存高部刚好在_heapbase前,且增长时向下可到SS。

下面的例子指出如何检查运行时栈的大小。

```
#include <stdio.h>
#include <conio.h>

unsigned long stack();

void main()
{

   unsigned long size = stack();
   printf("\nStack size = %lu",size);
}

unsigned long stack()
{
    unsigned long ret;
    extern unsigned long heapbase;      //Beginning of the far heap
    extern  unsigned brklvl;            //End of the near heap
    clrscr();
#if defined (__COMPACT__) || defined (__LARGE__) || defined
(__HUGE__)
    ret = 16 * (*((unsigned *)_heapbase + 1) - SS) - 16;

#else
    ret =   *((unsigned *) heapbase + 1 ) -   __brklvl - 1;
#endif
   return ret;
}
```

2 确定运行栈空间的尺寸(Ⅱ)

本程序提供了如何确定一个应用未使用的栈空间尺寸的例子,同时还可以确定已经使用的近堆的尺寸。

它是通过在程序的开头调用一个函数实现的,该函数用字符 0XFF 标记所有当前未使用的栈空间。在程序退出之前调用另一个函数统计栈中未改变的 0XFF 字符的数目。当我们试图确定使用 Calloc 函数而不是 malloc 函数分配的近堆空间时,由于 Calloc 把分配的内存都初始化为 0,因此即使其值未发生改变,这片内存也可能分配了。

这种方法虽然不是在任何情况下都正确,但在绝大多数情况下是可行的,其可靠性可以通过本程序的执行过程和特定情况分析得到,例如,必须改变用于标记未用栈空间的字符,便是一种特定情况。还要注意到,当我们试图改变一个内存模式中的栈尺寸时,必须改变_heaplen 或 DGROUP 的值。

为了使用这些函数,我们需要把如下函数原型放在要调用的源文件中:

```
void init_stack_count (void);
void stack_count (void);
```

为了确定未用的栈空间尺寸,必须首先调用 init_stack_count()函数,把它放在程序的开头,在退出本程序之前还要调用 stack_count()函数,在屏幕上打印未用的栈空间。如果我们要确定使用的近堆空间尺寸以便调整 near 内存模式中的栈,就应在该模块中定义符号_HEAP_,使用的语句如下:

```
#define HEAP_
```

并把它放在本程序的文件范围(任何函数定义的外层)中,这个源文件必须像程序中的其它模块一样被编译和连接到程序中。

本程序中包含的函数及其功能如下:

init_stack_count():把当前未使用的所有可能的栈空间设置为 0XFF 值。

stack_count():统计所有未使用或未修改的值的栈空间,结果为栈上未使用的空间。

本程序清单如下:

```
PRODUCT  :  Borland C++                    NUMBER  :  1036
VERSION  :  3.1
   OS    :  DOS
  DATE   :  October 23, 1992               PAGE    :  1/5

  TITLE  :  Determining the amount of stack used at runtime.

        The following program provides an example on how to determine
    how the amount of stack not being used by an application.  The exam-
    ple also determine the amount of the near heap that has beenused.
```

This is accomplished by making a call at the beginning of the program to a function that will mark all of the stack space not currently in use with the character 0xFF. Just before the program exits, a call is made to another function that will count the number of unchanged 0xFF bytes on the stack. When trying to determine the amount of the near heap that was used calloc should be used rather than malloc to allocate the memory because calloc will initialize the memory allocated to all zeros so it will later be apparent that memory was allocated even if the values were never changed.

This method is not a 100% guaranteed one, but it will work most of the time. Its reliablity can be determined by understanding how it performs its task and what problems that may incur with a particular program. For instance it may prove necessary to change the character used to mark the unused stack space. It is also important to note that when attempting to change the stack size in a near memory model one must change the value of _heaplen or DGROUP will always be 64K regardless of the value of _stklen.

In order to use these function you need to place the following prototypes in the source file you will be calling them from:

```
void init_stack_count(void);
void stack_count(void);
```

To determine the amount of unused stack you must first make an initial call to:

```
init_stack_count();
```

at the beginning of the program and just before exiting the program another call is required to:

```
stack_count();
```

The amount of unused stack space will be printed on the screen.

If you need to determine the amount of the near heap used in order to adjust the stack in the near memory model then the symbol _HEAP_ must be defined in this module. You can do this by adding the statement:

```
#define _HEAP_
```

at file scope (outside any function definition) in this source file. Then this souce file must be compiled and linked into your program as if it were any other module in your program.

```
#include <stdio.h>
#include <process.h>
#include <conio.h>
#include <dos.h>
#include <alloc.h>

#define SIZE_OF_EMULATOR 415          // Account for floating
                                      // point emulator
#define FILL_CHAR 0xFF                // Character used to mark
                                      // the stack

#pragma warn -aus                     // Turn off unecessary
#pragma warn -use                     // warnings


/*************************************************************\
|    Marks all possible stack space not currently in use by   |
|    setting the values    to 0xff.                           |
\*************************************************************/
void init_stack_count(void)
{
     char far *sp;                    // far memory model stack
                                      // pointer
     char near *nsp;                  // near memory model stack
                                      // pointer
     extern   unsigned brklvl;        // Internal variable marking
                                      // top of
                                      // the near heap.

#if  defined(__COMPACT__)     ||  \
     defined(__LARGE__)       ||  \
     defined(__HUGE__)
#undef    __HEAP__

     sp = MK_FP( _SS, _SP - 1 );      // initialize sp to point to
                                      // the next available space on
                                      // the stack.

     while( sp > (char far *)SIZE_OF_EMULATOR ) //Check
```

```
                                        // for stack overflow
    {
        *sp = FILL_CHAR;                // Initialize unused stack
                                        // space
        sp--;
    }
#else                                   // similar for near data
                                        // memory model
    nsp = (char *)_SP - 1;              // Initialize pointer
    while( nsp > (char *)__brklvl)      // Check for stack overflow
    {
        *nsp = FILL_CHAR;               // Assign fill character
        nsp--;                          // advance pointer
    }
#endif
    return;
}
```

```
/*************************************************************\
|   Goes though stack space counting unused (unmodified)      |
|   values meaning that space on the stack was not used.      |
\*************************************************************/
void stack_count( void )
{
    unsigned count= 0;              // Amount of unused stack
                                    // space
    unsigned mcount = 0;            // Amount of used Heap space
    char far *sp;                   // Far memory model pointer
    char *nsp;                      // Near memory model pointer


#if  defined(__COMPACT__)    ||  \
     defined(__LARGE__)      ||  \
     defined(__HUGE__)

    sp = MK_FP( _SS, SIZE_OF_EMULATOR+1);   // Initialize the
                                            // pointer
    if( _SP >  SIZE_OF_EMULATOR )           // Check for stack
                                            // overflow
    {
```

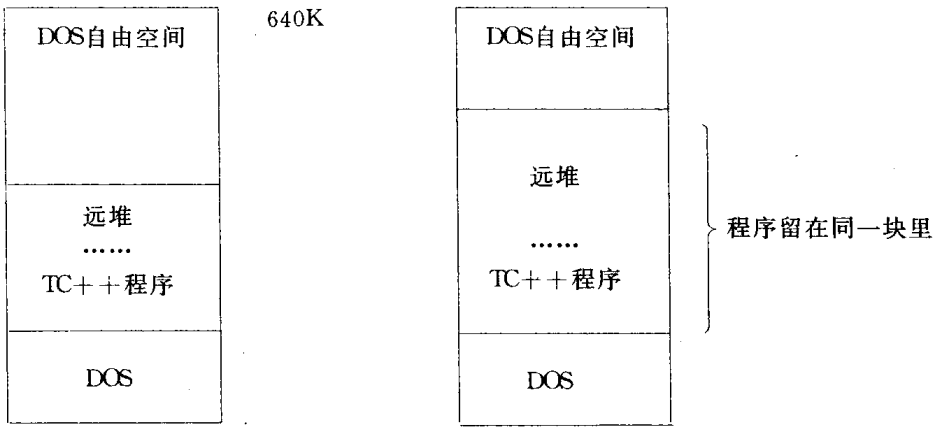
```
            while( sp < MK_FP(_SS,_SP)  )          // Check for stack
                                                   // overflow
            {
                if( *sp != (char) FILL_CHAR )      // compare pointer
                                                   //value to
                                                   // fill character
                    break;
                count++;                           // count unused
                                                   // stack space
                sp++;                              // increment pointer
            }
        }
#else
        extern unsigned brklvl;          // End of near heap
        extern unsigned heapbase;        // Beginning of near heap
        if( _SP > brklvl  )              // Check for stack
                                         // overflow
        {
            nsp = (char *) brklvl + 1;
            while( nsp < (char *)_SP )   // While we don't run
                                         // into the
                                         // current stack.
            {
                if( *nsp != (char) FILL_CHAR && count < 3)
                                         // Count near heap used
                    mcount++;
                                         // else count unused
                                         // stack
                else if( *(nsp + 1) == (char)FILL_CHAR &&
                        *(nsp + 2) == (char) FILL_CHAR  )
                    count++;             // Count unused bytes
                nsp++;
            }
        }
        count += 2;
#endif


#ifdef   _HEAP_                          // print amount of used
                                         // stack
        printf("\nHeap Used = %u bytes", (_brklvl - heapbase +
                                         mcount));
#endif                                   // print amount of unused
```

```
                                        // stack
    printf("\nStack not used = %u bytes", count);
    return;
}
```

3 allocmem 与 farmalloc

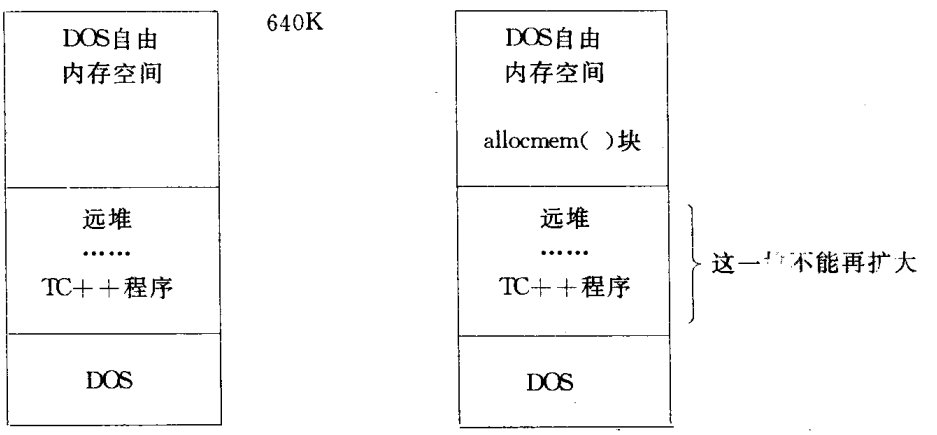
一般参考指出，farmalloc 与 allocmem 不能协调工作。这并不完全正确，这样的评述可使对内存分配子程序缺乏了解的用户避免切肤之痛。

当 Turbo C++程序工作时，它总驻在一个简单的 DOS 内存块里。如果需要从远堆里分配更多的内存，则内存块要扩充到适合要求的大小。



用farmalloc( )之前　　用farmalloc( )之后

如果调用的是 allocmem()，则将分配一个分离的块。通常这一个块将是紧接程序的上面一块。结果，如果你希望在这一点申请更多的远堆内存，那么你的程序所在块就不能扩大空间。所以，如果你在此时使用 farmalloc，就得到 NULL 返回值。



用allocmem( )之前　　用allocmem( )之后

所有运行 DOS 分配的操作对 TC++程序都有同样的结果，这包括驻留 TSR 程序。

可见，从远堆申请内存的程序才有可能引起致命的错误。使用 malloc 于近堆不受影响。另外，如果用一个从上到下分配内存的方式，farmalloc 仍可工作。

DOS 软中断 21h 函数 58h 正是做这些工作的。

```
AX=0x5801;
BX=2;
geninterrupt(0x21);
```

运行后,DOS 将从上到下地分配内存。下面语句将 DOS 返回到取第一个相邻的部分。

```
AX=0x5801;
BX=0;
geninterrupt(0x21);
```

下面是两个描述问题及解决问题的程序。

程序 1 显示出内存分配冲突。

```
#include <alloc.h>
#include <dos.h>
#include <stdio.h>

main()
{
    unsigned seg, count = 0;
    char far *ptr;

    allocmem( 100, &seg );   /* allocates next available block */
    printf(" Allocated block at %X:0000\n", seg);
    while ((ptr = farmalloc(50)) != NULL)
        count++;
    printf(" Allocated %u 50 byte pieces\n", count); /* can't allocate! */

    return 0;
}
```

程序 2 给出解决办法。

```
#include <alloc.h>
#include <dos.h>
#include <stdio.h>

main()
{
    unsigned seg, count = 0;
    char far *ptr;

    _AX = 0x5801;
    _BX = 2;
    geninterrupt(0x21);      /* set allocation strategy to last fit */
```

```
    allocmem( 100, &seg ); /* grab highest block */
    _AX = 0x5801;
    _BX = 0;


    geninterrupt(0x21);      /* reset allocation strategy */
    printf(" Allocated block at %X:0000\n", seg);
    while ((ptr = farmalloc(50)) != NULL)
        count++;
    printf(" Allocated %u 50 byte pieces\n", count);

    return 0;
}
```