数组和指针类型的分配和访问

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

数组的分配和访问

- **数组元素在内存的存放和访问**
  - **例如，定义一个具有4个元素的静态存储型 short 数据类型数组A，可以写成“static short A[4];”**
  - **第 i（0≤i≤3）个元素的地址计算公式为&A[0]+2*i。**
  - **假定数组A的首地址存放在EDX中，i 存放在ECX中，现要将A[i]取到AX中，则所用的汇编指令是什么？**

movw (%edx, %ecx, 2), %ax **比例因子是2！**

其中，ECX为变址（索引）寄存器，在循环体中增量，

数组的分配和访问

- **填写下表**

数组定义	数组名	数组元素类型	数组元素大小（B）	数组大小（B）	起始地址	元素 i 的地址
char S[10]	S	char				
char * SA[10]	SA	char *				
double D[10]	D	double				
double * DA[10]	DA	double *				

数组的分配和访问

- **填写下表**

数组定义	数组名	数组元素类型	数组元素大小（B）	数组大小（B）	起始地址	元素 i 的地址
char S[10]	S	char	1	10	&S[0]	&S[0]+i
char * SA[10]	SA	char *	4	40	&SA[0]	&SA[0]+4*i
double D[10]	D	double	8	80	&D[0]	&D[0]+8*i
double * DA[10]	DA	double *	4	40	&DA[0]	&DA[0]+4*i

数组元素在内存的存放和访问

- 分配在静态区的数组的初始化和访问

```
int buf[2] = {10, 20};
int main ( )
{
    int i, sum=0;
    for (i=0; i<2; i++)
        sum+=buf[i];
    return sum;
}
```

buf是在静态区分配的数组，链接后，buf在可执行目标文件的数据段中分配了空间

```
08049080 <buf> :
08049080 :  0A 00 00 00 14 00 00 00
```

此时，buf=&buf[0]=0x08049080

编译器通常将其先存放到寄存器(如EDX)中

假定 i 被分配在ECX中，sum被分配在EAX中，则

“sum+=buf[i];”和 i++ 可用什么指令实现？

addl buf(, %ecx, 4), %eax 或 addl 0(%edx , %ecx, 4), %eax

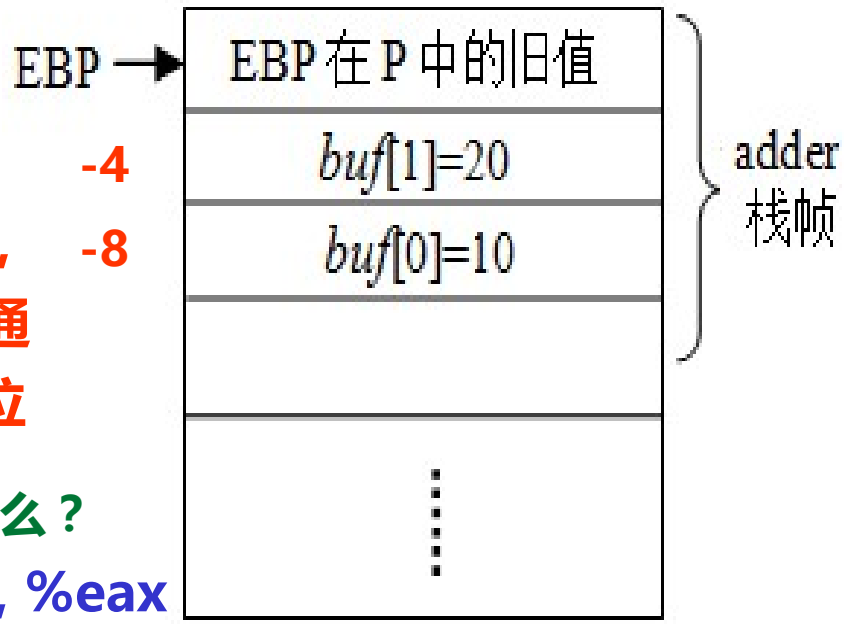
addl &1 , %ecx

数组元素在内存的存放和访问

- **auto型数组的初始化和访问**

```
int adder ( )
{
    int buf[2] = {10, 20};
    int i, sum=0;
    for (i=0; i<2; i++)
        sum+=buf[i];
    return sum;
}
```

分配在栈中，故数组首址通过EBP来定位



EDX、ECX各是什么？

```
addl (%edx, %ecx, 4), %eax
```

对buf进行初始化的指令是什么？

```
movl $10, -8(%ebp)    //buf[0]的地址为R[ebp]-8，将10赋给buf[0]
movl $20, -4(%ebp)    //buf[1]的地址为R[ebp]-4，将20赋给buf[1]
```

若buf首址在EDX中，则获得buf首址的对应指令是什么？

```
leal -8(%ebp), %edx  //buf[0]的地址为R[ebp]-8，将buf首址送EDX
```

数组元素在内存的存放和访问

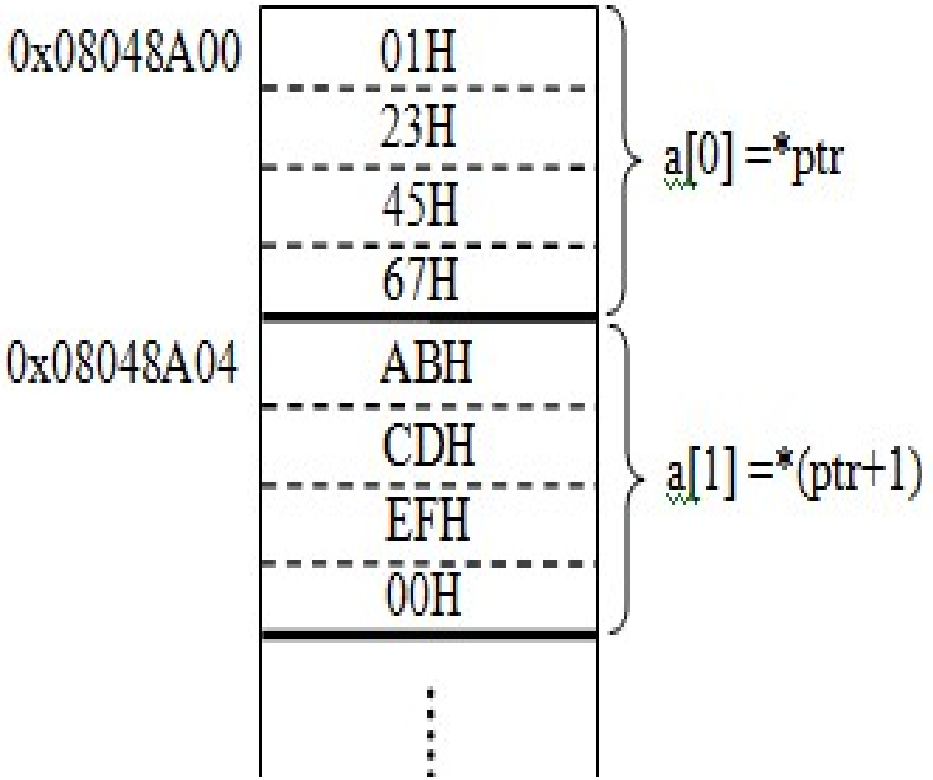
- **数组与指针**

✓ **在指针变量目标数据类型与数组类型相同的前提下，指针变量可以指向数组或数组中任意元素**

✓ **以下两个程序段功能完全相同，都是使ptr指向数组a的第0个元素a[0]。a的值就是其首地址，即a=&a[0]，因而a=ptr，从而有&a[i]=ptr+i=a+i以及a[i]=ptr[i]=*(ptr+i)=*(a+i)。**

（1）int a[10];
int *ptr=&a[0];

（2）int a[10], *ptr;
ptr=&a[0];



小端方式下a[0]=?,a[1]=?
a[0]=0x67452301, a[1]=0x0efcdab
数组首址0x8048A00在ptr中，ptr+i 并不是用0x8048A00加 i 得到，而是等于0x8048A00+4*i

数组元素在内存的存放和访问

- **数组与指针**

序号	表达式	类型	值的计算方式	汇编代码
1	A	int *		
2	A[0]	int		
3	A[i]	int		
4	&A[3]	int *		
5	&A[i]-A	int		
6	*(A+i)	int		
7	*(&A[0]+i-1)	int		
8	A+i	int *		

问题：

假定数组A的首址SA在ECX中，i在EDX中，表达式结果在EAX中，各表达式的计算方式以及汇编代码各是什么？

2、3、6和7对应汇编指令都需访存，指令中源操作数的寻址方式分别是“基址”、“基址加比例变址”、“基址加比例变址”和“基址加比例变址加位移”的方式，因为数组元素的类型为int型，故比例因子为4。

数组元素在内存的存放和访问

- **数组与指针** **假设A首址SA在ECX，i 在EDX，结果在EAX**

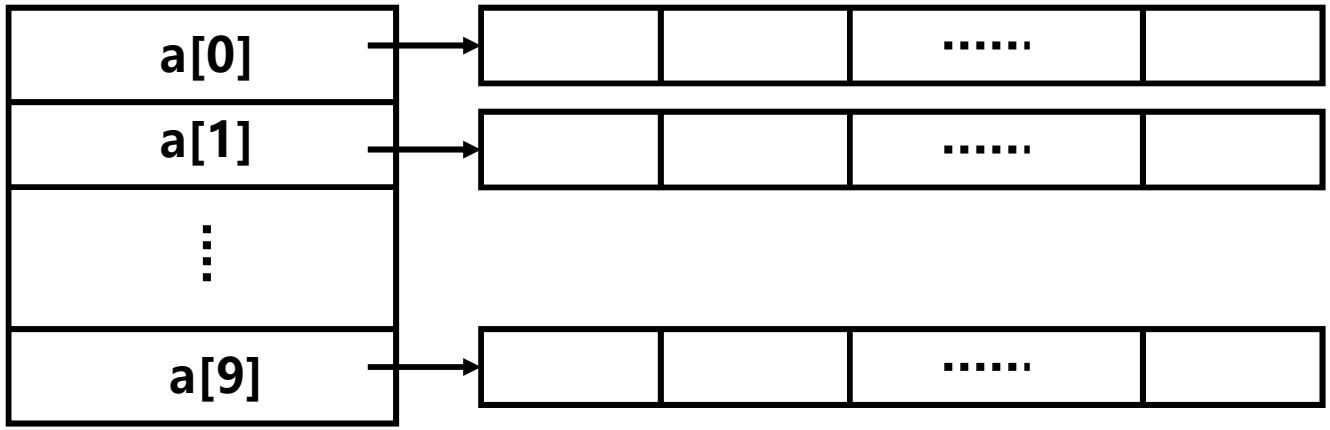
序号	表达式	类型	值的计算方式	汇编代码
1	A	int *	*SA*	leal (%ecx), %eax
2	A[0]	int	M[*SA*]	movl (%ecx), %eax
3	A[i]	int	M[*SA*+4**i*]	movl (%ecx, %edx, 4), %eax
4	&A[3]	int *	*SA*+12	leal 12(%ecx), %eax
5	&A[i]−A	int	(*SA*+4**i*−*SA*)/4=*i*	movl %edx, %eax
6	*(A+i)	int	M[*SA*+4**i*]	movl (%ecx, %edx, 4), %eax
7	*(&A[0]+i−1)	int	M[*SA*+4**i*−4]	movl −4(%ecx, edx, 4), %eax
8	A+i	int *	*SA*+4**i*	leal (%ecx, %edx, 4), %eax

2、3、6和7对应汇编指令都需访存，指令中源操作数的寻址方式分别是“基址”、“基址加比例变址”、“基址加比例变址”和“基址加比例变址加位移”的方式，因为数组元素的类型为int型，故比例因子为4 。

数组元素在内存的存放和访问

- **指针数组和多维数组**
  - **由若干指向同类目标的指针变量组成的数组称为指针数组。**
  - **其定义的一般形式如下：**

    存储类型 数据类型 *指针数组名[元素个数]；
  - **例如，“int *a[10];”定义了一个指针数组a，它有10个元素，每个元素都是一个指向int型数据的指针。**
    - **一个指针数组可以实现一个二维数组。**



数组元素在内存的存放和访问

- **指针数组和多维数组**　　按行优先方式存放数组元素
  - 计算一个两行四列整数矩阵中每一行数据的和。

```
main ( )
{
   static short num[ ][4]={ {2, 9, -1, 5},
                            {3, 8, 2, -6}};
   static short *pn[ ]={num[0], num[1]};
   static short s[2]={0, 0};
   int i, j;
   for (i=0; i<2; i++) {
      for (j=0; j<4; j++)
           s[i]+=*pn[i]++;
      printf (sum of line %d : %d\n” , i+1, s[i]);
   }
}
```

当i=1时，pn[i]=*(pn+i)=M[pn+4*i]=0x8049308

若处理“s[i]+=*pn[i]++;”时 i 在ECX，s[i]在AX，pn[i]在EDX，则对应指令序列可以是什么？

```
movl   pn(,%ecx,4), %edx
addw   (%edx), %ax
addl   $2, pn(, %ecx, 4)
```

pn[i]+“1”→pn[i]

若num=0x8049300,则num、pn和s在存储区中如何存放？

```
08049300 <num>:
08049300 :   02 00 09 00 ff ff 05 00 03 00 08 00 02 00 fa ff
08049310 <pn>:
08049310 :   00 93 04 08 08 93 04 08
08049318<s>:
08049318 :   00 00 00 00
```

num=num[0]=&num[0][0]=0x8049300

pn=&pn[0]=0x8049310
pn[0]=num[0]=0x8048300
pn[1]=num[1]=0x8048308

结构和联合数据类型的分配和访问

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

结构体数据的分配和访问

- 结构体成员在内存的存放和访问
  - 分配在栈中的auto结构型变量的首地址由EBP或ESP来定位
  - 分配在静态区的结构型变量首地址是一个确定的静态区地址
  - 结构型变量 x 各成员首址可用“基址加偏移量”的寻址方式

```
struct cont_info {
        char id[8];
        char name [12];
        unsigned post;
        char address[100];
        char phone[20];
};
```

若变量x分配在地址0x8049200开始的区域，那么
x=&(x.id)=0x8049200（若x在EDX中）
&(x.name)= 0x8049200+8=0x8049208
&(x.post)= 0x8049200+8+12=0x8049214
&(x.address)=0x8049200+8+12+4=0x8049218
&(x.phone)=0x8049200+8+12+4+100=0x804927C

struct cont_info x={ “0000000”, “ZhangS”, 210022, “273 long street, High Building #3015”, “12345678” } ;

x初始化后，在地址0x8049208到0x804920D处是字符串“ZhangS”，0x804920E处是字符‘\0’，从0x804920F到0x8049213处都是空字符。

“unsigned xpost=x.post;”对应汇编指令为“movl 20(%edx), %eax”

结构体数据的分配和访问

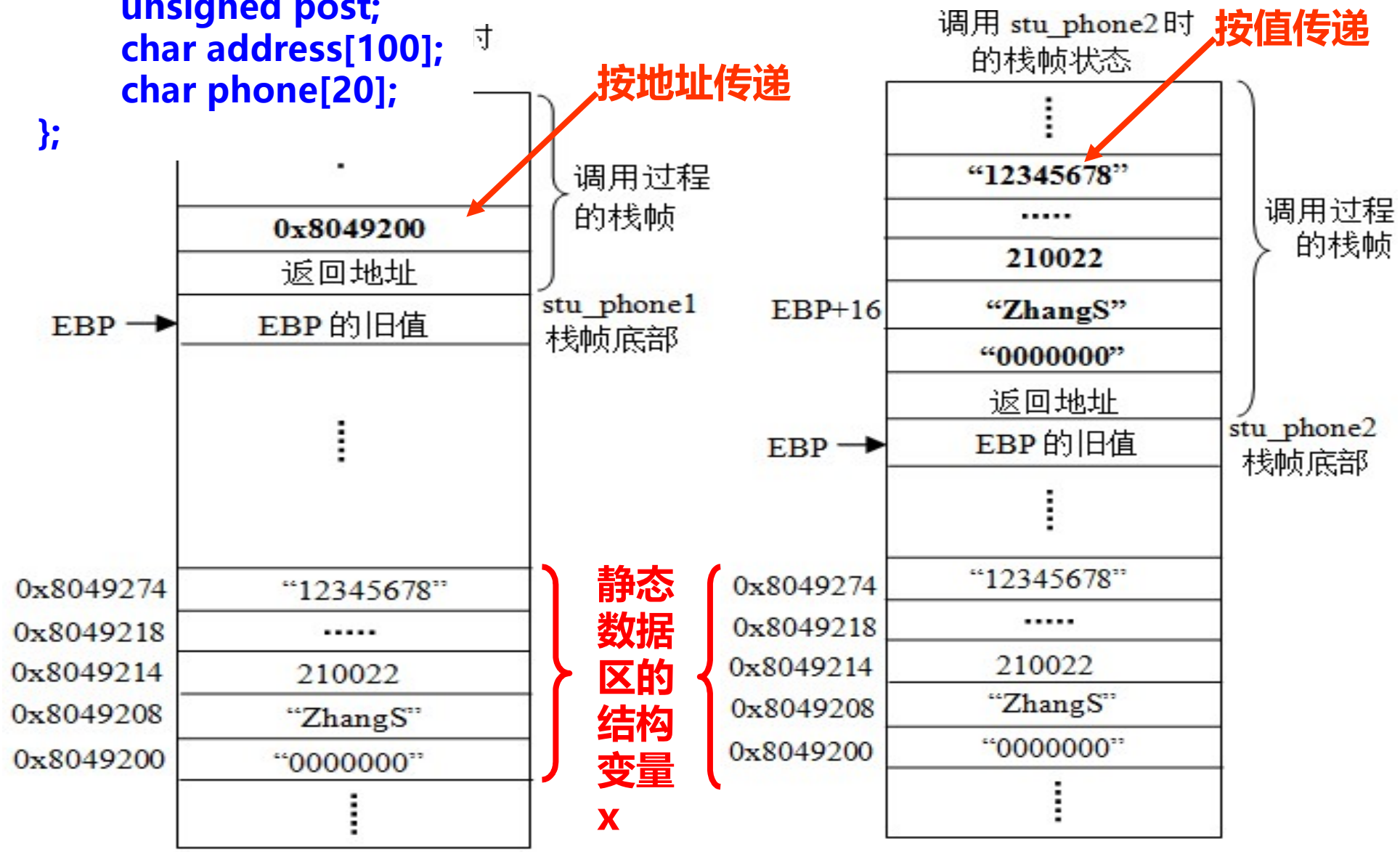
- **结构体数据作为入口参数**
  - **当结构体变量需要作为一个函数的形参时，形参和调用函数中的实参应具有相同结构**
  - **有按值传递和按地址传递两种方式**
    - **若采用按值传递，则结构成员都要复制到栈中参数区，这既增加时间开销又增加空间开销，且更新后的数据无法在调用过程使用**
    - **通常应按地址传递，即：在执行CALL指令前，仅需传递指向结构体的指针而不需复制每个成员到栈中**

```
void stu_phone1 ( struct cont_info *s_info_ptr)    按地址调用
{
   printf ( "%s phone number: %s", (*s_info_ptr).name, (*s_info_ptr).phone);
}
void stu_phone2 ( struct cont_info s_info)     按值调用
{
   printf ( "%s phone number: %s", s_info.name, s_info.phone);
}
```

结构体数据的分配和访问

```
struct cont_info {
        char id[8];
        char name [12];
        unsigned post;
        char address[100];
        char phone[20];
};
```

- **结构体数据作为入口参数（若对应实参是x）**



结构体数据的分配和访问

- **按地址传递参数**

  (*stu_info).name可写成

  stu_info->name，执行

  以下两条指令后:

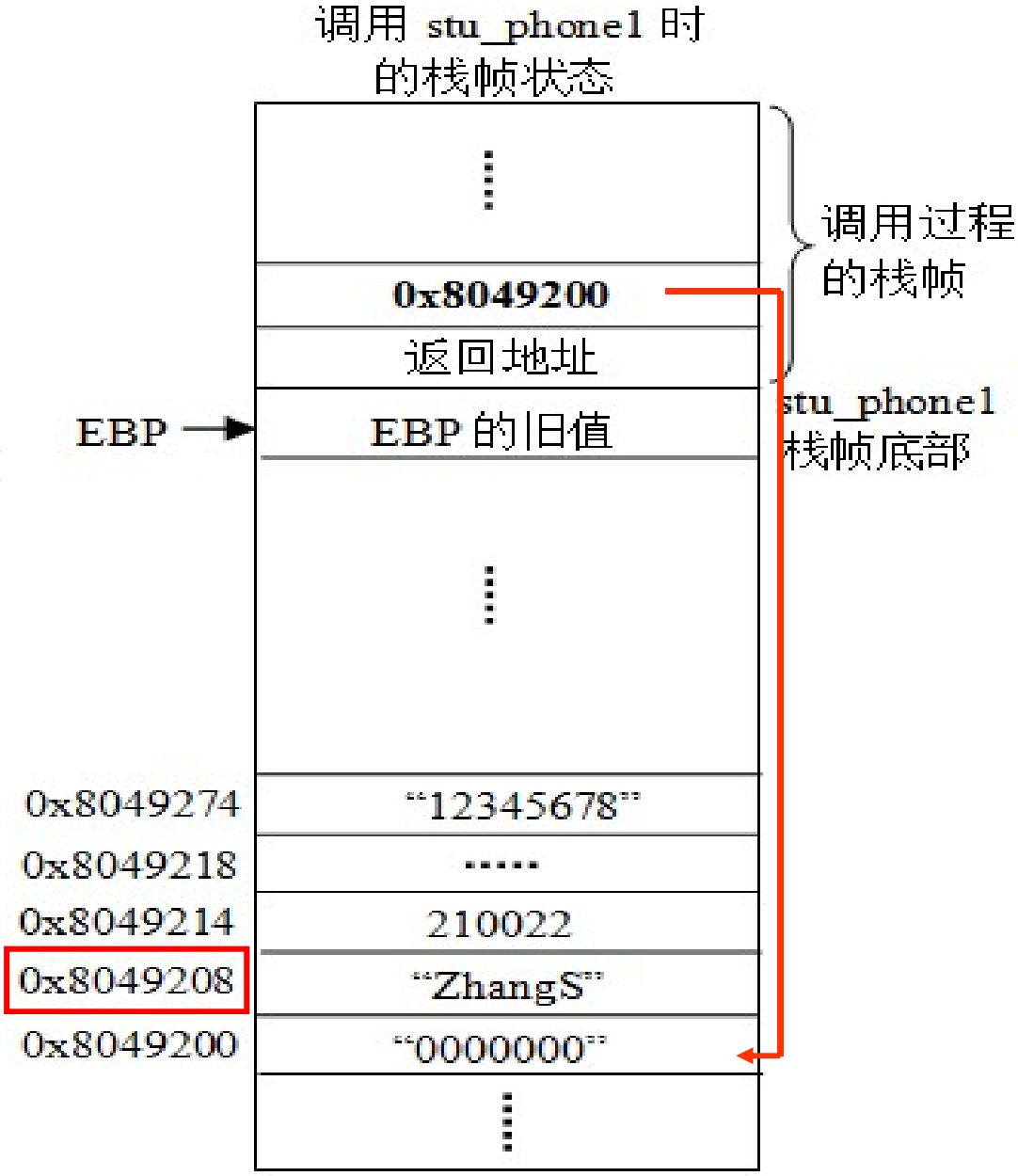
  movl 8(%ebp), %edx

  leal 8(%edx), %eax

  EAX中存放的是字符串

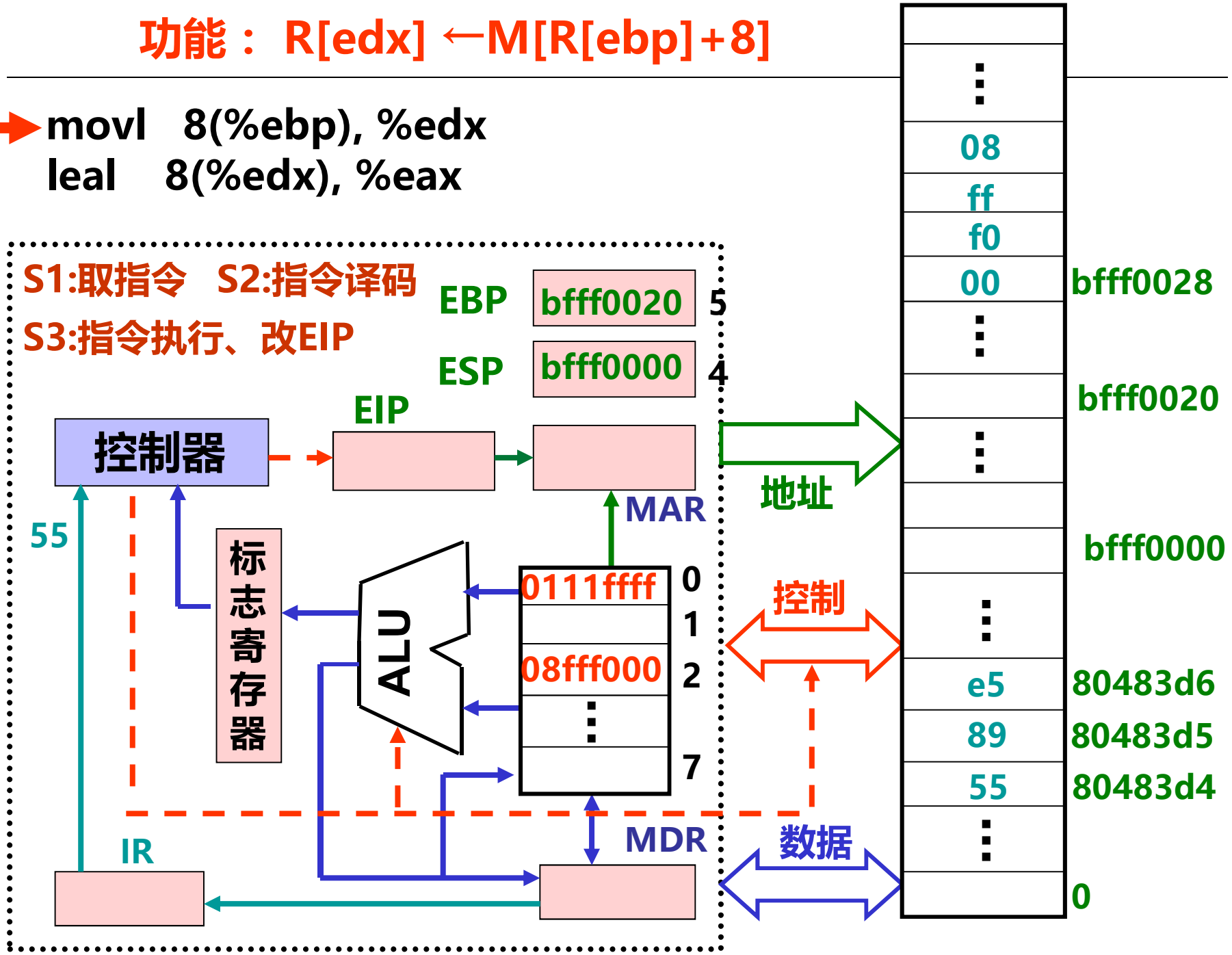
  “ZhangS”在静态存储区内的首地址

  0x8049208



功能： R[edx] ←M[R[ebp]+8]

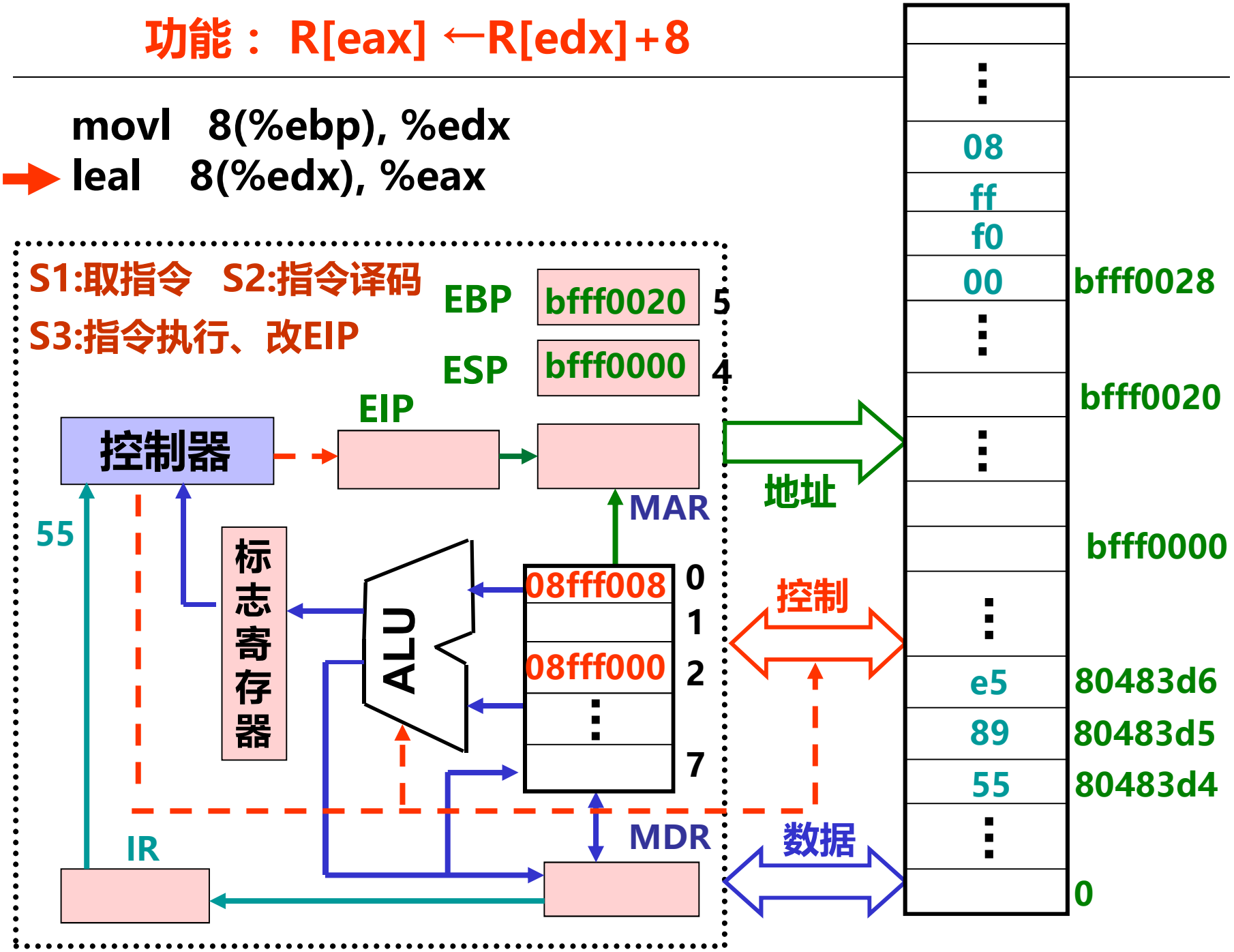
movl 8(%ebp), %edx
leal 8(%edx), %eax



功能： R[eax] ←R[edx]+8

movl 8(%ebp), %edx

leal 8(%edx), %eax



结构体数据的分配和访问

- **按值传递参数**

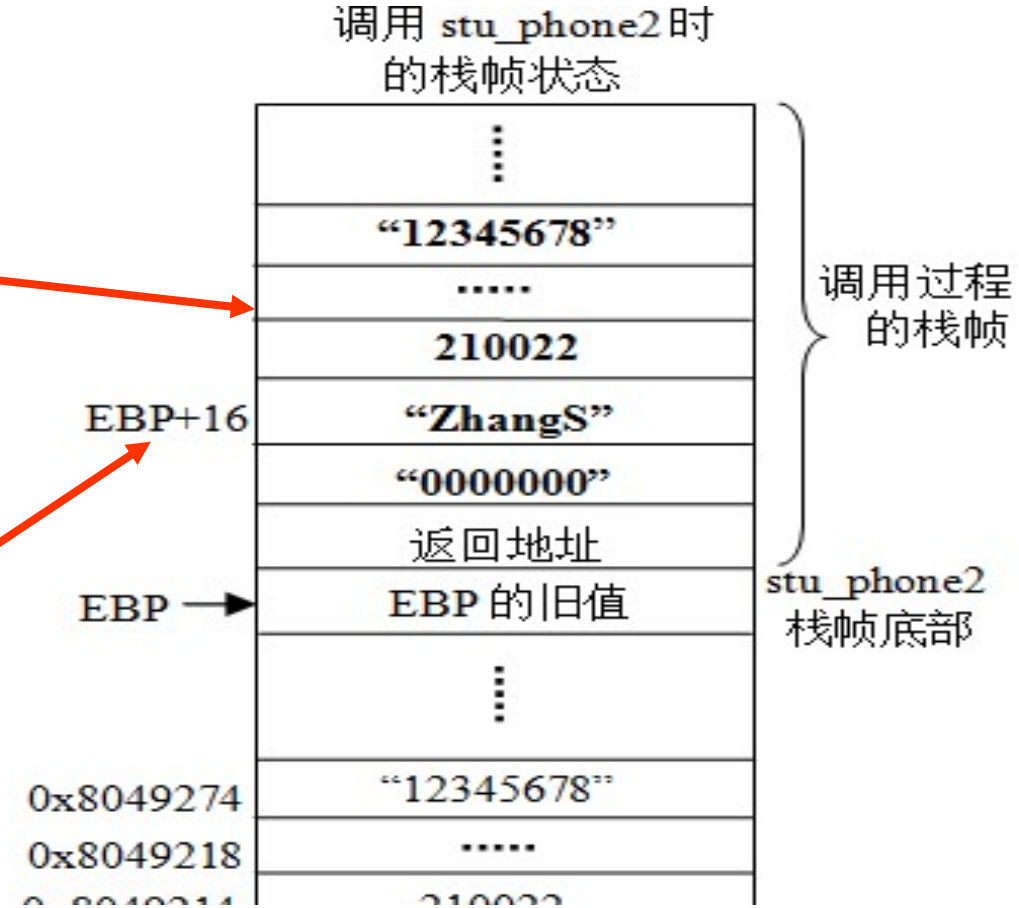
  x所有成员值作为实参存到参数区。

  stu_info.name送EAX的指令序列为：

  leal 8(%ebp), %edx

  leal 8(%edx), %eax

  EAX中存放的是“ZhangS”的栈内参数区首址。



  - **stu_phone1和stu_phone2功能相同，但两者的时、空开销都不一样。后者开销大，因为它需对结构体成员整体从静态区复制到栈中，需要很多条mov或其他指令，从而执行时间更长，并占更多栈空间和代码空间**

联合体数据的分配和访问

联合体各成员共享存储空间，按最大长度成员所需空间大小为目标

```
union uarea {
        char  c_data;
        short s_data;
        int     i_data;
        long   l_data;
};
```

IA-32中编译时，long和int长度一样，故uarea所占空间为4个字节。而对于与uarea有相同成员的结构型变量来说，其占用空间大小至少有11个字节，对齐的话则占用更多。

- **通常用于特殊场合，如，当事先知道某种数据结构中的不同字段的使用时间是互斥的，就可将这些字段声明为联合，以减少空间。**
- **但有时会得不偿失，可能只会减少少量空间却大大增加处理复杂性。**

联合体数据的分配和访问

- 还可实现对相同位序列进行不同数据类型的解释

```
unsigned
float2unsign( float f)
{
    union {
            float f;
            unsigned u;
     } tmp_union;
     tmp_union.f=f;
     return tmp_union.u;
}
```

函数形参是float型，按值传递参数，因而传递过来的实参是float型数据，赋值给非静态局部变量（联合体变量成员）

过程体为:

```
movl 8(%ebp), %eax
movl %eax, -4(%ebp)
movl -4(%ebp) , %eax
```

} 可优化掉！

将存放在地址R[ebp]+8处的入口参数 f 送到EAX（返回值）

从该例可看出：机器级代码并不区分所处理对象的数据类型，不管高级语言中将其说明成float型还是int型或unsigned型，都把它当成一个0/1序列来处理。

IA-32寄存器组织的模拟

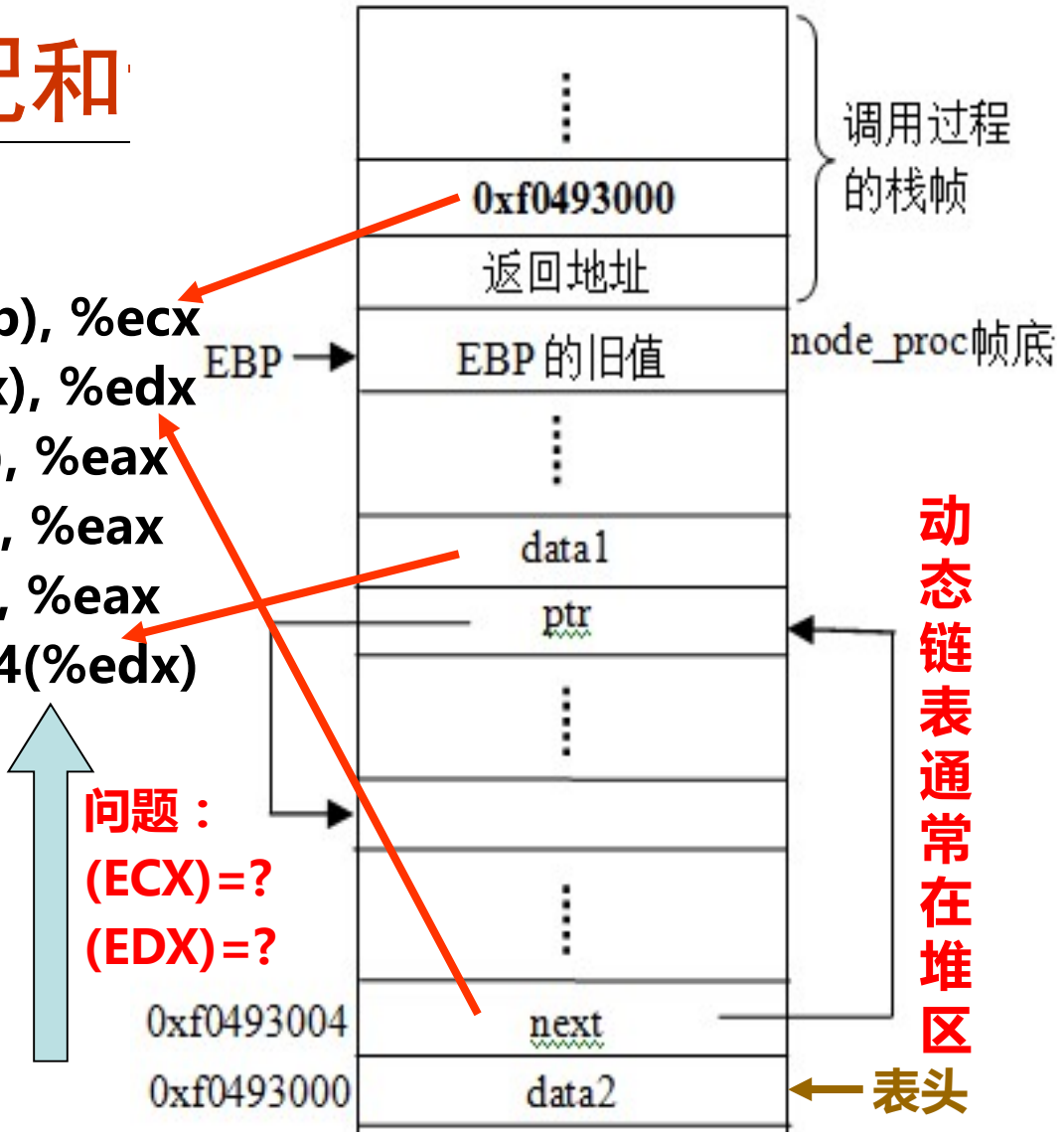
```
typedef struct{
union{
      struct {        uint32_t  eax;
                      uint32_t  ecx;
                      uint32_t  edx;
                      uint32_t  ebx;
                      uint32_t  esp;
                      uint32_t  ebp;
                      uint32_t  esi;
                      uint32_t  edi;};
      union{
                      uint32_t  _32;
                      uint16_t  _16;
                      uint8_t  _8[2];
            } gpr[8];
    };
} CPU_state;

extern CPU_state cpu;
enum { R_EAX, R_ECX, R_EDX, R_EBX, R_ESP, R_EBP, R_ESI, R_EDI };
enum { R_AX, R_CX, R_DX, R_BX, R_SP, R_BP, R_SI, R_DI };
enum { R_AL, R_CL, R_DL, R_BL, R_AH, R_CH, R_DH, R_BH };

#define reg_l(index)   (cpu.gpr[index]._32)
#define reg_w(index)  (cpu.gpr[index]._16)
#define reg_b(index)  (cpu.gpr[index & 0x3]._8[index >> 2])
```

联合体数据的分配和

- **利用嵌套可定义链表结构**

```
union node {
      struct {
          int *ptr;
          int data1
       } node1;
       struct {
           int data2;
           union node *next;
        } node2;
};
```

```
movl  8(%ebp), %ecx
movl  4(%ecx), %edx
movl  (%edx), %eax
movl  (%eax), %eax
addl  (%ecx), %eax
movl  %eax, 4(%edx)
```




```
void node_proc ( union node *np) {
  np->node2.next->node1.data1=*(np->node2.next->node1.ptr)+np->node2.data2;
}
```

数据的对齐存放

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

数据的对齐

Alignment: 要求数据的地址是相应的边界地址

- 目前机器字长为32位或64位，主存按一个传送单位（32/64/128位）进行存取，而按字节编址，例如：若传送单位为64位，则每次最多读写64位，即：第0~7字节同时读写，第8~15字节同时读写，……，以此类推。按边界对齐，可使读写数据位于8i~8i+7(i=0,1,2,...) 单元
- 指令系统支持对字节、半字、字及双字的运算
- 各种不同长度的数据存放时，有两种处理方式:
  - 按边界对齐（若一个字为32位）
    - 字地址：4的倍数(低两位为0)
    - 半字地址：2的倍数(低位为0)
    - 字节地址：任意
  - 不按边界对齐

    坏处：可能会增加访存次数！（学了存储器组织后会更明白！）

对齐(Alignment)

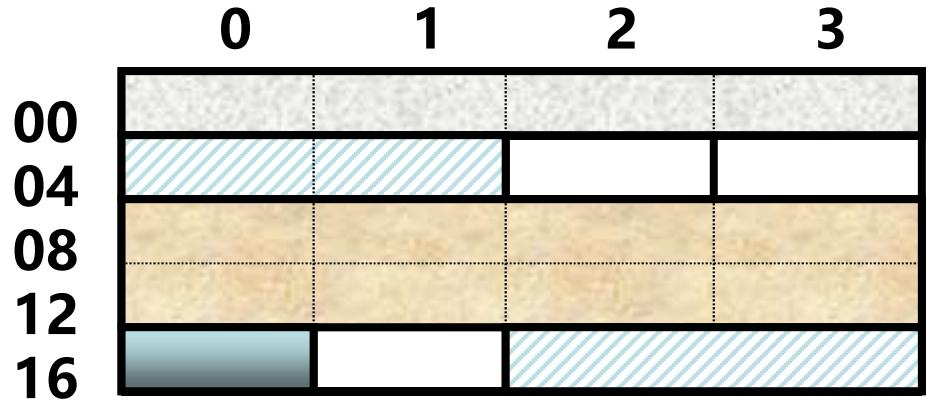
若1个字=32位，主存每次最多存取一个字，按字节编址，则每次**只能**读写某个字地址开始的4个单元中连续的1、2、3或4个字节

如：int i, short k, double x, char c, short j,……

按边界对齐

x：2个周期

j：1个周期



则：&i=0; &k=4; &x=8; &c=16; &j=18;……

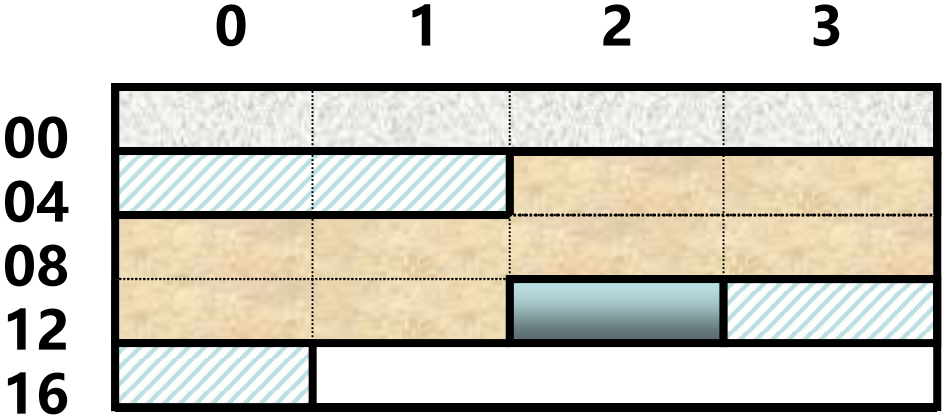
虽节省了空间，但增加了访存次数！

需要权衡，目前来看，浪费一点存储空间没有关系！

边界不对齐

x：3个周期

j：2个周期



则： &i=0; &k=4; &x=6; &c=14; &j=15;……

数据的对齐

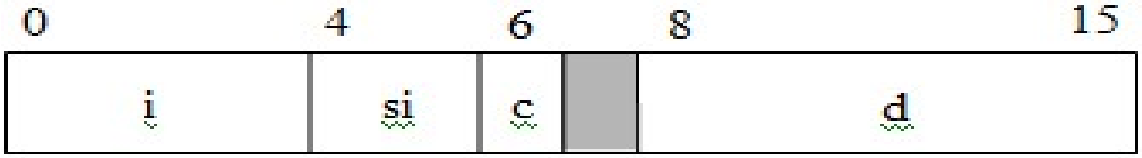
- 最简单的对齐策略是：按其数据长度进行对齐。例如，
  - Windows采用策略：int型地址是4的倍数，short型地址是2的倍数，double和long long型的是8的倍数，float型的是4的倍数，char不对齐
  - Linux采用更宽松策略：short型是2的倍数，其他类型如int、float、double和指针等都是4的倍数

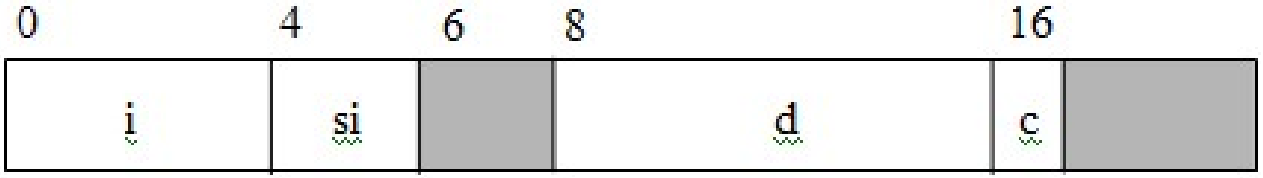
```
struct SDT {
    int       i;
    short     si;
    double    d;
    char      c;
} sa[10];
```

结构数组变量的最末可能需要插空，以使每个数组元素都按4字节边界对齐

```
struct SD {
    int       i;
    short     si;
    char      c;
    double    d;
};
```

结构变量首地址按4字节边界对齐





只要SD首址按4B边界对齐，所有字段都能按要求对齐

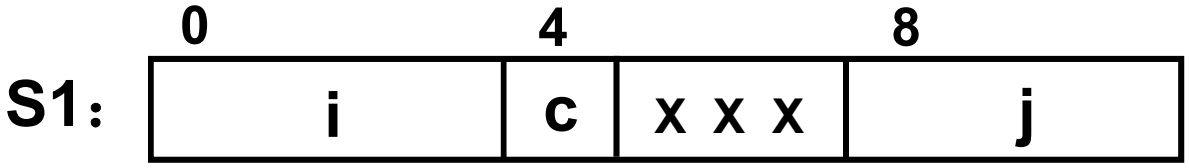
对齐(Alignment)举例

例如，考虑下列两个结构声明：

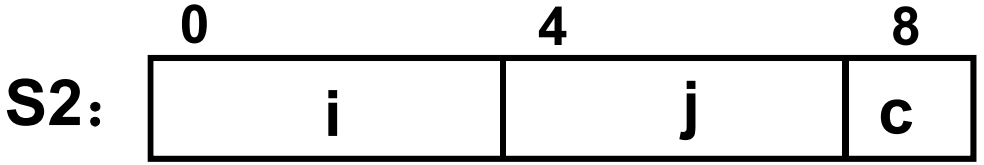
```
struct  S1 {
        int     i ;
        char    c ;
        int     j ;
} ;
```

```
struct  S2 {
        int     i ;
        int     j ;
        char    c ;
} ;
```

在要求对齐的情况下，哪种结构声明更好？ S2比S1好

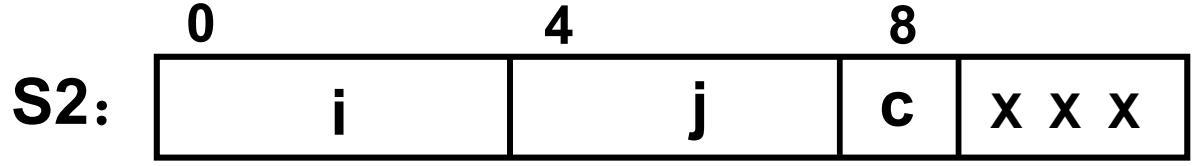


需要12个字节



只需要9个字节

对于“struct S2 d[4]”，只分配9个字节能否满足对齐要求？ 不能！



也需要12个字节

对齐方式的设定

#pragma pack(n)

- **为编译器指定结构体或类内部的成员变量的对齐方式。**
- **当自然边界（如int型按4字节、short型按2字节、float按4字节）比n大时，按n字节对齐。**
- **缺省或#pragma pack()，按自然边界对齐。**

__attribute__((aligned(m)))

- **为编译器指定一个结构体或类或联合体或一个单独的变量(对象)的对齐方式。**
- **按m字节对齐(m必须是2的幂次方)，且其占用空间大小也是m的整数倍，以保证在申请连续存储空间时各元素也按m字节对齐。**

__attribute__((packed))

- **不按边界对齐，称为紧凑方式。**

对齐方式的设定

```
#include<stdio.h>
#pragma pack(4)
typedef struct {
    uint32_t    f1;
    uint8_t     f2;
    uint8_t     f3;
    uint32_t    f4;
    uint64_t    f5;
}__attribute__((aligned(1024))) ts;
int main()
{
    printf("Struct size is: %d, aligned on 1024\n",sizeof(ts));
    printf("Allocate f1 on address: 0x%x\n",&(((ts*)0)->f1));
    printf("Allocate f2 on address: 0x%x\n",&(((ts*)0)->f2));
    printf("Allocate f3 on address: 0x%x\n",&(((ts*)0)->f3));
    printf("Allocate f4 on address: 0x%x\n",&(((ts*)0)->f4));
    printf("Allocate f5 on address: 0x%x\n",&(((ts*)0)->f5));
    return 0;
}
```

输出：

```
Struct size is: 1024, aligned on 1024
Allocate f1 on address: 0x0
Allocate f2 on address: 0x4
Allocate f3 on address: 0x5
Allocate f4 on address: 0x8
Allocate f5 on address: 0xc
```

```c
#include <stdio.h>
//#pragma pack(1)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long  x4;
}__attribute__((aligned(8)));

void main()
{
    printf("size=%d\n",sizeof(struct test));
    printf("size=%d\n",sizeof(struct test1));
    printf("size=%d\n",sizeof(struct test2));
}
```

输出结果是什么？

```
size=15
size=20
size=24
```

```c
#include <stdio.h>
#pragma pack(1)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long  x4;
}__attribute__((aligned(8)));
void main()
{
    printf("size=%d\n",sizeof(struct test));
    printf("size=%d\n",sizeof(struct test1));
    printf("size=%d\n",sizeof(struct test2));
}
```

如果设置了pragma pack(1)，结果又是什么？

```
size=15
size=15
size=16
```

```c
#include <stdio.h>
#pragma pack(2)
struct test
{
    char x2;
    int x1;
    short x3;
    long long x4;
}__attribute__((packed));
struct test1
{
    char x2;
    int x1;
    short x3;
    long long x4;
};
struct test2
{
    char x2;
    int x1;
    short x3;
    long long  x4;
}__attribute__((aligned(8)));
void main()
{
    printf("size=%d\n",sizeof(struct test));
    printf("size=%d\n",sizeof(struct test1));
    printf("size=%d\n",sizeof(struct test2));
}
```

如果设置了pragma pack(2)，结果又是什么？

```
size=15
size=16
size=16
```

越界访问和缓冲区溢出攻击

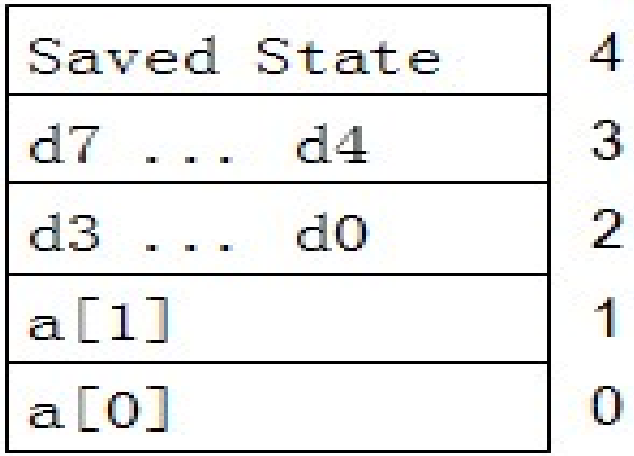
南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

越界访问和缓冲区溢出

大家还记得以下的例子吗？

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14, 然后存储保护错



为什么当 i>1 就有问题？
因为数组访问越界！

数组元素可使用指针来访问，因而对数组的引用没有边界约束

越界访问和缓冲区溢出

- C语言程序中对数组的访问可能会有意或无意地超越数组存储区范围而无法发现。
- 数组存储区可看成是一个缓冲区，超越数组存储区范围的写入操作称为缓冲区溢出。
  - 例如，对于一个有10个元素的char型数组，其定义的缓冲区有10个字节。若写一个字符串到这个缓冲区，那么只要写入的字符串多于9个字符（结束符‘\0’占一个字节），就会发生“写溢出”。
- 缓冲区溢出是一种非常普遍、非常危险的漏洞，在各种操作系统、应用软件中广泛存在。
- 缓冲区溢出攻击是利用缓冲区溢出漏洞所进行的攻击。利用缓冲区溢出攻击，可导致程序运行失败、系统关机、重新启动等后果。

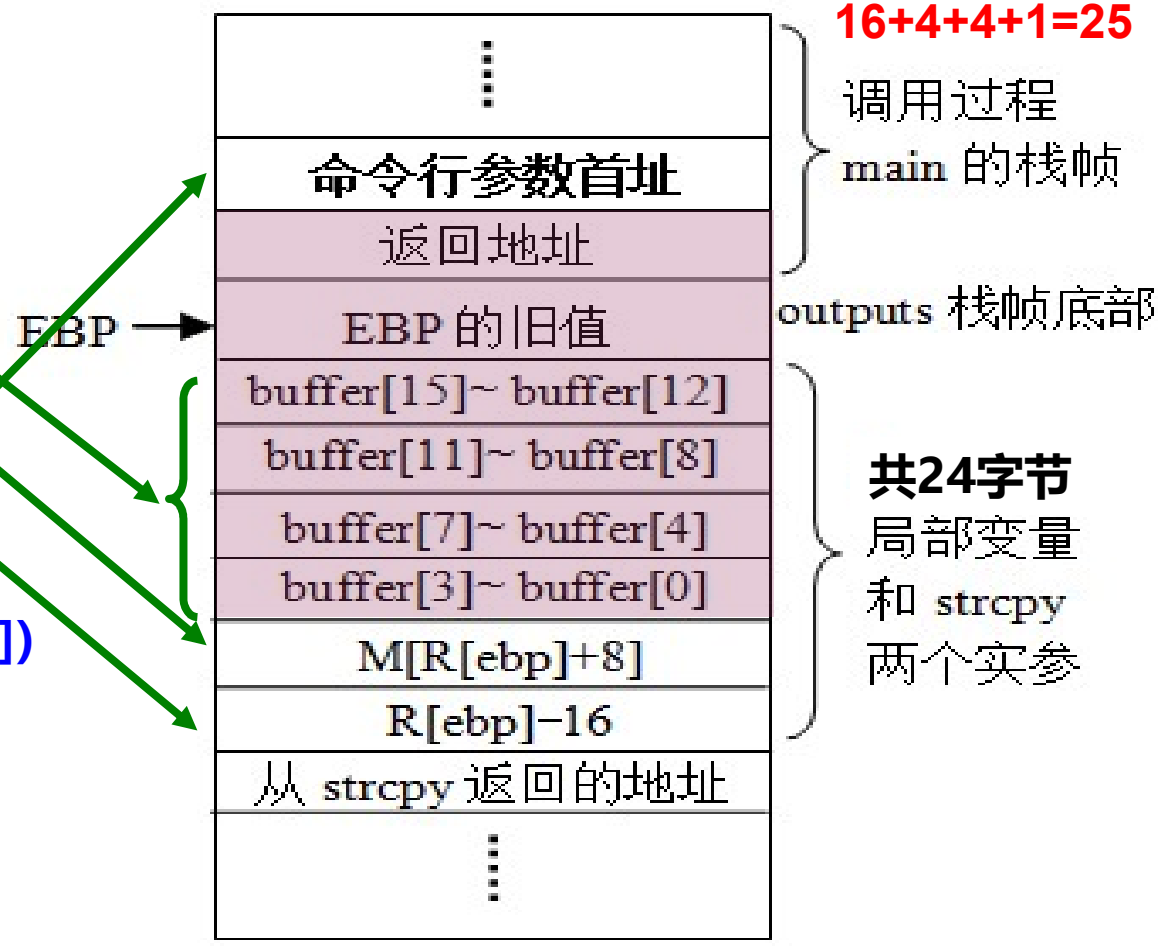
越界访问和缓冲区溢出

- 造成缓冲区溢出的原因是没有对栈中作为缓冲区的数组的访问进行越界检查。 举例：利用缓冲区溢出转到自设的程序hacker去执行

outputs漏洞：当命令行中字符串超25个字符时，使用strcpy函数就会使缓冲buffer造成写溢出并破坏返址

```
#include "stdio.h"
#include "string.h"
void outputs(char *str)
{
    char buffer[16];
    strcpy(buffer,str);
    printf("%s \n", buffer);
}
void hacker(void)
{
    printf("being hacked\n");
}
int main(int argc, char *argv[])
{
    outputs(argv[1]);
    return 0;
}
```

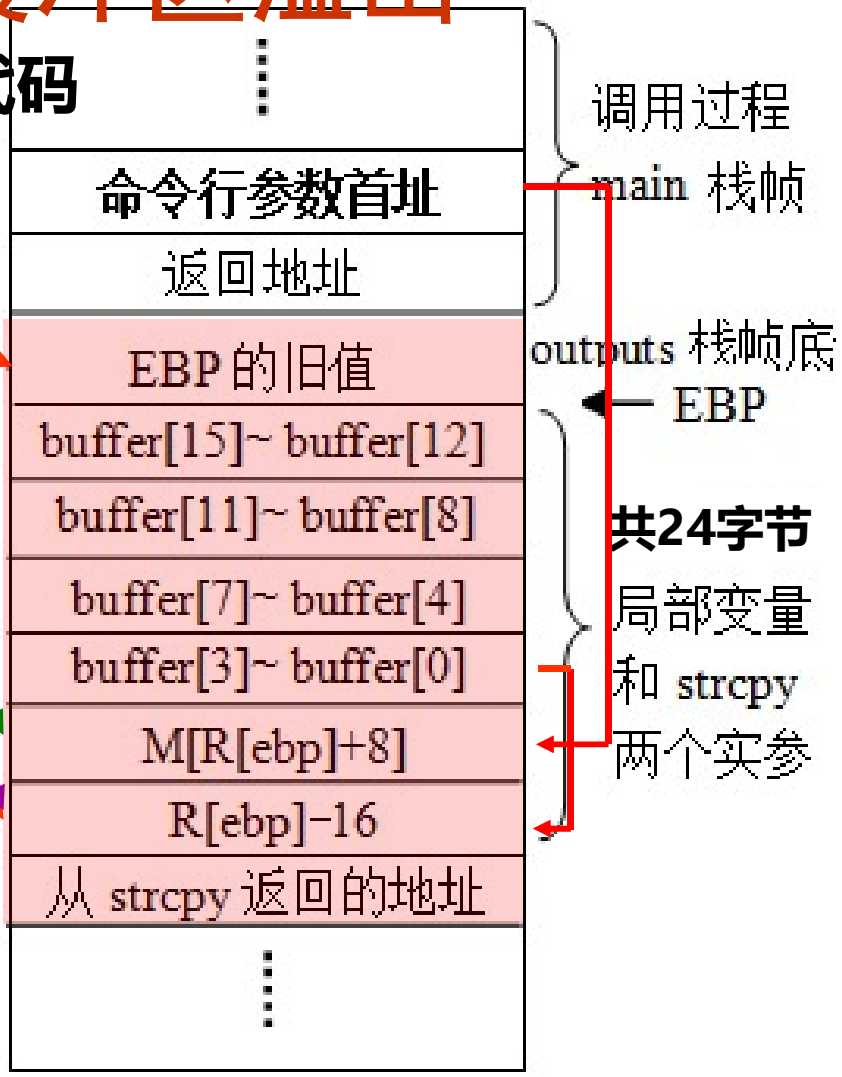
假定可执行文件名为test



越界访问和缓冲区溢出

test被反汇编得到的outputs汇编代码

```
080483e4 push   %ebp
080483e5 mov    %esp,%ebp
080483e7 sub    $0x18,%esp
080483ea mov    0x8(%ebp),%eax
080483ed mov    %eax,0x4(%esp)
080483f1 lea    0xfffffff0(%ebp),%eax
080483f4 mov    %eax,(%esp)
080483f7 call   0x8048330 <strcpy>
080483fc lea    0xfffffff0(%ebp),%eax
080483ff mov    %eax,0x4(%esp)
08048403 movl   $0x8048500,(%esp)
0804840a call   0x8048310
0804840f  leave
08048410 ret
```



若strcpy复制了25个字符到buffer中，并将hacker首址置于结束符 ‘\0’ 前4个字节，则在执行strcpy后，hacker代码首址被置于main栈帧返回地址处，当执行outputs代码的ret指令时，便会转到hacker函数实施攻击。

程序的加载和运行

- UNIX/Linux系统中，可通过调用execve()函数来加载并执行程序。
- execve()函数的用法如下：

  int execve(char *filename, char *argv[], *envp[]);

  filename是加载并运行的可执行文件名(如./hello)，可带参数列表argv和环境变量列表envp。若错误（如找不到指定文件filename），则返回-1，并将控制权交给调用程序； 若函数执行成功，则不返回，最终将控制权传递到可执行目标中的主函数main。
- 主函数main()的原型形式如下：

  int main(int argc, char **argv, char **envp); 或者：

  int main(int argc, char *argv[], char *envp[]);

  argc指定参数列表长度，参数列表中开始是命令名（可执行文件名），最后以NULL结尾

  例如：参数列表（命令行）为“.\hello” 时，argc=2

  前述例子：“.\test 0123456789ABCDEFXXXX▥ ▧▥▧” ,argc=3

  argv[0] argv[1]

缓冲区溢出攻击

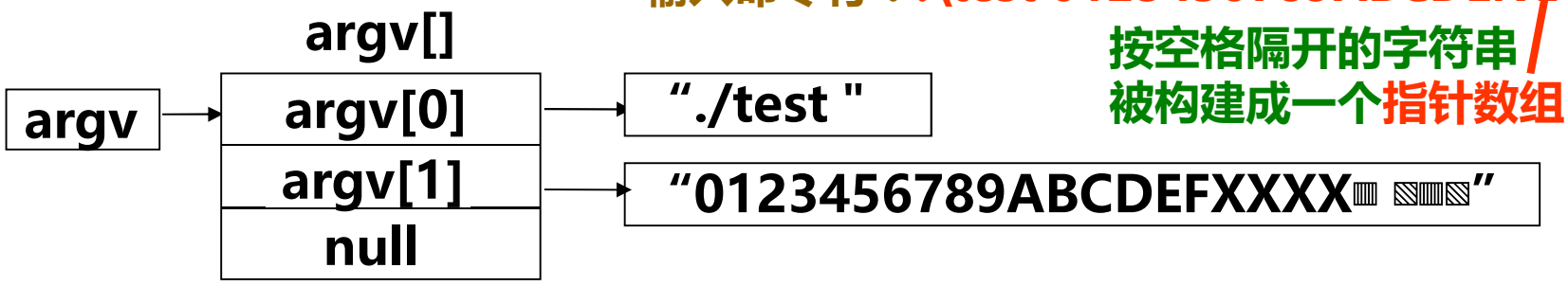
```
#include "stdio.h"
char code[]=
      "0123456789ABCDEFXXXX"
      "\x11\x84\x04\x08"
      "\x00";
int main(void)
{
      char *argv[3];
      argv[0]="./test";
      argv[1]=code;
      argv[2]=NULL;
      execve(argv[0],argv,NULL);
      return 0;
}
```

```
#include "stdio.h"
#include "string.h"
void outputs(char *str)
{
    char buffer[16];
    strcpy(buffer,str);
    printf("%s \n", buffer);
}
void hacker(void)
{
    printf("being hacked\n");
}
int main(int argc, char *argv[])
{
    outputs(argv[1]);
    return 0;
}
```

可执行文件名为test

输入命令行：.\test 0123456789ABCDEFXXXX▯ ▧▯▧

按空格隔开的字符串
被构建成一个指针数组



越界访问和缓冲区溢

假定hacker首址为0x08048411

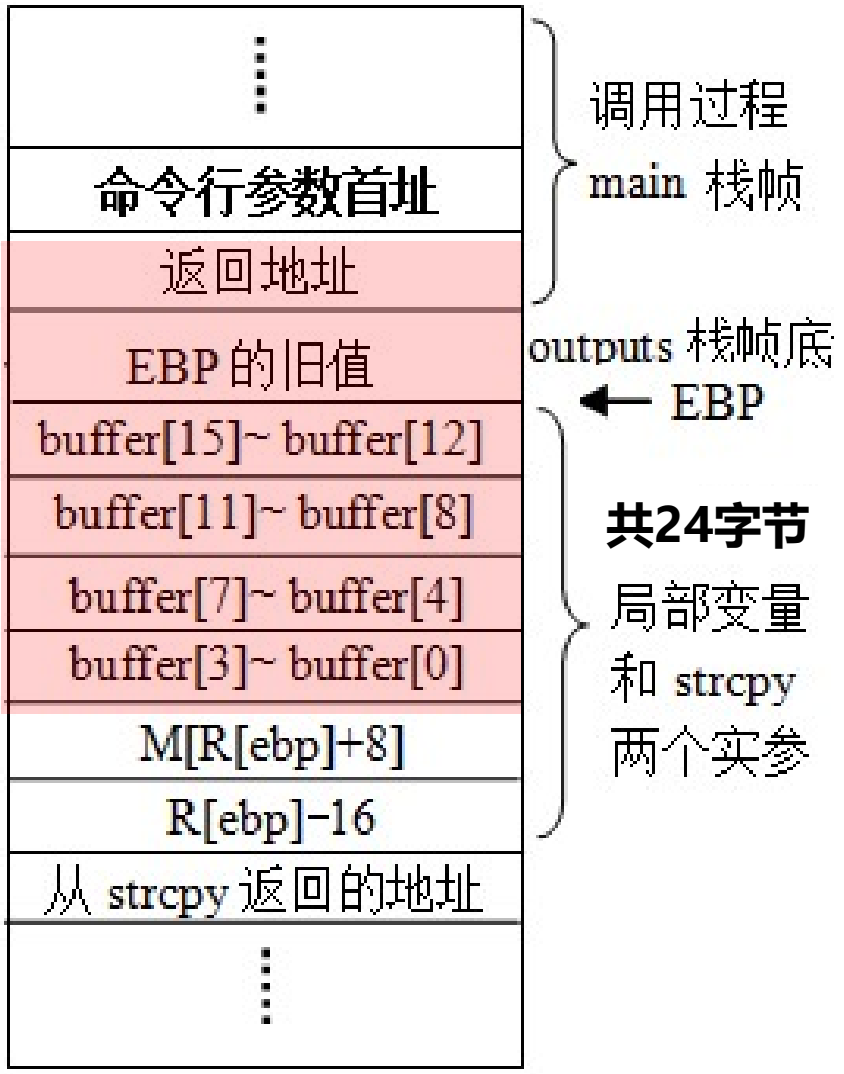
```
void hacker(void) {
    printf("being hacked\n");
}
#include "stdio.h"
char code[]=
      "0123456789ABCDEFXXXX"
      "\x11\x84\x04\x08"
      "\x00";
int main(void) {
      char *argv[3];
      argv[0]="./test";
      argv[1]=code;
      argv[2]=NULL;
      execve(argv[0],argv,NULL);
      return 0;
}
```

执行上述攻击程序后的输出结果为：

"0123456789ABCDEFXXXX▯ ▧▯▧

being hacked

Segmentation fault



最后显示“Segmentation fault”，原因是执行到hacker过程的ret指令时取到的“返回地址”是一个不确定的值，因而可能跳转到数据区或系统区或其他非法访问的存储区执行，因而造成段错误。