分段存储

BSS 段：BSS 段（bss segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。BSS 是英文 Block Started by Symbol 的简称。BSS 段属于静态内存分配。

数据段：数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

代码段：代码段（code segment/text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读，某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

堆（heap）：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。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）

栈(stack)：栈又称堆栈， 是用户存放程序临时创建的局部变量，也就是说我们函数括弧“{}”中定义的变量（但不包括 static 声明的变量，static 意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进先出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。

首先看下面的例子：

```
#include <stdio.h>
const int A = 10;
int a = 20;
static int b = 30;
int c;

int main(void)
{
   static int a = 40;
   char b[] = "Hello world";
   register int c = 50;

   printf("Hello world %d\n", c);

   return 0;
}
```

我们在全局作用域和 main 函数的局部作用域各定义了一些变量，并且引入一些新的关键字

const、static、register 来修饰变量，那么这些变量的存储空间是怎么分配的呢？我们编译之后用 readelf 命令看它的符号表，了解各变量的地址分布。注意在下面的清单中我把符号表按地址从低到高的顺序重新排列了，并且只截取我们关心的那几行。

```
$ gcc main.c -g
$ readelf -a a.out
...
    68: 08048540     4 OBJECT  GLOBAL DEFAULT    15 A
    69: 0804a018     4 OBJECT  GLOBAL DEFAULT    23 a
    52: 0804a01c     4 OBJECT  LOCAL  DEFAULT    23 b
    53: 0804a020     4 OBJECT  LOCAL  DEFAULT    23 a.1589
    81: 0804a02c     4 OBJECT  GLOBAL DEFAULT    24 c
...
```

变量 A 用 const 修饰，表示 A 是只读的，不可修改，它被分配的地址是 0x8048540，从 readelf 的输出可以看到这个地址位于.rodata 段：

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
...
  [13] .text             PROGBITS        08048360 000360 0001bc 00  AX  0   0 16
...
  [15] .rodata           PROGBITS        08048538 000538 00001c 00   A  0   0  4
...
  [23] .data             PROGBITS        0804a010 001010 000014 00  WA  0   0  4
  [24] .bss              NOBITS          0804a024 001024 00000c 00  WA  0   0  4
...
```

它在文件中的地址是 0x538~0x554，我们用 hexdump 命令看看这个段的内容：

```
$ hexdump -C a.out
...
00000530  5c fe ff ff 59 5b c9 c3  03 00 00 00 01 00 02 00  |\...Y[..........|
00000540  0a 00 00 00 48 65 6c 6c  6f 20 77 6f 72 6c 64 20  |....Hello world |
00000550  25 64 0a 00 00 00 00 00  00 00 00 00 00 00 00 00  |%d..............|
...
```

其中 0x540 地址处的 0a 00 00 00 就是变量 A。我们还看到程序中的字符串字面值"Hello world %d\n"分配在.rodata 段的末尾，在第 4 节 “字符串”说过字符串字面值是只读的，相当于在全局作用域定义了一个 const 数组：

```
const char helloworld[] = {'H', 'e', 'l', 'l', 'o', ' ',
                           'w', 'o', 'r', 'l', 'd', ' ', '%', 'd', '\n', '\0'};
```

程序加载运行时，.rodata 段和.text 段通常合并到一个 Segment 中，操作系统将这个 Segment 的页面只读保护起来，防止意外的改写。这一点从 readelf 的输出也可以看出来：

```
Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
```

```
02        .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn
 .rel.plt .init .plt .text .fini .rodata .eh_frame
    03        .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
    04        .dynamic
    05        .note.ABI-tag
    06
    07        .ctors .dtors .jcr .dynamic .got
```

注意，像 A 这种 const 变量在定义时必须初始化。因为只有初始化时才有机会给它一个值，一旦定义之后就不能再改写了，也就是不能再赋值了。

从上面 readelf 的输出可以看到.data 段从地址 0x804a010 开始，长度是 0x14，也就是到地址 0x804a024 结束。在.data 段中有三个变量，a，b 和 a.1589。

a 是一个 GLOBAL 的符号，而 b 被 static 关键字修饰了，导致它成为一个 LOCAL 的符号，所以 static 在这里的作用是声明 b 这个符号为 LOCAL 的，不被链接器处理，如果把多个目标文件链接在一起，LOCAL 的符号只能在某一个目标文件中定义和使用，而不能定义在一个目标文件中却在另一个目标文件中使用。一个函数定义前面也可以用 static 修饰，表示这个函数名符号是 LOCAL 的。

还有一个 a.1589 是什么呢？它就是 main 函数中的 static int a。函数中的 static 变量不同于以前我们讲的局部变量，它并不是在调用函数时分配，在函数返回时释放，而是像全局变量一样静态分配，所以用“static”（静态）这个词。另一方面，函数中的 static 变量的作用域和以前讲的局部变量一样，只在函数中起作用，比如 main 函数中的 a 这个变量名只在 main 函数中起作用，在别的函数中说变量 a 就不是指它了，所以编译器给它的符号名加了一个后缀，变成 a.1589，以便和全局变量 a 以及其它函数的变量 a 区分开。

.bss 段从地址 0x804a024 开始（紧挨着.data 段），长度为 0xc，也就是到地址 0x804a030 结束。变量 c 位于这个段。从上面的 readelf 输出可以看到，.data 和.bss 在加载时合并到一个 Segment 中，这个 Segment 是可读可写的。.bss 段和.data 段的不同之处在于，.bss 段在文件中不占存储空间，在加载时这个段用 0 填充。全局变量如果不初始化则初值为 0，同理可以推断，static 变量（不管是函数里的还是函数外的）如果不初始化则初值也是 0，也分配在.bss 段。

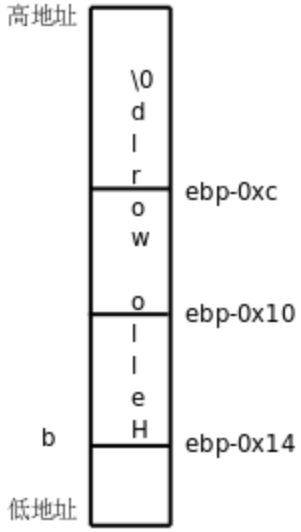
现在还剩下函数中的 b 和 c 这两个变量没有分析。b 是数组也一样，也是分配在栈上的，我们看 main 函数的反汇编代码：

```
$ objdump -dS a.out
...
            char b[]="Hello world";
 8048430:        c7 45 ec 48 65 6c 6c      movl   $0x6c6c6548,-0x14(%ebp)
 8048437:        c7 45 f0 6f 20 77 6f      movl   $0x6f77206f,-0x10(%ebp)
 804843e:        c7 45 f4 72 6c 64 00      movl   $0x646c72,-0xc(%ebp)
            register int c = 50;
```

```
8048445:        b8 32 00 00 00          mov     $0x32,%eax

        printf("Hello world %d\n", c);
804844a:        89 44 24 04             mov     %eax,0x4(%esp)
804844e:        c7 04 24 44 85 04 08    movl    $0x8048544,(%esp)
8048455:        e8 e6 fe ff ff          call    8048340 <printf@plt>
...
```

可见，给 b 初始化用的这个字符串"Hello world"并没有分配在.rodata 段，而是直接写在指令里了，通过三条 movl 指令把 12 个字节写到栈上，这就是 b 的存储空间，如下图所示。



数组的存储布局

变量 c 并没有在栈上分配存储空间，而是直接存在 eax 寄存器里，后面调用 printf 也是直接从 eax 寄存器里取出 c 的值当参数压栈，这就是 register 关键字的作用，指示编译器尽可能分配一个寄存器来存储这个变量。我们还看到调用 printf 时对于"Hello world %d\n"这个参数压栈的是它在.rodata 段中的首地址，而不是把整个字符串压栈，字符串在使用时可以看作数组名，如果做右值则表示数组首元素的地址（或者说指向数组首元素的指针），我们以后讲指针还要继续讨论这个问题。

例子：

例 1：
程序 1：

```
#include<stdio.h>
long a[10];
int main(void)
{
        return 0;
}
```

程序 2：

```
#include<stdio.h>
long a[10]={1,2,3,5,6,7,8,9,10};
int main(void)
{
        return 0;
}
```

说明：

bss 段（未手动初始化的数据）并不给该段的数据分配空间，只是记录数据所需空间的大小

data（已手动初始化的数据）段则为数据分配空间，数据保存在目标文件中。

标识符的链接属性（Linkage）有三种：

外部链接（External Linkage），如果最终的可执行文件由多个程序文件链接而成，一个标识符在任意程序文件中即使声明多次也都代表同一个变量或函数，则这个标识符具有 External Linkage。具有 External Linkage 的标识符编译后在符号表中是 GLOBAL 的符号。例如上例中 main 函数外面的 a 和 c，main 和 printf 也算。

内部链接（Internal Linkage），如果一个标识符在某个程序文件中即使声明多次也都代表同一个变量或函数，则这个标识符具有 Internal Linkage。例如上例中 main 函数外面的 b。如果有另一个 foo.c 程序和 main.c 链接在一起，在 foo.c 中也声明一个 static int b;，则那个 b 和这个 b 不代表同一个变量。具有 Internal Linkage 的标识符编译后在符号表中是 LOCAL 的符号，但 main 函数里面那个 a 不能算 Internal Linkage 的，因为即使在同一个程序文件中，在不同的函数中声明多次，也不代表同一个变量。

无链接（No Linkage）。除以上情况之外的标识符都属于 No Linkage 的，例如函数的局部变量，以及不表示变量和函数的其它标识符。

变量的生存期（Storage Duration，或者 Lifetime）分为以下几类：

静态生存期（Static Storage Duration），具有外部或内部链接属性，或者被 static 修饰的变量，在程序开始执行时分配和初始化一次，此后便一直存在直到程序结束。这种变量通常位于.rodata，.data 或.bss 段，例如上例中 main 函数外的 A，a，b，c，以及 main 函数里的 a。

自动生存期（Automatic Storage Duration），链接属性为无链接并且没有被 static 修饰的变量，这种变量在进入块作用域时在栈上或寄存器中分配，在退出块作用域时释放。例如上例中 main 函数里的 b 和 c。

动态分配生存期（Allocated Storage Duration），以后会讲到调用 malloc 函数在进程的堆空间中分配内存，调用 free 函数可以释放这种存储空间。

全局变量、局部变量和作用域

一、局部变量

“局部”有两层含义：

1、一个函数中定义的变量不能被另一个函数使用。

例 1：

```
#include<stdio.h>
void print_time(void)
{
    printf("%d:%d\n", hour, minute);
}

int main(void)
{
    int hour = 23, minute = 59;
    print_time(hour, minute);
    return 0;
}
```

2、每次调用函数时局部变量都表示不同的存储空间。

例 2：

```
#include<stdio.h>
void print_time(int hour, int minute)
{
    hour = 20;
    minute = 50;
    printf("%d:%d\n", hour, minute);
}

int main(void)
{
    int hour = 23, minute = 59;
    print_time(hour, minute);
    printf("%d:%d\n", hour, minute);
    return 0;
}
```

二、全局变量

全局变量在任何函数中都可以访问

例 1：

```
#include <stdio.h>
int hour = 23, minute = 59;
```

```
void print_time(void)
{
	printf("%d:%d in print_time\n", hour, minute);
}

int main(void)
{
	print_time();
	printf("%d:%d in main\n", hour, minute);
	return 0;
}
```

正因为全局变量在任何函数中都可以访问，所以在程序运行过程中全局变量被读写的顺序从源代码中是看不出来的，源代码的书写顺序并不能反映函数的调用顺序。程序出现了Bug往往就是因为在某个不起眼的地方对全局变量的读写顺序不正确，如果代码规模很大，这种错误是很难找到的。而对局部变量的访问不仅局限在一个函数内部，而且局限在一次函数调用之中，从函数的源代码很容易看出访问的先后顺序是怎样的，所以比较容易找到Bug。因此，虽然全局变量用起来很方便，但一定要慎用，能用函数传参代替的就不要用全局变量。

例2：

```
#include <stdio.h>
int hour = 23, minute = 59;

void print_time(void)
{
	hour = 20;
	minute = 50;
	printf("%d:%d in print_time\n", hour, minute);
}

int main(void)
{
	print_time();
	printf("%d:%d in main\n", hour, minute);
	return 0;
}
```

到目前为止我们在初始化一个变量时都是用常量，其实也可以用表达式，但要注意一点：**局部变量可以用类型相符的任意表达式来初始化，而全局变量只能用常量表达式初始化**。例如，全局变量pi这样初始化是合法的：

```
double pi = 3.14 + 0.0016;
```

但这样初始化是不合法的：

```
double pi = acos(-1.0);
```

然而局部变量这样初始化却是可以的。程序开始运行时要用适当的值来初始化全局变量，所以初始值必须保存在编译生成的可执行文件中，因此初始值在编译时就要计算出来，然而上面第二种初始化的值必须在程序运行时调用 acos 函数才能得到，所以不能用来初始化全局变量。请注意区分编译时和运行时这两个概念。

定义和声明

如果用 static 关键字修饰一个函数声明，则表示该标识符具有 Internal Linkage，例如有以下两个程序文件：

```
/* foo.c */
static void foo(void) {}
/* main.c */
void foo(void);
int main(void) { foo(); return 0; }
```

编译链接在一起会出错：

```
$ gcc foo.c main.c
/tmp/ccRC2Yjn.o: In function `main':
main.c:(.text+0x12): undefined reference to `foo'
collect2: ld returned 1 exit status
```

虽然在 foo.c 中定义了函数 foo，但这个函数只具有 Internal Linkage，只有在 foo.c 中多次声明才表示同一个函数，而在 main.c 中声明就不表示它了。如果把 foo.c 编译成目标文件，函数名 foo 在其中是一个 LOCAL 的符号，不参与链接过程，所以在链接时，main.c 中用到一个 External Linkage 的 foo 函数，链接器却找不到它的定义在哪儿，无法确定它的地址，也就无法做符号解析，只好报错。凡是被多次声明的变量或函数，必须有且只有一个声明是定义，如果有多个定义，或者一个定义都没有，链接器就无法完成链接。

volatile 限定符

现在探讨一下编译器优化会对生成的指令产生什么影响，在此基础上介绍 C 语言的 volatile 限定符。看下面的例子。

```
/* artificial device registers */
unsigned char recv;
unsigned char send;

/* memory buffer */
unsigned char buf[3];

int main(void)
{
    buf[0] = recv;
    buf[1] = recv;
    buf[2] = recv;
    send = ~buf[0];
```

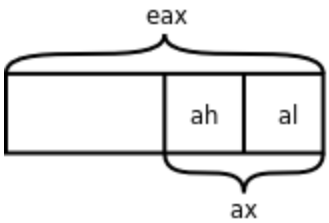
```
    send = ~buf[1];
    send = ~buf[2];

    return 0;
}
```

我们用 recv 和 send 这两个全局变量来模拟设备寄存器。假设某种平台采用内存映射 I/O，串口发送寄存器和串口接收寄存器位于固定的内存地址，而 recv 和 send 这两个全局变量也有固定的内存地址，所以在这个例子中把它们假想成串口接收寄存器和串口发送寄存器。在 main 函数中，首先从串口接收三个字节存到 buf 中，然后把这三个字节取反，依次从串口发送出去。我们查看这段代码的反汇编结果：

```
buf[0] = recv;
80483a2:        0f b6 05 19 a0 04 08     movzbl 0x804a019,%eax
80483a9:        a2 1a a0 04 08            mov    %al,0x804a01a
        buf[1] = recv;
80483ae:        0f b6 05 19 a0 04 08     movzbl 0x804a019,%eax
80483b5:        a2 1b a0 04 08            mov    %al,0x804a01b
        buf[2] = recv;
80483ba:        0f b6 05 19 a0 04 08     movzbl 0x804a019,%eax
80483c1:        a2 1c a0 04 08            mov    %al,0x804a01c
        send = ~buf[0];
80483c6:        0f b6 05 1a a0 04 08     movzbl 0x804a01a,%eax
80483cd:        f7 d0                     not    %eax
80483cf:        a2 18 a0 04 08            mov    %al,0x804a018
        send = ~buf[1];
80483d4:        0f b6 05 1b a0 04 08     movzbl 0x804a01b,%eax
80483db:        f7 d0                     not    %eax
80483dd:        a2 18 a0 04 08            mov    %al,0x804a018
        send = ~buf[2];
80483e2:        0f b6 05 1c a0 04 08     movzbl 0x804a01c,%eax
80483e9:        f7 d0                     not    %eax
80483eb:        a2 18 a0 04 08            mov    %al,0x804a018
```

movz 指令把字长较短的值存到字长较长的存储单元中，存储单元的高位用 0 填充。该指令可以有 b（byte）、w（word）、l（long）三种后缀，分别表示单字节、两字节和四字节。比如 movzbl 0x804a019,%eax 表示把地址0x804a019处的一个字节存到 eax 寄存器中，而 eax 寄存器是四字节的，高三字节用0填充，而下一条指令 mov %al,0x804a01a 中的 al 寄存器正是 eax 寄存器的低字节，把这个字节存到地址0x804a01a 处的一个字节中。可以用不同的名字单独访问 x86寄存器的低8位、次低8位、低16位或者完整的32位，以 eax 为例，al 表示低8位，ah 表示次低8位，ax 表示低16位，如下图所示。



eax 寄存器

但如果指定优化选项-O 编译，反汇编的结果就不一样了：

```
$ gcc main.c -g -O
$ objdump -dS a.out|less
...
        buf[0] = recv;
 80483ae:       0f b6 05 19 a0 04 08    movzbl 0x804a019,%eax
 80483b5:       a2 1a a0 04 08          mov    %al,0x804a01a
        buf[1] = recv;
 80483ba:       a2 1b a0 04 08          mov    %al,0x804a01b
        buf[2] = recv;
 80483bf:       a2 1c a0 04 08          mov    %al,0x804a01c
        send = ~buf[0];
        send = ~buf[1];
        send = ~buf[2];
 80483c4:       f7 d0                   not    %eax
 80483c6:       a2 18 a0 04 08          mov    %al,0x804a018
...
```

前三条语句从串口接收三个字节，而编译生成的指令显然不符合我们的意图：只有第一条语句从内存地址0x804a019读一个字节到寄存器 eax 中，然后从寄存器 al 保存到 buf[0]，后两条语句就不再从内存地址0x804a019读取，而是直接把寄存器 al 的值保存到 buf[1]和 buf[2]。后三条语句把 buf 中的三个字节取反再发送到串口，编译生成的指令也不符合我们的意图：只有最后一条语句把 eax 的值取反写到内存地址0x804a018了，前两条语句形同虚设，根本不生成指令。

为什么编译器优化的结果会错呢？因为编译器并不知道0x804a018和0x804a019是设备寄存器的地址，把它们当成普通的内存单元了。如果是普通的内存单元，只要程序不去改写它，它就不会变，可以先把内存单元里的值读到寄存器缓存起来，以后每次用到这个值就直接从寄存器读取，这样效率更高，我们知道读寄存器远比读内存要快。另一方面，如果对一个普通的内存单元连续做三次写操作，只有最后一次的值会保存到内存单元中，所以前两次写操作是多余的，可以优化掉。访问设备寄存器的代码这样优化就错了，因为设备寄存器往往具有以下特性：

设备寄存器中的数据不需要改写就可以自己发生变化，每次读上来的值都可能不一样。

连续多次向设备寄存器中写数据并不是在做无用功，而是有特殊意义的。

用优化选项编译生成的指令明显效率更高，但使用不当会出错，为了避免编译器自作聪明，把不该优化的也优化了，程序员应该明确告诉编译器哪些内存单元的访问是不能优化的，在 C 语言中可以用 volatile 限定符修饰变量，就是告诉编译器，即使在编译时指定了优化选项，每次读这个变量仍然要老老实实从内存读取，每次写这个变量也仍然要老老实实写回内存，不能省略任何步骤。我们把代码的开头几行改成：

```
/* artificial device registers */
volatile unsigned char recv;
volatile unsigned char send;
```

然后指定优化选项-O 编译，查看反汇编的结果：

```
        buf[0] = recv;
80483a2:        0f b6 05 19 a0 04 08     movzbl 0x804a019,%eax
80483a9:        a2 1a a0 04 08            mov    %al,0x804a01a
        buf[1] = recv;
80483ae:        0f b6 15 19 a0 04 08     movzbl 0x804a019,%edx
80483b5:        88 15 1b a0 04 08         mov    %dl,0x804a01b
        buf[2] = recv;
80483bb:        0f b6 0d 19 a0 04 08     movzbl 0x804a019,%ecx
80483c2:        88 0d 1c a0 04 08         mov    %cl,0x804a01c
        send = ~buf[0];
80483c8:        f7 d0                     not    %eax
80483ca:        a2 18 a0 04 08            mov    %al,0x804a018
        send = ~buf[1];
80483cf:        f7 d2                     not    %edx
80483d1:        88 15 18 a0 04 08         mov    %dl,0x804a018
        send = ~buf[2];
80483d7:        f7 d1                     not    %ecx
80483d9:        88 0d 18 a0 04 08         mov    %cl,0x804a018
```

确实每次读 recv 都从内存地址0x804a019读取，每次写 send 也都写到内存地址0x804a018了。值得注意的是，每次写 send 并不需要取出 buf 中的值，而是取出先前缓存在寄存器 eax、edx、ecx 中的值，做取反运算然后写下去，这是因为 buf 并没有用 volatile 限定，读者可以试着在 buf 的定义前面也加上 volatile，再优化编译，再查看反汇编的结果。

gcc 的编译优化选项有-O0、-O、-O1、-O2、-O3、-Os 几种。-O0表示不优化，这是缺省的选项。-O1、-O2和-O3这几个选项一个比一个优化得更多，编译时间也更长。-O 和-O1 相同。-Os 表示为缩小目标文件的尺寸而优化。

从上面的例子还可以看到，如果在编译时指定了优化选项，源代码和生成指令的次序可能无法对应，甚至有些源代码可能不对应任何指令，被彻底优化掉了。这一点在用 gdb 做源码级调试时尤其需要注意（做指令级调试没关系），在为调试而编译时不要指定优化选项，否则可能无法一步步跟踪源代码的执行过程。