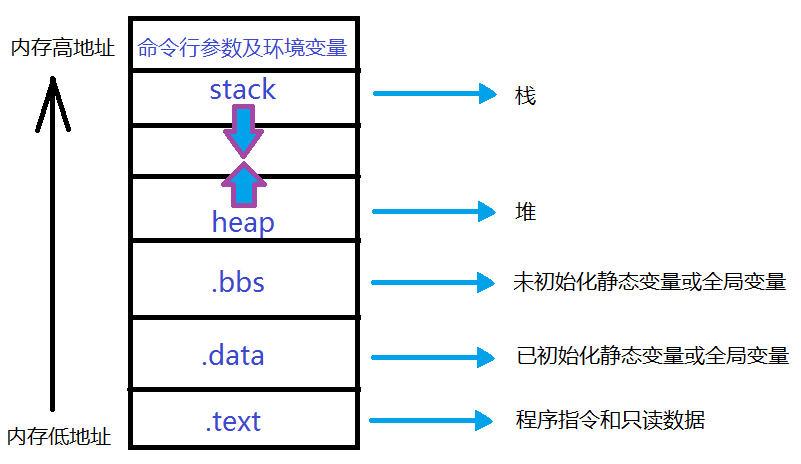
# 内存管理

# 基本知识

## 一、内存管理

### 1.1 程序在内存里的分配



代码段(.text)，也称文本段(Text Segment)，存放着程序的机器码和只读数据，可执行指令就是从这里取得的。如果可能，系统会安排好相同程序的多个运行实体共享这些实例代码。这个段在内存中一般被标记为只读，任何对该区的写操作都会导致段错误（Segmentation Fault）。

数据段，包括已初始化的数据段(.data)和未初始化的数据段（.bss），前者用来存放保存全局的和静态的已初始化变量，后者用来保存全局的和静态的未初始化变量。数据段在编译时分配。

### 1.2堆和栈

* 堆（Heap）：用来存储程序运行时分配的变量。

  堆的大小并不固定，可动态扩张或缩减。其分配由malloc()、new()等这类实时内存分配函数来实现。当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用free等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减） 堆的内存释放由应用程序去控制，通常一个new()就要对应一个delete()，如果程序员没有释放掉，那么在程序结束后操作系统会自动回收。

栈（Stack）是一种用来存储函数调用时的临时信息的结构，如函数调用所传递的参数、函数的返回地址、函数的局部变量等。 在程序运行时由编译器在需要的时候分配，在不需要的时候自动清除。

栈的特性: 最后一个放入栈中的物体总是被最先拿出来，这个特性通常称为先进后出(FILO)队列。

栈的基本操作： PUSH操作：向栈中添加数据，称为压栈，数据将放置在栈顶； POP操作：POP操作相反，在栈顶部移去一个元素，并将栈的大小减一，称为弹栈。

堆和栈的区别：

1)分配和管理方式不同 ：

       堆是动态分配的，其空间的分配和释放都由程序员控制。

       栈由编译器自动管理。栈有两种分配方式：静态分配和动态分配。

                    静态分配由编译器完成，比如局部变量的分配。

                    动态分配由alloca()函数进行分配，但是栈的动态分配和堆是不同的，它的动态分配是由编译器进行释放，无须手工控制。

2)产生碎片不同

      对堆来说，频繁的new/delete或者malloc/free势必会造成内存空间的不连续，造成大量的碎片，使程序效率降低。

      对栈而言，则不存在碎片问题，因为栈是先进后出的队列，永远不可能有一个内存块从栈中间弹出。

3)生长方向不同

      堆是向着内存地址增加的方向增长的，从内存的低地址向高地址方向增长。

      栈的生长方向与之相反，是向着内存地址减小的方向增长，由内存的高地址向低地址方向增长。

## 二、内存分配方式

### 2.1内存分配方式有三种：

（1）从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static 变量。

以下是采用静态分配方式的例子。

int a=100;

此行代码指示编译器分配足够的存储区以存放一个整型值，该存储区与名字a相关联，并用数值100初始化该存储区。

（2）在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

（3） 从堆上分配，亦称动态内存分配。程序在运行的时候用malloc 或new 申请任意多少的内存，程序员自己负责在何时用free 或delete 释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

以下是采用动态分配方式的例子。

P1=(char \*)malloc(10\*sizeof(int));//分配10\*4字节在堆区

此行代码分配了10个int类型的对象，然后返回对象在内存中的地址，接着这个地址被用来初始化指针对象p1，对于动态分配的内存唯一的访问方式是通过指针间接地访问，其释放方法为：

free(p1);

### 2.2内存管理函数

#### 2.2.1malloc/free

malloc()函数用来在堆中申请内存空间，free()函数释放原先申请的内存空间。malloc()函数是在内存的动态存储区中分配一个长度为size字节的连续空间。其参数是一个无符号整型数，返回一个指向所分配的连续存储域的起始地址的指针。当函数未能成功分配存储空间时（如内存不足）则返回一个NULL指针。

由于内存区域总是有限的，不能无限制地分配下去，而且程序应尽量节省资源，所以当分配的内存区域不用时，则要释放它，以便其他的变量或程序使用。

函数定义如下：

void \*malloc(size\_t size)//返回类型为空指针类型

void free(voide \*ptr)

例如：

Int \*p1,\*p2;

p1=(int \*)malloc(10\*sizeof(int));

p2=p1;

….

Free(p2); //或者free(p1)

p1=NULL; //或者p2=NULL

malloc()函数返回值赋给p1，又把p1的值赋给p2，所以此时p1，p2都可作为free函数的参数。使用free()函数时，需要特别注意下面几点：

（1）调用free()释放内存后，不能再去访问被释放的内存空间。内存被释放后，很有可能该指针仍然指向该内存单元，但这块内存已经不再属于原来的应用程序，此时的指针为悬挂指针（可以赋值为NULL）。

（2）不能两次释放相同的指针。因为释放内存空间后，该空间就交给了内存分配子程序，再次释放内存空间会导致错误。也不能用free来释放非malloc()、calloc()和realloc()函数创建的指针空间，在编程时，也不要将指针进行自加操作，使其指向动态分配的内存空间中间的某个位置，然后直接释放，这样也有可能引起错误。

（3）在进行C语言程序开发中，malloc/free是配套使用的，即不需要的内存空间都需要释放回收。

下面是使用这两个函数的一个例子。

#include  <stdio.h>            //printf()    //(1)头文件信息

#include   <malloc.h>        //malloc()    //(2)

int main(int argc,char\* argv[],char\* envp[])   //(3)

{

int count;

int\* array;

if((array=(int \*)malloc(10\*sizeof(int)))==NULL)  //(4)分配空间

{

printf("malloc memory unsuccessful");

exit(1);

}

for (count=0;count<10;count++)      //(5) 赋值

{

\*array=count;

array++;

}

for(count=9;count>=0;count--)                  //(6)赋值

{

array--;

printf("%4d",\*array);

}

printf("\n");

free(array);        //(7)释放空间

array=NULL;       //(8)将指针置为空，避免不安全访问

exit (0);

}

在以上程序中，（1）句中包含stdio.h头文件，从而在后面可以调用printf()函数。（2）句中包含stdlib.h头文件，其是malloc()函数的头文件。（3）句为函数的入口位置，此处采用Linux下编程标准，返回值为int型，argc为参数个数， argv[]为参数，envp[]存放的是所有环境变量。（4）句动态分配了10个整型存储区域，此语句可以分为以下几步。

① 分配10个整型的连续存储空间，并返回一个指向其起始地址的整型指针。

② 把此整型指针地址赋给array。

③ 检测返回值是否为NULL。

（5）、（6）句为数组赋值并打印输出，以免内存泄漏。（7）句调用free()函数释放内存空间。（8）句将一个NULL指针传递给array，虽然在很多情况下可以不用此句，但这样处理可以避免此指针成为野指针。

#### 2.2.2realloc--更改已经配置的内存空间

realloc()函数用来从堆上分配内存，当需要扩大一块内存空间时，realloc()试图直接从堆上当前内存段后面的字节中获得更多的内存空间，如果能够满足，则返回原指针；如果当前内存段后面的空闲字节不够，那么就使用堆上第一个能够满足这一要求的内存块，将目前的数据复制到新的位置，而将原来的数据块释放掉。如果内存不足，重新申请空间失败，则返回NULL。此函数定义如下：

void \*realloc(void \*ptr,size\_t size)；

参数ptr为先前由malloc、calloc和realloc所返回的内存指针，而参数size为新配置的内存大小。其库头文件为：

＃include<stdlib.h>

当调用realloc()函数重新分配内存时，如果申请失败，将返回NULL，此时原来指针仍然有效，因此在程序编写时需要进行判断，如果调用成功，realloc()函数会重新分配一块新内存，并将原来的数据拷贝到新位置，返回新内存的指针，而释放掉原来指针（realloc()函数的参数指针）指向的空间，原来的指针变为不可用（即不需要再释放，也不能再释放），因此，一般不使用以下语句：

ptr=realloc(ptr,new\_amount)

如果内存减少，malloc仅仅改变索引信息，但并不代表被减少的部分还可以访问，这一部分内存将交给系统内存分配子程序。

下面是一个使用relloc函数的实例。

#include <stdio.h>

#include <stdlib.h>

int main (int argc,char\* argv[],char\* envp[]) //(1)主函数

{

int input;

int n;

int \*numbers1;

int \*numbers2;

numbers1=NULL;

if((numbers2=(int \*)malloc(5\*sizeof(int)))==NULL) //(2)numbers2指针申请空间

{

printf("malloc memory unsuccessful");

//free(numbers2);

numbers2=NULL;

exit(1);

}

for (n=0;n<5;n++) //(3)初始化

{

\*(numbers2+n)=n;

printf("numbers2's data: %d\n",\*(numbers2+n));

}

printf("Enter an integer value you want to remalloc ( enter 0 to stop)\n");//(4)新申请空间大小

scanf ("%d",&input);

numbers1=(int \*)realloc(numbers2,(input+5)\*sizeof(int)); //(5)重新申请空间

if (numbers1==NULL)

{

printf("Error (re)allocating memory");

exit (1);

}

for(n=0;n<5;n++) //(6)这5个数是从numbers2拷贝而来

{

printf("the numbers1s's data copy from numbers2: %d\n",\*(numbers1+n));

}

for(n=0;n<input;n++) //(7)新数据初始化

{

\*(numbers1+5+n)=n\*2;

printf ("nummber1's new data: %d\n",\*(numbers1+5+n)); // numbers1++;

}

printf("\n");

free(numbers1); //(8)释放numbers1

numbers1=NULL;

// free(numbers2); //(9)不能再释放numbers2

return 0;

}

[root@localhost yangzongde]# gcc -o realloc\_example realloc\_example.c

[root@localhost yangzongde]# ./realloc\_example

numbers2's data: 0

numbers2's data: 1

numbers2's data: 2

numbers2's data: 3

numbers2's data: 4

Enter an integer value you want to remalloc ( enter 0 to stop) //重新申请空间

5

the numbers1s's data copy from numbers2: 0

the numbers1s's data copy from numbers2: 1

the numbers1s's data copy from numbers2: 2

the numbers1s's data copy from numbers2: 3

the numbers1s's data copy from numbers2: 4

nummber1's new data: 0

nummber1's new data: 2

nummber1's new data: 4

nummber1's new data: 6

nummber1's new data: 8

此程序是一个简单的重新申请内存空间的实例，（1）为函数入口，前面已经介绍过。（2）从堆空间中申请5个int空间，将返回地址赋给numbers2，如果返回值为NULL，将返回错误信息，释放numbers2并退出。（3）为新申请的空间初始化。（4）输入需要增加的内存数量。（5）调用realloc()函数重新申请内存空间，重新申请内存空间大小为原有空间大小加上用户输入的内存空间数。如果申请失败，将返回NULL，此时numbers2仍然有效。如果申请成功，将重新分配一块大小合适的空间，并将新空间首地址赋给numbers1，同时将numbers2所指向的5个空间的数据复制到新的内存空间中，释放掉原来numbers2所指向的内存空间。（6）打印从numbers2所指向的原空间拷贝的数据，（7）句对新增加的空间进行初始化。（8）句释放number1所指向的新申请空间。（9）为注释掉的代码，提示读者此时对原空间再次释放，因为第（5）已经完成了这一操作。

## 三、常见的内存错误和对策

定义了指针变量，但是没有为指针分配内存，即指针没有指向一块合法的内存。

### 3.1结构体成员指针未初始化

struct student

{

char \*name;

int score;

}stu,\*pstu;

intmain()

{

strcpy(stu.name,"Jimy");

stu.score = 99;

return 0;

}

这里定义了结构体变量stu，但是他没想到这个结构体内部char \*name 这成员在定义结构体变量stu 时，只是给name 这个指针变量本身分配了4 个字节。name 指针并没有指向一个合法的地址，这时候其内部存的只是一些乱码。所以在调用strcpy 函数时，会将字符串"Jimy"往乱码所指的内存上拷贝，而这块内存name 指针根本就无权访问，导致出错。解决的办法是为name 指针malloc 一块空间。

同样，也有人犯如下错误：

intmain()

{

pstu = (struct student\*)malloc(sizeof(struct student));

strcpy(pstu->name,"Jimy");

pstu->score = 99;

free(pstu);

return 0;

}

为指针变量pstu 分配了内存，但是同样没有给name 指针分配内存。错误与上面第一种情况一样，解决的办法也一样。这里用了一个malloc 给人一种错觉，以为也给name 指针分配了内存

### 3.2没有为结构体指针分配足够的内存

intmain()

{

pstu = (struct student\*)malloc(sizeof(struct student\*));

strcpy(pstu->name,"Jimy");

pstu->score = 99;

free(pstu);

return 0;

}

为pstu 分配内存的时候，分配的内存大小不合适。这里把sizeof(struct student)误写为sizeof(struct student\*)。当然name 指针同样没有被分配内存。解决办法同上。

### 3.3函数的入口校验

不管什么时候，我们使用指针之前一定要确保指针是有效的。一般在函数入口处使用assert(NULL != p)对参数进行校验。在非参数的地方使用if（NULL != p）来校验。但这都有一个要求，即p 在定义的同时被初始化为NULL 了。比如上面的例子，即使用if（NULL != p）校验也起不了作用，因为name 指针并没有被初始化为NULL，其内部是一个非NULL 的乱码。

assert 是一个宏，而不是函数，包含在assert.h 头文件中。如果其后面括号里的值为假，则程序终止运行，并提示出错；如果后面括号里的值为真，则继续运行后面的代码。这个宏只在Debug 版本上起作用，而在Release 版本被编译器完全优化掉，这样就不会影响代码的性能。有人也许会问，既然在Release 版本被编译器完全优化掉，那Release 版本是不是就完全没有这个参数入口校验了呢？这样的话那不就跟不使用它效果一样吗？是的，使用assert 宏的地方在Release 版本里面确实没有了这些校验。但是我们要知道，assert 宏只是帮助我们调试代码用的，它的一切作用就是让我们尽可能的在调试函数的时候把错误排除掉，而不是等到Release 之后。它本身并没有除错功能。再有一点就是，参数出现错误并非本函数有问题，而是调用者传过来的实参有问题。assert 宏可以帮助我们定位错误，而不是排除错误。

为指针分配的内存太小

为指针分配了内存，但是内存大小不够，导致出现越界错误。

char \*p1 = “abcdefg”;

char \*p2 = (char \*)malloc(sizeof(char)\*strlen(p1));

strcpy(p2,p1);

p1 是字符串常量，其长度为7 个字符，但其所占内存大小为8 个byte。初学者往往忘了字符串常量的结束标志“\0”。这样的话将导致p1 字符串中最后一个空字符“\0”没有被

拷贝到p2 中。解决的办法是加上这个字符串结束标志符：

char \*p2 = (char \*)malloc(sizeof(char)\*strlen(p1)+1\*sizeof(char));

这里需要注意的是，只有字符串常量才有结束标志符。比如下面这种写法就没有结束标志符了：

char a[7] = {‘a’,’b’,’c’,’d’,’e’,’f’,’g’};

另外，不要因为char 类型大小为1 个byte 就省略sizof（char）这种写法。这样只会使你的代码可移植性下降。

### 3.4内存分配成功，但并未初始化

犯这个错误往往是由于没有初始化的概念或者是以为内存分配好之后其值自然为0。未初始化指针变量也许看起来不那么严重，但是它确确实实是个非常严重的问题，而且往往

出现这种错误很难找到原因。曾经有一个学生在写一个windows 程序时，想调用字库的某个字体。而调用这个字库需要填充一个结构体。他很自然的定义了一个结构体变量，然后把他想要的字库代码赋值给了相关的变量。但是，问题就来了，不管怎么调试，他所需要的这种字体效果总是不出来。我在检查了他的代码之后，没有发现什么问题，于是单步调试。在观察这个结构体变量的内存时，发现有几个成员的值为乱码。就是其中某一个乱码惹得祸！因为系统会按照这个结构体中的某些特定成员的值去字库中寻找匹配的字体，当这些值与字库中某种字体的某些项匹配时，就调用这种字体。但是很不幸，正是因为这几个乱码，导致没有找到相匹配的字体！因为系统并无法区分什么数据是乱码，什么数据是有效的数据。只要有数据，系统就理所当然的认为它是有效的。也许这种严重的问题并不多见，但是也绝不能掉以轻心。所以在定义一个变量时，第一件事就是初始化。你可以把它初始化为一个有效的值，比如:

int i = 10；

char \*p = (char \*)malloc(sizeof(char))；

但是往往这个时候我们还不确定这个变量的初值，这样的话可以初始化为0 或NULL。

int i = 0；

char \*p = NULL；

如果定义的是数组的话，可以这样初始化：

int a[10] = {0};

或者用memset 函数来初始化为0：

memset（a,0,sizeof(a)）;

memset 函数有三个参数，第一个是要被设置的内存起始地址；第二个参数是要被设置的值；第三个参数是要被设置的内存大小，单位为byte。这里并不想过多的讨论memset 函数的用法，如果想了解更多，请参考相关资料。至于指针变量如果未被初始化，会导致if 语句或assert 宏校验失败。这一点，上面已有分析。

### 3.5内存越界

内存分配成功，且已经初始化，但是操作越过了内存的边界。这种错误经常是由于操作数组或指针时出现“多1”或“少1”。比如：

int a[10] = {0};

for (i=0; i<=10; i++)

{

a[i] = i;

}

所以，for 循环的循环变量一定要使用半开半闭的区间，而且如果不是特殊情况，循环变量尽量从0 开始。

3.6如何避免产生野指针

这里列出了5条规则，平常写程序时多注意一下，养成良好的习惯。

规则1：用malloc或new申请内存之后，应该立即检查指针值是否为NULL。防止使用指针值为NULL的内存。

规则2：不要忘记为数组和动态内存赋初值。防止将未被初始化的内存作为右值使用。

规则3：避免数组或指针的下标越界，特别要当心发生“多1”或者“少1”操作。

规则4：动态内存的申请与释放必须配对，防止内存泄漏。

规则5：用free或delete释放了内存之后，立即将指针设置为NULL，防止产生“野指针”。

#include <stdio.h>

#include <malloc.h>

//全局变量定义

int iGlobalInt1=0;

int iGlobalInt2=0;

int iGlobalInt3=0;

//全局常量定义

const int iGlobalConstInt1=1;

const int iGlobalConstInt2=5;

const int iGlobalConstInt3=6;

//全局静态变量定义

static int iGlobalStaticInt1=0;

static int iGlobalStaticInt2=0;

static int iGlobalStaticInt3=0;

//函数定义

void funcParamTest(int iFuncParam1,int iFuncParam2,int iFuncParam3)

{

//函数私有变量定义

int iLocalInt1=iFuncParam1;

int iLocalInt2=iFuncParam2;

int iLocalInt3=iFuncParam3;

printf("函数参数变量内存地址\n");

printf("iFuncParam1=0x%08x\n",&iFuncParam1);

printf("iFuncParam2=0x%08x\n",&iFuncParam2);

printf("iFuncParam3=0x%08x\n\n",&iFuncParam3);

printf("函数本地变量的内存地址\n");

printf("iLocalInt1=0x%08x\n",&iLocalInt1);

printf("iLocalInt2=0x%08x\n",&iLocalInt2);

printf("iLocalInt3=0x%08x\n\n",&iLocalInt3);

return;

}

//入口函数

int main(int argc, char\* argv[])

{

//局部静态变量

static int iStaticInt1=0;

static int iStaticInt2=0;

static int iStaticInt3=0;

//局部静态常量定义

const static int iConstStaticInt1=0;

const static int iConstStaticInt2=0;

const static int iConstStaticInt3=0;

//局部常量

const int iConstInt1=1;

const int iConstInt2=5;

const int iConstInt3=6;

//局部变量

int iLocalInt1=0;

int iLocalInt2=0;

int iLocalInt3=0;

char \* pMalloc1,\*pMalloc2,\*pMalloc3;

char \* pNew1,\*pNew2,\*pNew3;

printf("全局常量的内存地址\n");

printf("iGlobalConstInt1=0x%08x\n",&iGlobalConstInt1);

printf("iGlobalConstInt2=0x%08x\n",&iGlobalConstInt2);

printf("iGlobalConstInt3=0x%08x\n\n",&iGlobalConstInt3);

printf("iConstStaticInt1=0x%08x\n",&iConstStaticInt1);

printf("iConstStaticInt2=0x%08x\n",&iConstStaticInt2);

printf("iConstStaticInt3=0x%08x\n\n",&iConstStaticInt3);

printf("全局变量的内存地址\n");

printf("iGlobalInt1=0x%08x\n",&iGlobalInt1);

printf("iGlobalInt2=0x%08x\n",&iGlobalInt2);

printf("iGlobalInt3=0x%08x\n\n",&iGlobalInt3);

printf("全局静态变量的内存地址\n");

printf("iGlobalStaticInt1=0x%08x\n",&iGlobalStaticInt1);

printf("iGlobalStaticInt2=0x%08x\n",&iGlobalStaticInt2);

printf("iGlobalStaticInt3=0x%08x\n\n",&iGlobalStaticInt3);

printf("局部静态变量的内存地址\n");

printf("iStaticInt1=0x%08x\n",&iStaticInt1);

printf("iStaticInt2=0x%08x\n",&iStaticInt2);

printf("iStaticInt3=0x%08x\n\n",&iStaticInt3);

printf("局部常量的内存地址\n");

printf("iConstInt1=0x%08x\n",&iConstInt1);

printf("iConstInt2=0x%08x\n",&iConstInt2);

printf("iConstInt3=0x%08x\n\n",&iConstInt3);

printf("局部变量的内存地址\n");

printf("iLocalInt1=0x%08x\n",&iLocalInt1);

printf("iLocalInt2=0x%08x\n",&iLocalInt2);

printf("iLocalInt3=0x%08x\n\n",&iLocalInt3);

printf("开始调用函数\n");

funcParamTest(iLocalInt1,iLocalInt2,iLocalInt3);

//在堆上分配内存，使用new

pNew1=new char[16];

pNew2=new char[16];

pNew3=new char[16];

printf("在堆上分配内存地址\n");

printf("pNew1=0x%08x\n",pNew1);

printf("pNew2=0x%08x\n",pNew2);

printf("pNew3=0x%08x\n\n",pNew3);

//在堆上分配内存，使用malloc

pMalloc1 = (char \*)malloc( 16 );

pMalloc2 = (char \*)malloc( 16 );

pMalloc3 = (char \*)malloc( 16 );

printf("在堆上分配内存地址\n");

printf("pMalloc1=0x%08x\n",pMalloc1);

printf("pMalloc2=0x%08x\n",pMalloc2);

printf("pMalloc3=0x%08x\n\n",pMalloc3);

//释放new 分配的内存空间

delete [] pNew1;

delete [] pNew2;

delete [] pNew3;

pNew1=NULL;

pNew2=NULL;

pNew3=NULL;

//释放malloc 分配的内存空间

free(pMalloc1);

free(pMalloc2);

free(pMalloc3);

pMalloc1=NULL;

pMalloc2=NULL;

pMalloc3=NULL;

return 0;

}

# 实训任务

## 编写代码查看各个段的信息

/\*代码段、数据段和BSS段存储变量类型\*/

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

#include <string.h>

const int g\_A = 10; //代码段

int g\_B = 20; //数据段

static int g\_C = 30; //数据段

static int g\_D; //BSS段

int g\_E; //BSS段

char \*p1; //BSS段

void main( )

{

int local\_A; //栈

static int local\_C = 0; //数据段

static int local\_D; //数据段

char \*p3 = "123456"; //123456在代码段，p3在栈上

p1 = (char \*)malloc( 10 ); //堆，分配得来得10字节的区域在堆区

strcpy( p1, "123456" ); //123456{post.content}放在常量区，编译器可能会将它与p3所指向 的"123456"优化成一块

printf("\n");

printf( "代码段,全局初始化变量, 只读const, g\_A, addr:0x%08x\n", &g\_A);

printf("\n");

printf( "数据段,全局变量, 初始化 g\_B, addr:0x%08x\n", &g\_B);

printf( "数据段,静态全局变量, 初始化, g\_C, addr:0x%08x\n", &g\_C);

printf("\n");

printf( "BSS段, 全局变量, 未初始化 g\_E, addr:0x%08x\n", &g\_E, g\_E );

printf( "BSS段, 静态全局变量, 未初始化, g\_D, addr:0x%08x\n", &g\_D );

printf( "BSS段, 静态局部变量, 初始化, local\_C, addr:0x%08x\n", &local\_C);

printf( "BSS段, 静态局部变量, 未初始化, local\_D, addr:0x%08x\n", &local\_D);

printf("\n");

printf( "栈, 局部变量, local\_A, addr:0x%08x\n", &local\_A );

printf("\n");

printf( "堆, malloc分配内存, p1, addr:0x%08x\n", p1 );

}

## 使用malloc函数分配内存

创建一个 结构体类型的指针，其中包含两个成员，一个是整型，另一个是结构体指针。使用malloc函数分配一个结构体的内存空间，然后给两个成员赋值，并显示出来。

## 调用calloc()函数动态分配内存

调用时calloc函数动态分配内存存放若干个数据。该函数返回值为分配域的起始地址；如果分配不成功，返回0。



## 动态内存分配

动态内存分配一个常见的用途就是为那些长度在运行时才知的数组分配内存空间。编写一个程序，要求读取一列整数，并按升序排列它们，最后打印这个列表。

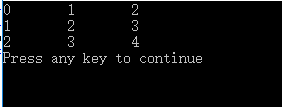
## 为一维数组分配内存

为一个具有10个元素的一维数组动态分配内存，为元素赋值并输出。

运行：

## 为二维数组动态分配内存

设计一个程序，为二维数组动态分配内存并释放内存空间。数组元素的赋值结果如下：



## 以空格分隔字符串

输入一行字符串，要求以空格分隔每个单词，并输出，运行结果：

要求：

实现函数char \*\*splitstr( char \*src, char ch)

src:待分隔的字符串

ch:分隔符，也就是空格

返回值：分隔后的所有单词

要求使用malloc动态分配内存存储每个单词

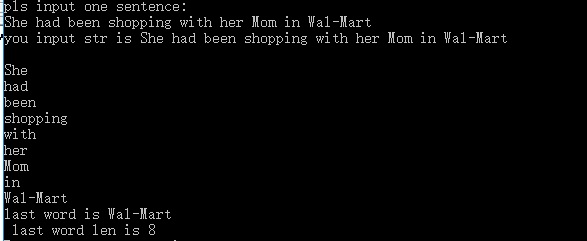
## 计算单词长度

计算字符串最后一个单词的长度，单词以空格隔开。

输入描述:

一行字符串，非空，长度小于5000。

运行：



## 拆分字串

连续输入字符串(输出次数为N,字符串长度小于100)，请按长度为8拆分每个字符串后输出到新的字符串数组，

长度不是8整数倍的字符串请在后面补数字0，空字符串不处理。

首先输入一个整数，为要输入的字符串个数。

例如：

输入：2

      abc

      12345789

输出：abc00000

      12345678

      90000000

输入描述:

首先输入数字n，表示要输入多少个字符串。连续输入字符串(输出次数为N,字符串长度小于：100)。

输出描述:

按长度为8拆分每个字符串后输出到新的字符串数组，长度不是8整数倍的字符串请在后面补数字0，空字符串不处理。

**示例1**

输入

2

abc

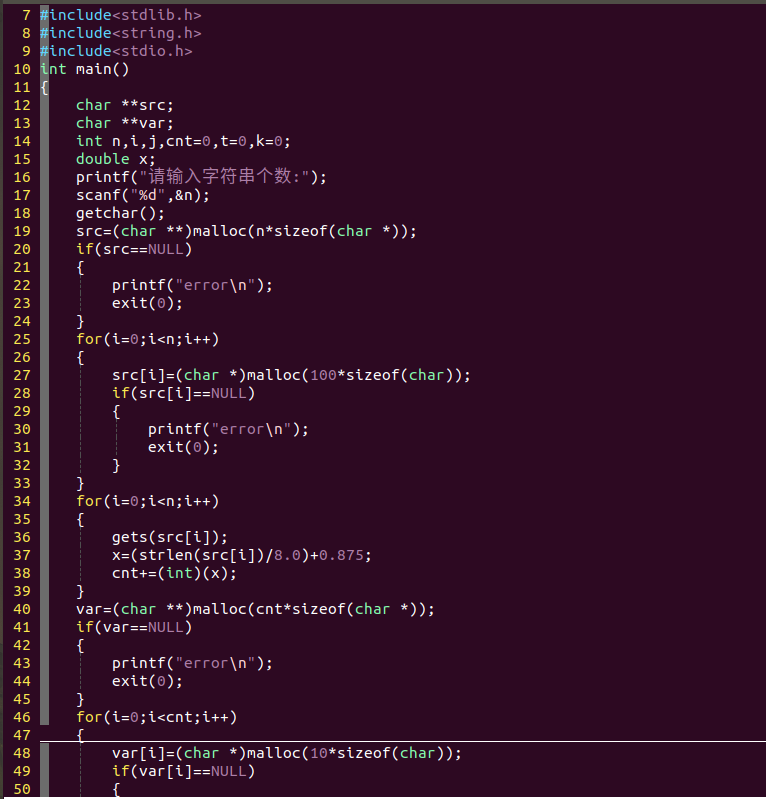
123456789

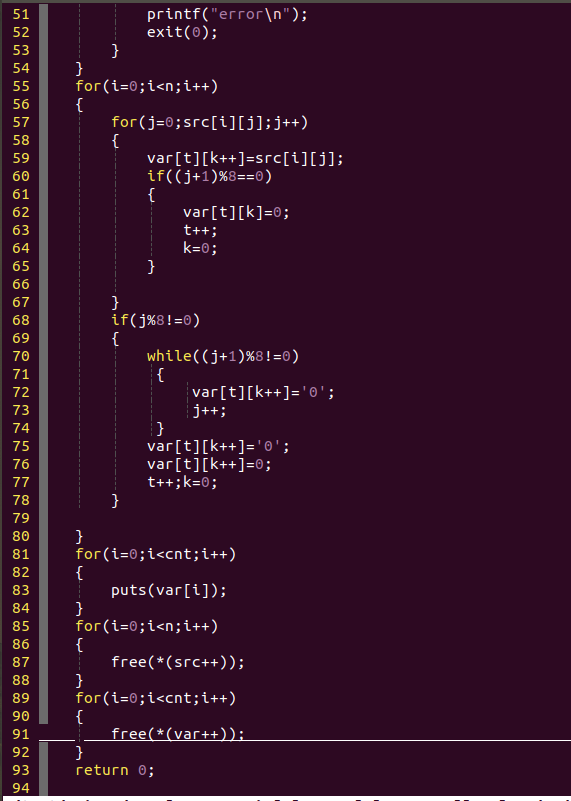
输出

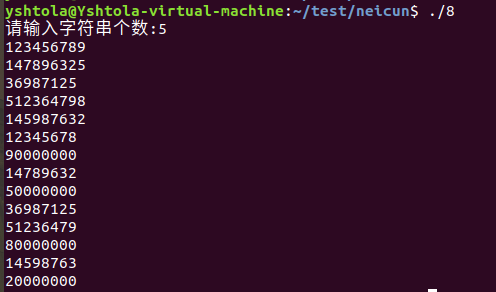
abc00000

12345678

90000000







## DNA序列

题目描述

一个DNA序列由A/C/G/T四个字母的排列组合组成。G和C的比例（定义为GC-Ratio）是序列中G和C两个字母的总的出现次数除以总的字母数目（也就是序列长度）。在基因工程中，这个比例非常重要。因为高的GC-Ratio可能是基因的起始点。

给定一个很长的DNA序列，以及要求的最小子序列长度，研究人员经常会需要在其中找出GC-Ratio最高的子序列。

输入描述:

输入一个string型基因序列，和int型子串的长度

输出描述:

找出GC比例最高的子串,如果有多个输出第一个的子串

示例1

输入

AACTGTGCACGACCTGA

5

输出

GCACG：