

高级语言数据类型与 x86 汇编内存访问——以C语言为例

实验背景

程序就是算法加数据结构，数据的存储和访问在程序中不可谓不重要。高级语言大都提供了多种基础数据类型以及构造复杂数据类型的方法，这些数据存储在不同的区域中，结构不一，程序究竟是如何对它们进行访问的？相信经历本次探究后会对高级语言有一个更加清楚地认识。

实验工具与方法

实验工具

- Ubuntu 22.04 + vscode + x86-64 gcc-11.3。
- [compiler explore](#)(x86-64 gcc-11.3)。

实验方法

- 通过定义各种不同的变量并使用它们进行一些简单或嵌套的计算，观察汇编代码

实验过程

1. 局部变量、全局变量、静态变量

初学编程时都学过局部变量存储在栈中，全局变量和静态变量存储于静态数据区，但是对于程序究竟如何访问它们从高级语言层次是难以看出来的。通过汇编代码我们可以一窥其究竟。

1.1 首先是变量的简单初始化。

```
1  int global = 1;
2  int global2;
3  int* global3;
4
5  int main()
6  {
7      static int si=1;
8      static int ss = 0x8888;
9
10     int local=2, local2=3;
11     int* local3 ;
12     char sl[8] = "123456";
13
14     local = si;
15     local = ss;
16     // local3 = local;
17
18     return 0;
19 }
```

```
1  global:
2      .long    1
3  global2:
4      .zero    4
5  global3:
6      .zero    4
7  main:
8      push    ebp
9      mov     ebp, esp
10     sub     esp, 16
11     mov     DWORD PTR [ebp-4], 2
12     mov     DWORD PTR [ebp-8], 3
13     mov     DWORD PTR [ebp-16], 875770417
14     mov     DWORD PTR [ebp-12], 13877
15     mov     eax, DWORD PTR si.1
16     mov     DWORD PTR [ebp-4], eax
17     mov     eax, DWORD PTR ss.0
18     mov     DWORD PTR [ebp-4], eax
19     mov     eax, 0
20     leave
21     ret
22 si.1:
23     .long    1
24 ss.0:
25     .long    34952
```

从左右着色的对比我们就可以看出来，全局变量在程序外有特定的存储区域，静态变量一样（实际上在网站上原本未显示出来，在经过对局部变量赋值使用后才出现），这两者在程序中都有固定的存储地址，读写时只需要直接访问对应地址即可。

而对于局部变量，通过栈进行管理存储，可见程序开头 `sub esp, 16` 指令为局部变量分配了空间，并在初始化过程中将对应的值存储到对应的栈空间中。当然，仔细的话，我们可以注意到我们使用了两个整形一个指针以及一个字符数组，应该有 $2*4+4+8 = 20$ （（网站中64位未编译出栈的分配指令，注意32位系统指针为4字节））个字节栈空间，实际上只分配了16个空间，通过其他

变量的赋值我们可以明显察觉到指针变量并未分配栈空间。

<pre>9 10 int local=2, local2=3; 11 int* local3,q,w,e,r,t,y,u; 12 char sl[4] = "123"; 13</pre>	<pre>8 9 10 11</pre>	<pre>push ebp mov ebp, esp sub esp, 16 mov DWORD PTR [ebp-4], 2</pre>
--	----------------------	---

通过进一步调整，可以看到即使多出很多未使用的指针，栈空间也不会增加指针的空间，而我们将字符数组缩小四个字节后，栈空间却没有变小。

<pre>10 int local=2, local2=3; 11 int* local3,q,w,e,r,t,y,u; 12 char sl[8] = "123"; 13 local3 = &local2;</pre>	<pre>8 9 10 11 12</pre>	<pre>push ebp mov ebp, esp sub esp, 32 mov DWORD PTR [ebp-4], 2 mov DWORD PTR [ebp-12], 3</pre>
--	-------------------------	---

再次调整，为一个指针赋值，栈空间增加了16字节，即可以看出栈的空间分配以一定字节数为单位。即使不足的字节数也会按这个字节数压栈。（在本地的gdb调试中同样按16字节压栈，但是鉴于机器种类单一，无法断定16字节是否绝对）

因此，对于局部变量可以总结如下，未赋值的野指针并不分配栈空间，其余变量由函数在栈中统一分配空间与管理，栈空间会按一定的字节数进行分配而非需要多少给多少。

1.2 函数参数的传递

函数参数的传递和使用同样依赖于栈，但是实际使用起来与原本所想也是有诸多不同。

<

```

Register group: general
rax      0x2      2
rbx      0x0      0
rcx      0x3      3
rdx      0x7fffffffdb74  140737488345972
rsi      0x3      3
rdi      0x2      2
rbp      0x7fffffffdb90  0x7fffffffdb90
rsp      0x7fffffffdb70  0x7fffffffdb70
r8       0x7ffff7f9cf10  140737353731856

0x5555555519d <main+48>    lea    rax,[rbp-0x1c]
0x555555551a1 <main+52>    mov    QWORD PTR [rbp-0x10],rax
B+ 0x555555551a5 <main+56>    mov    eax,DWORD PTR [rbp-0x1c]
0x555555551a8 <main+59>    mov    rdx,QWORD PTR [rbp-0x10]
0x555555551ac <main+63>    mov    ecx,DWORD PTR [rbp-0x18]
0x555555551af <main+66>    mov    esi,ecx
0x555555551b1 <main+68>    mov    edi,eax
> 0x555555551b3 <main+70>    call   0x55555555149 <calcu>

```

```

Register group: general
rax      0x2      2
rbx      0x0      0
rcx      0x3      3
rdx      0x7fffffffdb74  140737488345972
rsi      0x3      3
rdi      0x2      2
rbp      0x7fffffffdb90  0x7fffffffdb90
rsp      0x7fffffffdb60  0x7fffffffdb60
r8       0x7ffff7f9cf10  140737353731856

0x55555555149 <calcu>    endbr64
0x5555555514d <calcu+4>    push   rbp
> 0x5555555514e <calcu+5>    mov    rbp,rsi
0x55555555151 <calcu+8>    mov    DWORD PTR [rbp-0x4],edi
0x55555555154 <calcu+11>   mov    DWORD PTR [rbp-0x8],esi
0x55555555157 <calcu+14>   mov    QWORD PTR [rbp-0x10],rdx
0x5555555515b <calcu+18>   mov    edx,DWORD PTR [rbp-0x4]
0x5555555515e <calcu+21>   mov    eax,DWORD PTR [rbp-0x8]

```

```

Register group: general
rax      0x2      2
rbx      0x0      0
rcx      0x3      3
rdx      0x7fffffffdb74  140737488345972
rsi      0x3      3
rdi      0x2      2
rbp      0x7fffffffdb60  0x7fffffffdb60
rsp      0x7fffffffdb60  0x7fffffffdb60
r8       0x7ffff7f9cf10  140737353731856

0x55555555149 <calcu>    endbr64
0x5555555514d <calcu+4>    push   rbp
0x5555555514e <calcu+5>    mov    rbp,rsi
0x55555555151 <calcu+8>    mov    DWORD PTR [rbp-0x4],edi
0x55555555154 <calcu+11>   mov    DWORD PTR [rbp-0x8],esi
0x55555555157 <calcu+14>   mov    QWORD PTR [rbp-0x10],rdx
> 0x5555555515b <calcu+18>   mov    edx,DWORD PTR [rbp-0x4]
0x5555555515e <calcu+21>   mov    eax,DWORD PTR [rbp-0x8]
0x55555555161 <calcu+24>   add    edx,eax

```

在函数调用数据过程中使用 `rbp` 作为栈的基地址，但是另一方面 `rsp` 的值却和 `rbp` 相同，而非指向栈顶。为了继续深入，我们为函数添加局部变量以及添加另一个函数，并调试。

```
4 int f(int a){
5     return a;
6 }
7 int calcu(int a,int b,int* mul)
8 {
9     int res = f(a)+b+*mul;
10    return res;
11 }
12 int main()
13 {
14     int a=1,b=2,*mul=&a;
15     calcu(a,b,mul);
16 }
```

gdb

Register group: general

rax	0x2	2
rbx	0x0	0
rcx	0x3	3
rdx	0x7fffffffdb74	140737488345972
rsi	0x3	3
rdi	0x2	2
rbp	0x7fffffffdb30	0x7fffffffdb30
rsp	0x7fffffffdb30	0x7fffffffdb30

```
0x55555555140 <frame_dummy> endbr64
0x55555555144 <frame_dummy+4> jmp 0x5555555550c0 <register_tm_clones>
0x55555555149 <f> endbr64
0x5555555514d <f+4> push rbp
0x5555555514e <f+5> mov rbp, rsp
> 0x55555555151 <f+8> mov DWORD PTR [rbp-0x4], edi
0x55555555154 <f+11> mov eax, DWORD PTR [rbp-0x4]
0x55555555157 <f+14> pop rbp
0x55555555158 <f+15> ret
0x55555555159 <calcu> endbr64
0x5555555515d <calcu+4> push rbp
0x5555555515e <calcu+5> mov rbp, rsp
0x55555555161 <calcu+8> sub rsp, 0x20
0x55555555165 <calcu+12> mov DWORD PTR [rbp-0x14], edi
0x55555555168 <calcu+15> mov DWORD PTR [rbp-0x18], esi
0x5555555516b <calcu+18> mov QWORD PTR [rbp-0x20], rdx
0x5555555516f <calcu+22> mov eax, DWORD PTR [rbp-0x14]
0x55555555172 <calcu+25> mov edi, eax
0x55555555174 <calcu+27> call 0x55555555149 <f>
```

重新编译调试后，可以发现发现函数栈是根据整个函数的变量情况在函数开头进行统一分配的，而如果函数内部不存在另一个调用，那么对于这最后一个栈帧，`rsp` 并不需要指向顶部为下一个栈帧准备，那么编译器也不需要增加指令去操作 `rsp`。

2. 数组与结构体

数组和结构体可将多个简单的数据联系为一个整体，数组可同时构造管理多个同类数据，结构体可以联系不同种类的数据。它们并不能单独算作一个数据类型，也不会超出“局部，全局，静态”的限制，但是由于它们占用比普通类型更多的空间且需要支持“内部成员访问”，对它们的内存访问也会相对稍复杂。

2.1 简单的一维和二维数组

介绍到数组，我们就需要引入新的存储空间“堆”，用以存储在程序运行过程中通过 `malloc` 申请空间确定下来的数组（当然实际上也可以只申请单个数据的空间）。因此该步探究我们主要有三个对象，局部数组，全局数组，`malloc` 数组。

首先依旧是数组的声明和初始化。

```
1 #include <stdlib.h>
2 int query(int array[100], int i);
3 int query2(int array[10][10], int i, int j);
4
5 int global[100] = {9,8};
6 int global2[10][10];
7 int main()
8 {
9     int local=2, local2=3, local4[100]={1,2}, local5[10][10];
10    int* local3 =(int*)malloc(sizeof(int)*100);
11    int **local6=(int**)malloc(sizeof(int)*10);
12    for(local = 0; local<10; local++){
13        local6[local] = (int*)malloc(sizeof(int)*10);
14    }
15
16    int i;
17    for(i=0; i<100; i++){
18        local4[i]=i;
19        local3[i]=i;
20        global[i]=i;
21    }
22    local3[50] = 6;
23
24    int j;
25    for(i = 0; i<10; i++){
26        for(j = 0; j<10; j++){
27            local5[i][j] = i;
28            local6[i][j] = i;
29            global2[i][j] = i;
30        }
31    }
32    return 0;
33 }
34
35 int query(int array[100], int i)
36 {
37     return array[i];
38 }
39
40 int query2(int array[10][10], int i, int j)
```

```
1 global:
2     .long 9
3     .long 8
4     .zero 392
5 global2:
6     .zero 400
7 main:
8     push rbp
9     mov rbp, rsp
10    push rbx
11    sub rsp, 840
12    mov DWORD PTR [rbp-20], 2
13    mov DWORD PTR [rbp-32], 3
14    lea rdx, [rbp-448]
15    mov eax, 0
16    mov ecx, 50
17    mov rdi, rdx
18    rep stosq
19    mov DWORD PTR [rbp-448], 1
20    mov DWORD PTR [rbp-444], 2
21    mov edi, 400
22    call malloc
23    mov QWORD PTR [rbp-40], rax
24    mov edi, 80
25    call malloc
26    mov QWORD PTR [rbp-48], rax
27    mov DWORD PTR [rbp-20], 0
28    jmp .L2
29 .L3:
30    mov eax, DWORD PTR [rbp-20]
31    cdqe
32    lea rdx, [0+rax*8]
33    mov rax, QWORD PTR [rbp-48]
34    lea rbx, [rdx+rax]
35    mov edi, 40
36    call malloc
37    mov QWORD PTR [rbp-40], rax
```

对于全局变量，数组的表现并没有改变多少，存储于静态区域，将给定的初始化数据赋给内存后，直接将其余所需的内存初始化为0。

而对于局部变量，首先同样是初始化栈的大小。但是程序对于局部变量并不会默认初始化，我们可以看到对于变量 `local5` 没有任何操作，而对于变量 `local4` 先是通过循环将所有给定空间初始化为0，然后将给定的值一一赋给对应内存。即使我们将指定初始化的数据再次增加，循环赋0的操作不会减少。

而对于 `malloc` 数组，虽然数据并不存储于栈中，但是数组头的指针是存放于栈中的，即通过 `malloc` 得到的首地址会存储于栈中。每次 `malloc` 得到的地址是连续的，但是连续多次 `malloc` 得到的地址是不一定连续的。具体地说，对于 `local5` 来说，它在栈中是连续存储的，但是对于 `local6`，栈中存储的二级指针指向的是一级指针组的首地址，然后各一级地址内存存储的地址是具体的数据所在区域，显而易见，`malloc` 所申请的二维数组连占用的内存多出了88字节（二级和一级共11个指针）。而它们访存的差别可从下面的探究看出。

一维数组访问

```
local6[local] = (int*)malloc(sizeof(int)*10);
}
int i;
for(i=0; i<100; i++){
    local4[i]=i;
    local3[i]=i;
    global[i]=i;
}
int j;
for(i = 0; i<10; i++){
    for(j = 0; j<10; j++){
        local5[i][j] = i;
        local6[i][j] = i;
        global2[i][j] = i;
    }
}
```

```
43 jmp .L4
44 .L5:
45 mov eax, DWORD PTR [rbp-24]
46 cdqe
47 mov edx, DWORD PTR [rbp-24]
48 mov DWORD PTR [rbp-448+rax*4], edx
49 mov eax, DWORD PTR [rbp-24]
50 cdqe
51 lea rdx, [0+rax*4]
52 mov rax, QWORD PTR [rbp-40]
53 add rdx, rax
54 mov eax, DWORD PTR [rbp-24]
55 mov DWORD PTR [rdx], eax
56 mov eax, DWORD PTR [rbp-24]
57 cdqe
58 mov edx, DWORD PTR [rbp-24]
59 mov DWORD PTR [global[0+rax*4]], edx
60 add DWORD PTR [rbp-24], 1
```

局部数组和全局数组的访问基本相同，先直接取数组索引并拓展位，然后通过偏移地址与基址可以得到很方便的得到具体的地址。

而 `malloc` 数组多出了从栈中取出数据真实存储地址的步骤，将取出的地址加上（索引*数据字节数）才得到具体地址。

二维数组

```
int **local6=(int**)malloc(sizeof(int)*10);
for(local = 0;local<10;local++){
    local6[local] = (int*)malloc(sizeof(int)*10);
}

int i;
for(i=0;i<100;i++){
    local4[i]=i;
    local3[i]=i;
    global[i]=i;
}

int j;
for(i = 0;i<10;i++){
    for(j = 0;j<10;j++){
        local5[i][j] = i;
        local6[i][j] = i;
        global2[i][j] = i;
    }
}

return 0;

int query(int array[100] ,int i)
{
    return array[i];
}

int query2(int array[10][10],int i,int j)
{
    return array[i][j];
}
```

```
70 mov     eax, DWORD PTR [rbp-28]
71 movsx   rcx, eax
72 mov     eax, DWORD PTR [rbp-24]
73 movsx   rdx, eax
74 mov     rax, rdx
75 sal     rax, 2
76 add     rax, rdx
77 add     rax, rax
78 lea     rdx, [rax+rcx]
79 mov     eax, DWORD PTR [rbp-24]
80 mov     DWORD PTR [rbp-848+rdx*4], eax
81 mov     eax, DWORD PTR [rbp-24]
82 cdq     rdx
83 lea     rdx, [0+rax*8]
84 mov     rax, QWORD PTR [rbp-48]
85 add     rax, rdx
86 mov     rdx, QWORD PTR [rax]
87 mov     eax, DWORD PTR [rbp-28]
88 cdq     rdx
89 sal     rax, 2
90 add     rdx, rax
91 mov     eax, DWORD PTR [rbp-24]
92 mov     DWORD PTR [rdx], eax
93 mov     eax, DWORD PTR [rbp-28]
94 movsx   rcx, eax
95 mov     eax, DWORD PTR [rbp-24]
96 movsx   rdx, eax
97 mov     rax, rdx
98 sal     rax, 2
99 add     rax, rdx
100 add    rax, rax
101 lea     rdx, [rax+rcx]
102 mov     eax, DWORD PTR [rbp-24]
103 mov     DWORD PTR global2[0+rdx*4], eax
```

二维局部数组和全局数组存储的空间连续，实际上可以当做一维数组进行访问（中间的移位和加法计算实际上是对乘法的等价运算，将高维数索引转化成低维数索引），访存方法也基本是按部就班计算索引，然后通过地址进行读写。

`malloc` 数组的基本思路是从栈中地址开始向下依次计算访问各级地址内存，直至计算出需要操作的数据对应的具体地址，多出几步访问内存的操作。

函数参数

虽然作为数组这三者的使用格式基本相同（`[]` or `*`）但这是编译器帮我们自动区分访问访问方式的结果，从上面的探究我们已经可以知道直接声明的数组（`a[]`）和通过指针和 `malloc` 申请的数组（`*a`）是不同。接下来我们将从作为参数时的表现看它们的不同。

```
int query(int array[100] ,int i);
int query_(int *array ,int i);
int query2(int array[10][10],int i,int j);
int query2_(int **array,int i,int j);
```


<pre> 21 local5[i]=1; 22 global[i]=i; 23 } 24 25 int j; 26 for(i = 0;i<10;i++){ 27 for(j = 0;j<10;j++){ 28 local5[i][j] = i; 29 local6[i][j] = i; 30 global2[i][j] = i; 31 } 32 } 33 34 local= query(local4,1); 35 local= query_(local3,1); 36 local= query_(local4,1); 37 38 local= query2(local5,1,5); 39 local= query2_(local6,1,5); 40 local= query2_(local5,1,5); 41 42 return 0; 43 } 44 45 int query(int array[100],int i) 46 { 47 return array[i]; 48 } 49 int query_(int *array ,int i) 50 { 51 return array[i]; 52 } 53 int query2(int array[10][10],int i,int j) 54 { 55 return array[i][j]; 56 } </pre>	<pre> 111 .lea .L9 112 lea rax, [rbp-448] 113 mov esi, 1 114 mov rdi, rax 115 call query 116 mov DWORD PTR [rbp-20], eax 117 mov rax, QWORD PTR [rbp-40] 118 mov esi, 1 119 mov rdi, rax 120 call query_ 121 mov DWORD PTR [rbp-20], eax 122 lea rax, [rbp-448] 123 mov esi, 1 124 mov rdi, rax 125 call query_ 126 mov DWORD PTR [rbp-20], eax 127 lea rax, [rbp-848] 128 mov edx, 5 129 mov esi, 1 130 mov rdi, rax 131 call query2 132 mov DWORD PTR [rbp-20], eax 133 mov rax, QWORD PTR [rbp-48] 134 mov edx, 5 135 mov esi, 1 136 mov rdi, rax 137 call query2_ 138 mov DWORD PTR [rbp-20], eax 139 lea rax, [rbp-848] 140 mov edx, 5 141 mov esi, 1 142 mov rdi, rax 143 call query2_ 144 mov DWORD PTR [rbp-20], eax </pre>
---	---

在函数调用过程中，我们可以看到，对于一维数组，两者并没有什么差别，通过编译器的调整，最终传给函数的地址都是真实的数据所在地址。但是对于二维数组，编译器无法调整两级的差别，这两种访存方式的差异体现了出来，编译器给错误的参数添加了蓝色波浪线。

```

int query(int array[100],int i)
{
    ... return array[i];
}

int query_(int *array ,int i)
{
    ... return array[i];
}

int query2(int array[10][10],int i,int j)
{
    ... return array[i][j];
}

int query2_(int **array,int i,int j)
{
    ... return array[i][j];
}

```

(汇编代码内容略)

通过函数栈得到了地址后，数组的访存也基本如所想的。得到具体数据地址的两个一维数组函数的动作相同，而参数本质不同两个二维数组函数（一个依旧是具体数据地址，另一个只是一级指针的地址）表现也不太相同，但也只是取地址所经历的操作有所区别。

2.2 结构体

结构体的内存访问其实和其余变量没什么差别，结构体只是方便我们操作一个整体，但是对于编译器来说它们依旧可视为一个个分开的数据，访问结构体内元素不过是为结构体整体的地址再添加一个偏移量而已。

结构体初始化时也和数组相同，先全部初始化为0再将给定值赋给对应内存。

```

int main()
{
    point localp;
    line local={1,2};
    int local;

    local = query(localp,globalp);
    local = queryl(local);
}

```

```

33      push    rbx
34      sub     rsp, 88
35      mov     QWORD PTR [rbp-96], 0
36      mov     QWORD PTR [rbp-88], 0
37      mov     QWORD PTR [rbp-80], 0
38      mov     QWORD PTR [rbp-72], 0
39      mov     DWORD PTR [rbp-64], 0
40      mov     DWORD PTR [rbp-96], 1
41      mov     DWORD PTR [rbp-92], 2

```

需要注意结构体的函数参数传递有所不同，结构体所占的空间比普通数据大，又并不能像数组一样仅使用首地址取代，因此传参数时容易出现寄存器不足的情况。

```

1  typedef struct point{
2      int x;
3      int y;
4      int index;
5      int value;
6  }point;
7
8  typedef struct line{
9      struct point points[2];
10     int num;
11 }line;

```

结构体

```

14 point globalp;
15 line globalc;
16
17 int query(point a,point b)
18 {
19     return a.x*b.x;
20 }
21 int queryl(line a)
22 {
23     return a.points[1].index-a.points[0].index;
24 }
25
26 int main()
27 {
28     point localp;
29     line local={1,2};
30     int local;
31
32     local = query(localp,globalp);
33     local = queryl(local);
34
35     return query(localp,localp);
36
37 }

```

函数体

源代码

```

38      mov     QWORD PTR [rbp-72], 0
39      mov     DWORD PTR [rbp-64], 0
40      mov     DWORD PTR [rbp-96], 1
41      mov     DWORD PTR [rbp-92], 2
42      mov     rax, QWORD PTR globalp[rip]
43      mov     rdx, QWORD PTR globalp[rip+8]
44      mov     rdi, QWORD PTR [rbp-48]
45      mov     rsi, QWORD PTR [rbp-40]
46      mov     rcx, rdx
47      mov     rdx, rax
48      call    query
49      mov     DWORD PTR [rbp-20], eax
50
51      sub     rsp, 8
52      sub     rsp, 40
53      mov     rax, rsp
54      mov     rcx, QWORD PTR [rbp-96]
55      mov     rbx, QWORD PTR [rbp-88]
56      mov     QWORD PTR [rax], rcx
57      mov     QWORD PTR [rax+8], rbx
58      mov     rcx, QWORD PTR [rbp-80]
59      mov     rbx, QWORD PTR [rbp-72]
60      mov     QWORD PTR [rax+16], rcx
61      mov     QWORD PTR [rax+24], rbx
62      mov     edx, DWORD PTR [rbp-64]
63      mov     DWORD PTR [rax+32], edx
64      call    queryl
65      add     rsp, 48
66      mov     DWORD PTR [rbp-20], eax

```

函数调用

通过 query 的调用代码，我们可以发现在结构体比较小时，可以用多个寄存器传递一个参数。而在 queryl 函数中，显然所有参数寄存器都用上也未必够，此时程序提前进行压栈，先将所有参数压入栈中再调用函数，完成了参数传递。

实验感想

本次探究到此也就结束了，当然还有许多没有涉及到的地方，诸如本次探究基本在64位下进行编译（除1.1忘记修改参数）未对32位进行一个探索，虽然感觉两者的差距应该并不是很大，毕竟寄存器的大小和数量对本次实验涉及的内容影响比较小。另外编写的程序实际上都比较简单，对于复杂情况可能会有所不及，不过看过各种汇编代码后，我们基本可以确认在无优化（O0）情况下数据的访问基本都是计算地址+实际读写的模式，区别只在于地址的计算复杂程度。当然在优化的情况下就复杂了.....由于编写的程序较为简单，编译器会删减没有使用的所有变量，即使使用主函数返回值也大都直接由编译器给出结果作为立即数进行返回。而通过其余函数可以看到部分优化为直接使用寄存器和复杂的地址表达式来得到结果。

本次实验与其说是探究，不如说是一次验证，初接触汇编，便使用它来验证曾经所学习的一些的C语言知识，应该说对于C语言数据结构以及内存管理有了更加清晰的认识，不再像以前一样一知半解。由于编译器版本原因，也不得不在ubuntu中使用gdb进行程序调试，增加了熟练度。对编译器的强大也有了了解，从前只当编译出来的程序需要存储大量的多余信息，内存访问全靠操作系统，现在发现即使是

动态变化的栈，在某一栈帧下，程序完全了解各变量的对应地址而不需要额外存储变量表，或者说编译器完全安排好了各变量所处的与指令所要操作的位置。

从最初的茫然不知道做什么到老师指点从寻址入手到最后逐渐确定这一主题，虽然“浪费”了不少时间，但是尝试编写“特殊”代码寻找探究点的过程在结束后也是别有风味，可能这就是探究吧。