



内存管理

程序设计的本质

所谓程序就是由语句来操纵存放在计算机的内存中的数据。

程序设计就是对内存直接或间接地使用。



可见，了解内存的分配和活动乃是编程者的第一要务。



变量的属性

对变量起作用的属性有两个：数据类型和存储类型。

存储类型决定着该变量的存储区域，存储区域又决定着变量的作用域和存储时期。

同一存储区域内的所有变量具有相同的属性，不同存储区域的变量各有不同的属性。

不同的存储类型提供了变量的作用域、存储区域的链接以及存储时期的不同组合。

变量的属性

所谓作用域就是一个标识符能够起作用的程序范围。

从作用域的角度看，变量可分为：
局部变量、全局变量。

一个变量有以下两个存储时期之一：

静态存储时期（static修饰）和自动存储时期（auto修饰）

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

```
int max ( )
```

```
{
```

```
    int a;
```

```
    static int n=0;
```

```
}
```

```
void main( )
```

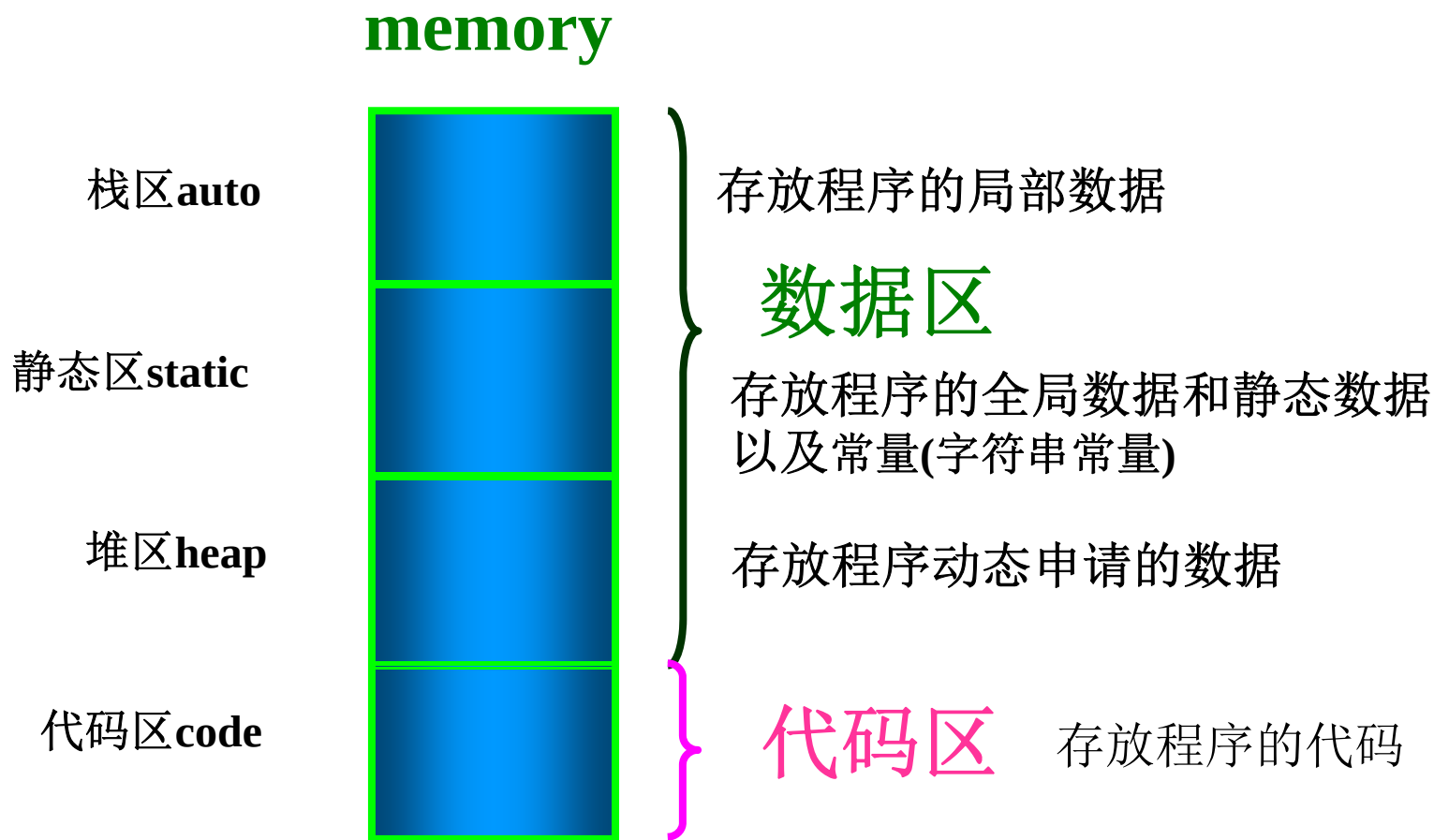
```
{
```

```
    int b;
```

```
}
```

内存划分

分为4个区域:



内存划分

栈区（stack区）

- 由系统按栈原则管理
- 编译时就已经“规划”好了，运行时才使用
- 变量短命，函数执行完则释放。
- 用户无法干预变量的诞生和消亡

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

```
int fun()
```

```
{
```

```
    int i = 0;
```

```
    char j = 1;
```

```
    float k = 2.0f;
```

```
    printf("%d %c %f", i, j, k);
```

```
    return 0;
```

```
}
```

&i	0x0013ff7c
&j	0x0013ff78
&k	0x0013ff74

内存划分

```
void fun1( )
{int n = 0;.....}
void fun2( )
{int n = 1;.....}
int main( )
{
    int n = 0;
    fun1( );
    fun2( );
    return 0;
}
```

fun2函数栈

主函数栈

千万不要返回指向栈内存的指针！



内存划分

静态区(static区)

- 编译时已确定并初始化了;
- 用户无法干预;
- 不按栈原则管理,但仍受“作用域”的制约。

静态区存有全局变量、局部静态变量和常量。

```
void fun( )
{
    static int n = 0;
    printf("%d\n",n);
}
int main( )
{
    fun( );
    printf("%d\n",n);
    return 1;}

```

```
void fun( )
{
    int n = 0;
    printf("%d\n",n);
}
int main( )
{
    fun( );
    printf("%d\n",n);
    return 1;}

```



内存划分

堆区（heap区）

- 长命
- 无名
- 不受作用域限制
- 用户可以干预变量的诞生和消亡
- 在程序运行时才动态分配
- 申请可能失败

代码区（code区）

- 存有程序的所有执行代码
- 每段代码都有名称—函数名
- 用户无法干预，只能通过函数名调用
- 程序结束即释放

内存分类

内存类型：

- 静态内存：程序在编译的时候就已经分配好，这块内存存在程序的整个运行期间都存在。
- 动态内存：程序在运行的时候创建的。包括堆内存、栈内存。

内存分配方式相应分为：



静态分配
栈式分配
堆式分配

内存划分

	分配时机/分配责任	释放时机/释放责任
静态内存 (静态分配)	这种内存在编译时就规划好了，在程序运行期间一直存在。是由编译器负责分配的。	程序退出时释放，不用程序员参与。
栈内存 (栈式分配)	函数调用临时创建的。为函数返回、函数形参、函数中定义的局部变量在栈里申请空间。	函数退出时释放，不用程序员参与。
堆内存 (堆式分配)	程序员根据需要用 malloc 等函数创建的内存。	在适当时机由程序员用 free 释放。

判断数据所在的区域

静态区：全局变量（靠位置分辨），局部静态变量（靠static分辨），常量（靠自身分辨）。

栈 区：局部变量、函数的形参。

堆 区：堆变量（靠申请分辨）。

内存划分

```
#include<stdio.h>
int func(void)
{
    static int i=0;
    return ++i;
}
void main(void)
{
    int i;
    i = func( );
    i = func( );
    printf(" %d\n",i);
}
```

2

static局部变量
只做一次初始化，而且这次初始化的时机是在编译的时候，而不是运行的时候。

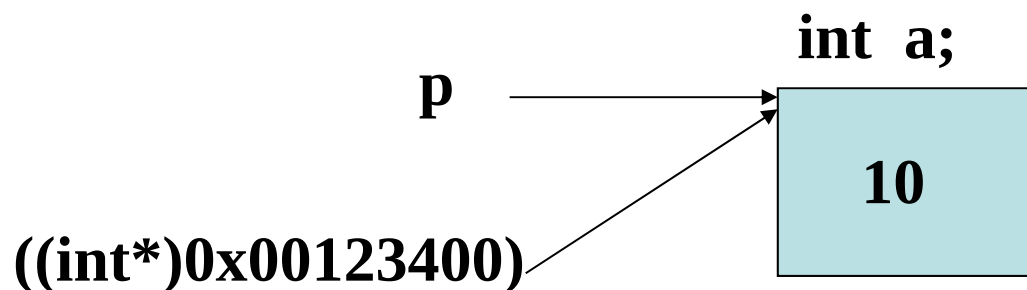


静态变量的特点

- **static**变量分配在静态存储区, 在程序运行期间不释放。
- **static**局部变量虽然在程序运行期间一直存在, 在程序结束时才释放, 但是它的作用域也仅仅是在函数或者块的内部才可见。在函数外部, 我们无法使用变量名来访问, 尽管如此我们可以使用指针在它不可见的位置来访问它。
- **static**局部变量在编译时进行初始化, 且只作一次。如果不赋初值, 编译期会自动赋初值0。

对内存的访问

- 通过变量名访问
如 `int a;` 则可以用 `a` 来访问变量的内容;
- 通过地址访问
如 `int a;` (假设 `a` 的地址值为 `0x00123400`), 则可以通过 `*((int*)0x00123400)` 来访问 `a` 的内容。
- 通过指针变量访问
如 `int a; int *p=&a;` 则可以用 `*p` 来访问变量的内容;





动态分配内存

C语言的动态内存分配核心函数有：

- malloc()、calloc()分配堆内存
- free() 释放堆内存

使用这三个函数必须包含头文件**stdlib.h**。

```
# include stdlib.h
```



内存的使用

- malloc()
- 函数原型：void* malloc(size_t size);
- 功 能：从堆内存中分配连续的大小为size字节的块。
- 返 回 值：若有足够的内存，则返回一个指向新分配内存地址的指针(注意为void *类型)；否则返回NULL。实际应用中需要使用(类型 *)把返回值强制转换为特定类型的地址。

内存的使用

```
#include <stdio.h>
#include<stdlib.h>
void main()
{
    char *p; //如果改成int *p;
    int i =0;
    p=malloc(40*sizeof(char));//给指针p赋40个字节的动态空间
    // p = (int *)malloc(10*sizeof(int)); 强制类型转换
    for(i=0; i < 40; i++) //就得改成i<10
    {
        *(p + i)='1';
        printf("%c",*(p + i));
    }
    free (p);
}
```

内存的使用

- `calloc()`
- 函数原型: `void *calloc( size_t num, size_t size );`
- 功能: 申请分配一块内存, 并将该块内存全部“清零”。
- 返回值: 若内存足够, 返回一个指向新分配内存地址的`void`类型的指针, 否则返回`NULL`。

注意: 该函数有两个参数, 第一个参数设定元素的个数, 第二个参数设定每个元素的大小。



`malloc`与`calloc`的区别?

`calloc` 指定了个数, 而且赋了初值。



内存的使用

- `free( )`
- 函数原型: `void free( void *memblock );`
- 功 能: 用来释放被`malloc( )`或`calloc( )`分配的内存空间。

注意: 在程序的执行过程中, 函数体内的局部指针变量在函数结束时会自动消亡, 但是它所指向的动态分配的内存却不会自动释放。需要`free( )`函数来完成该任务。

释放的是整个块, 而不是指向该块的指针类型所指的大小。

例如: `int *p = (int *)malloc(sizeof(int)*10);`
`free(p);` //所释放的是40个字节而非4个字节。

动态内存分配的实现

- 说明：指向动态分配的内存的指针是靠“-1”单元所记载的块大小等信息来行动的。所谓“-1”单元是该块前的某个空间。
- 使用malloc()函数分配空间时，编译器会记下总空间大小，而不是利用指针来判断大小，以便于完全释放。
- 注意：千万不要对非NULL指针所指向的堆空间重复释放。

不要将堆空间重复释放

```
int main( )
{
    int *p = (int *)malloc(40);
    free(p);
    //p = NULL; //如果没有这条语句的
                后果?
    free(p);    //编译器不会检查是否重
                复释放! 运行时出错!
    return 0;
}
```

内存的使用



```
int main( void )
{  char *pstring;
   pstring = (char *) malloc( 500 );
   if( NULL == pstring )
       printf( "Insufficient memory available\n" );
   else{
       printf(" Space allocated for path
name\n" );
       free( pstring );
       pstring = NULL;
       printf( "Memory freed\n" );
   }
   return 1;
}
```

内存的使用



```
#include<stdio.h>
#include<stdlib.h>
#include <string.h>
int main( )
{
    char *p = (char *)malloc(8);
    strcpy(p, "1234567");
    printf("%s\n",p); //能否正确打印?
    free(p);          //能否正确释放?
    return 1;
}
```



内存使用注意事项

- 用**malloc** 申请内存之后，应该立即检查指针值是否为 **NULL**，防止使用值为**NULL**的指针。
- 避免未初始化的内存作右值。
- 避免数组或指针的下标越界，特别要当心“边界值”问题。
- 动态内存的申请与释放必须配对，防止内存泄漏。
- 用**free**释放了内存之后，立即将指针设置为**NULL**，防止“野指针”的产生，也防止了内存的重复释放。

内存使用注意事项

数组或指针的下标越界

```
int main( )
{
    int i = 0;
    int a[5] = {0};
    int count = 0;
    for(i = 0; i <= 5; i++) //结果如何?
    {
        a[i] = 1;
        printf("执行了%d次循环.\n", ++count);
    }
    return 0;
}
```



内存使用注意事项

内存释放不代表指针消亡

```
free(p);  
p = NULL;
```

内存被释放了，并不表示指向这块内存的指针会消亡或者置为了NULL。

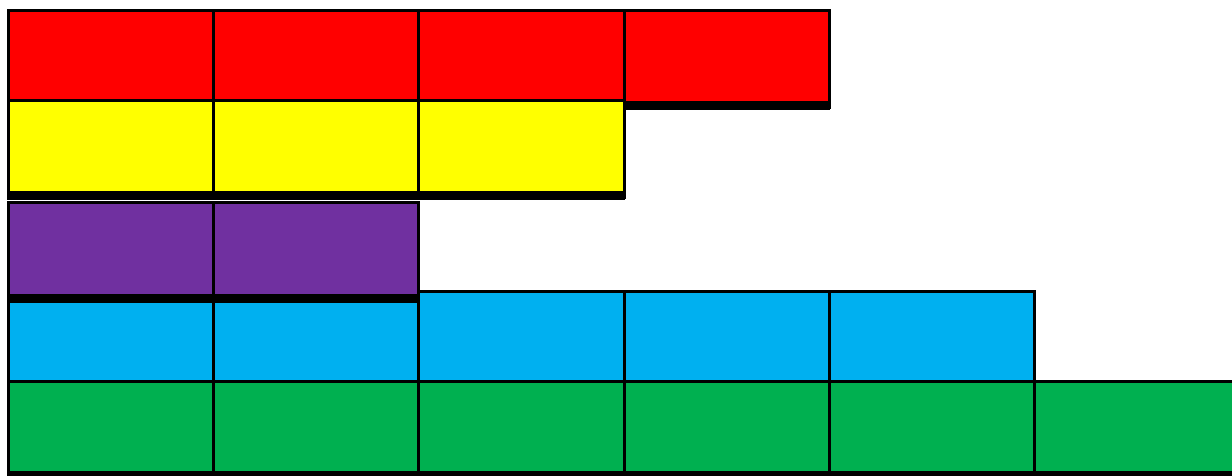
指针消亡不代表内存释放

```
int* MyMalloc(int n)  
{  
    int *p = NULL;  
    p = (int *)malloc(sizeof(int)*n);  
    return p;  
}
```

练习

实现一个“可变长二维数组”，这个二维数组的行数可由输入决定，每行的元素个数仍可由输入决定。

例如：如下“数组”有五行，每行的元素如图。



注意：要求有正确的释放语句。

预期结果

- 请输入行数:
- 5
- 请输入第1行的元素个数:
- 4
- 请输入第2行的元素个数:
- 3
- 请输入第3行的元素个数:
- 2
- 请输入第4行的元素个数:
- 5
- 请输入第5行的元素个数:
- 6
- 11111
- 1111
- 11
- 11111
- 111111
- Press any key to continue