第七章：存储类、作用域、生命周期、链接属性

存储类

- 定义：存储类型，描述C语言变量在哪个内存段中存储（局部变量分配在栈上，存储类是栈...）
- **linux下C程序的内存映像**
  - linux内存模型（C语言程序在linux环境下怎么存）
    - 每个进程认为自己独享全部内存空间，认为整个系统只有它自己和内核，因为用了虚拟地址

地址	作用	说明
>=0xc000 0000	内核虚拟存储器	用户代码不可见区域
<0xc000 0000	Stack（用户栈）	ESP指向栈顶
	↓ ↑	空闲内存
>=0x4000 0000	文件映射区	
<0x4000 0000		空闲内存
	Heap(运行时堆)	通过brk/sbrk系统调用扩大堆，向上增长。
	.data、.bss(读写段)	从可执行文件中加载
>=0x0804 8000	.init、.text、.rodata(只读段)	从可执行文件中加载
<0x0804 8000	保留区域	

  - 1、**内核映射区：**操作系统内核映射到该区域
  - 2、**栈段**：非static局部变量、函数调用传参
  - 3、**文件映射区：**进程打开文件后，把文件内容从硬盘读到文件映射区，读写结束把文件写回硬盘（操作文件在内存中进行）
  - 4、**堆段：**程序员手工操作，C语言不会自动向堆里存东西
  - 5、**.data、.bss**（数据段，用于读写）
    - .data数据段：显式初始化为非0的全局变量/static局部变量
    - .bss段：显式初始化为0或未显式初始化（默认初始化为0）的全局变量/static局部变量（值为0的特殊数据段）

        - .data数据段和.bss段本质一样，分开是c语言编译器优化的结果，数据段中的内容需要一个个的复制（读写内存多）
    - 6、**.text、.rodata**（只读段）
        - 代码段在linux中称为文本段 .text
        - 只读数据段：程序运行时只能读不能写，烧录下载时可以写，const常量可能在只读数据段中（不同平台不一样，linux中放在普通数据段）
- **存储类相关关键字**
    - auto：自动局部变量，分配在栈上，只能修饰局部变量（不初始化值随机；平时定义局部变量默认都是auto的）
    - **static：最复杂，面试常考，两种独立用法**
        - **1、静态局部变量**
            - **静态局部变量、非静态局部变量本质区别是存储类不同（表现是生命周期不同）**
            - **非static局部变量分配在栈上，static局部变量分配在data段/bss段上**
        - **2、静态函数、静态全局变量**

	存储类	作用域	生命周期	链接属性	其它
普通局部变量（auto)	栈	代码块作用域	临时	无链接	未显示初始化值随机
静态局部变量	数据段、bss段	代码块作用域	永久	无链接	未显示初始化值为0；变量地址在程序加载时决定，运行时唯一
普通全局变量/函数	数据段、bss段/*	文件作用域	永久	外链接（所有文件范围）	
静态全局变量/函数	数据段、bss段/*	文件作用域	永久	内链接（当前c文件）	

    - register：不常用，编译器尽量分配在寄存器中（不保证，因为寄存器数量有限），读写效率更高

- 用在变量反复使用的场景，改变变量访问效率可极大提升运行效率，是一种极致提升程序运行效率的手段
- uboot中用到一个register类型的变量gd-global data，用来存uboot的全局变量
- 慎用，想用的话先写好程序再换成register测试，测试功能是否正常、效率是否提高

- extern：
  - **使用场景：在b.c中定义全局变量，a.c中使用变量，则要在a.c中声明**

示例

```c
//a.c文件
#include<stdio.h>
extern g_b;//生命使用外部的g_b
extern g_c;
int main(void)
{
  printf("g_b=%d.\n",g_b);
  printf("g_c=%d.\n",g_c);//可以在本文件用本文件生命，但是没有意义
  return 0;
}

//b.c文件
int g_b=4;
```

- volatile：编译器不会对声明为volatile的变量做优化
  - **使用场景：**
    - **变量可能会被编译器之外的东西改变（改变不是当前代码造成，编译器编译时无法预知）**
      - 大多情况下编译器优化有助于提升效率

编译器优化示例

```c
int a, b, c;
a = 3;
b = a;
c = b;//等效于c=a=b=3
//无优化，内存读3次，写3次；优化后，内存读1次，写3次
```

      - 变量可能会被其它力量改变时，可能会带来优化错误
    - **不知道该不该加→加**（在不该加volatile的地方加了volatile不会出错，只是会降低效率）

        - 变量由非当前代码改变的三种场景：
            - 中断处理程序ISR改变该变量的值
            - 多线程在别的线程中更改了该变量的值
            - 硬件自动改变了该变量的值（这种情况下这个变量一般是一个寄存器的值）
    - restrict：只修饰指针，不修饰普通变量（没有意义）
        - 该指针指向的内容只能通过该指针访问，不能通过其他方式访问，告诉编译器可以进行优化
        - C89不支持，C99支持，gcc支持
    - typedef：属于存储类关键字，但和存储类没有关系

作用域（变量起作用的范围）

- 作用域规则
    - 局部变量作用域为自己所在代码块中定义之后的部分（代码块：一对大括号括起的部分）
    - 函数名和全局变量为文件作用域（整个.c文件内定义之后的部分）
    - 总结：准确的说，每个变量的作用域都是自己所在代码块/文件，但是定义式之前缺少声明因此没法用→定义到前面/定义到后面，但在前面加声明
- 同名变量作用域
    - 同名变量作用域不同且无交叠→无影响
    - 同名变量作用域有交叠→作用域小的掩蔽掉作用域大的

变量的生命周期

- 栈变量（局部变量）：临时生命周期，函数返回时变量消亡
- 堆变量：malloc开始free消亡
- 数据段、bss段变量：永久生命周期，程序终止时消亡（申请过多全局变量会导致程序占用大量内存）
- 代码段、只读段：永久生命周期

链接属性

- 链接本质：根据符号（函数名、变量名），找哪里定义过，再链接起来

    - 编译后生成.o目标文件，目标文件中有很多符号（编程中的变量名、函数名）、代码段、数据段、bss段
    - 链接时.o目标文件→最终可执行程序，符号和对应的段链接起来（如：一个函数调用另一个函数时，已知符号，从别的.o文件中根据符号找到代码段，找到的代码和现有代码链接起来）
- 变量的三种链接属性
    - 外链接：链接时找符号的工作可以跨文件，**普通函数和全局变量**
    - 内链接：链接时找符号的工作只能在当前文件内部（不能在其它.c文件中访问当前.c文件）**static修饰的函数/全局变量**
    - 无链接：符号本身不参与链接 **所有局部变量、宏和inline函数**
- 程序生成=编译+ 链接
    - 编译：c语言程序→一个个独立的二进制机器码
    - 链接：将独立的二进制程序按照一定格式组织起来，形成整体的二进制可执行程序
    - 编译以文件为单位（编译a.c时不考虑b.c）
    - 链接以工程为单位（多个.o作为整体输入，链接成可执行程序，多少个.c文件就会生成多少个.o文件）
- 普通函数和全局变量的命名冲突
    - 普通函数和全局变量都是外部链接→一个程序里所有.c文件中不能出现同名的函数/全局变量
    - 解决方法：
        - 是不起同样的名字（很难做到）
        - 高级语言中：namespace，给各个变量加上各个级别的前缀
        - C语言中：将只会在当前.c文件中引用的函数/全局变量使用static修饰（没有从根本上解决问题，因此c语言写大型项目难度很大）

其它

- os下和裸机下C程序加载的差异
    - 裸机环境：个人写的C语言程序没法在内存中直接运行，需要加载运行代码协助，主要作用是给全局变量赋值、清bss段
    - os环境：os帮助自动完成

- c89标准编译器，局部变量必须定义在最前面，c99标准编译器（gcc兼容c99）允许在函数中任意位置定义变量
- 全局变量起名前面加上g_