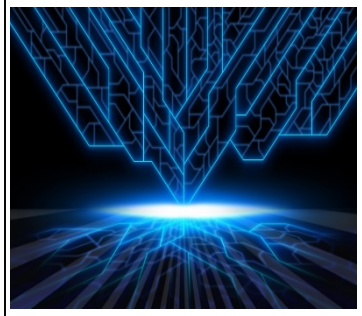
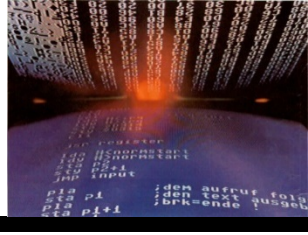
第11讲
数据结构



吴海军
南京大学计算机科学与技术系

主要内容

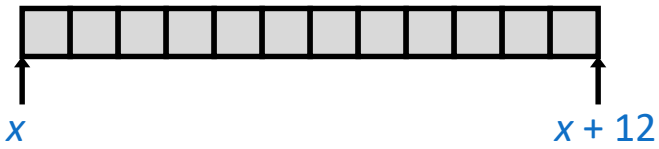


- 数组
- 指针
- 结构
- 联合

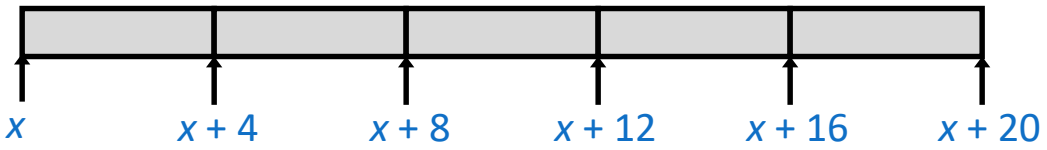
数组分配

- 格式：*T* `A[`*N*`];`
  - 数据类型为T、长度为L的数组
  - 连续分配*L* * `sizeof(`*N*`)` 字节空间

char string[12];

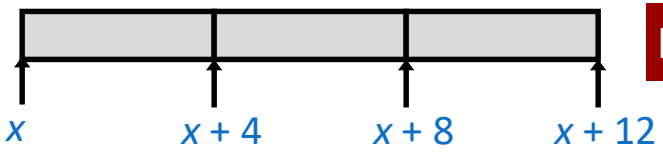


int val[5];

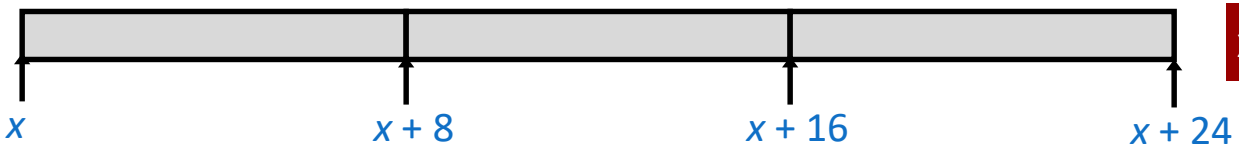


double a[3];

char *p[3];

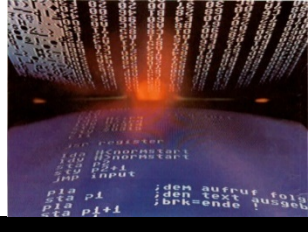


IA32



x86-64

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
```

其中，ECX为变址（索引）寄存器，在循环体中增量 **比例因子是2!**

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

```
08049908 <buf>:
08049908:    0A 00 00 00 14 00 00 00
```

此时，buf=&buf[0]=0x08049908

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

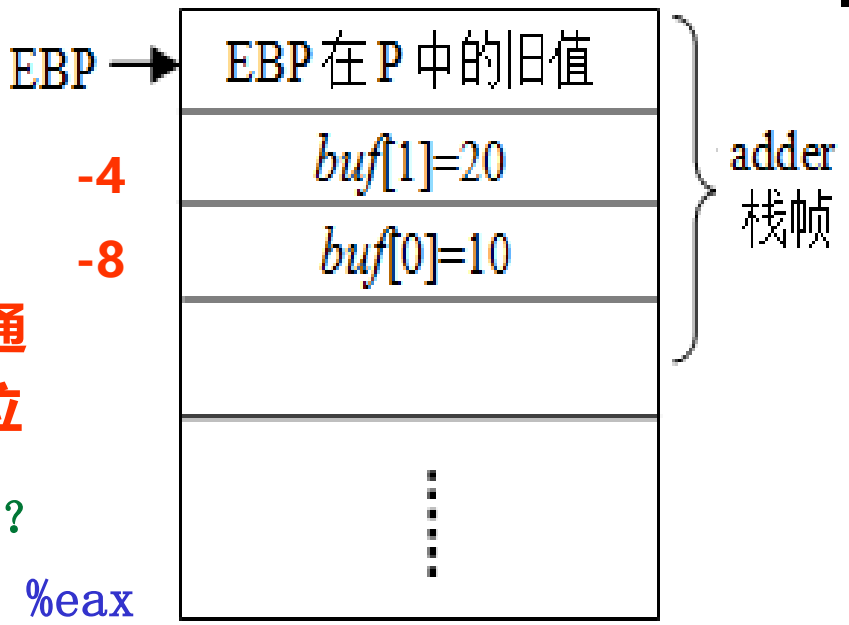
- auto型数组的初始化和访问

```
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？

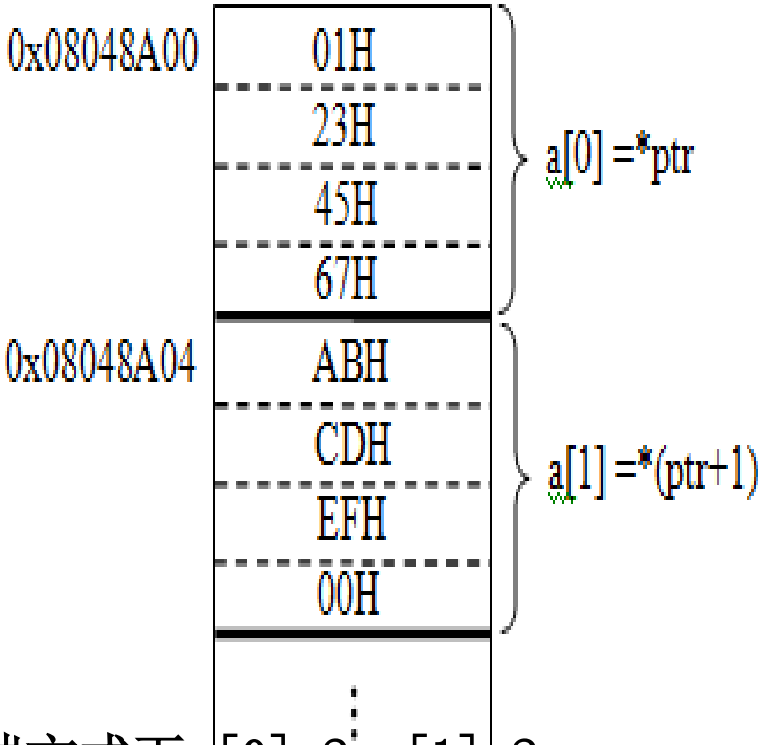
```
leal -8(%ebp), %edx  //buf[0]的地址为R[ebp]-8，将buf首址送EDX
```

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。
  - 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

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

假设A首址SA在ECX，i 在EDX，结果在EAX

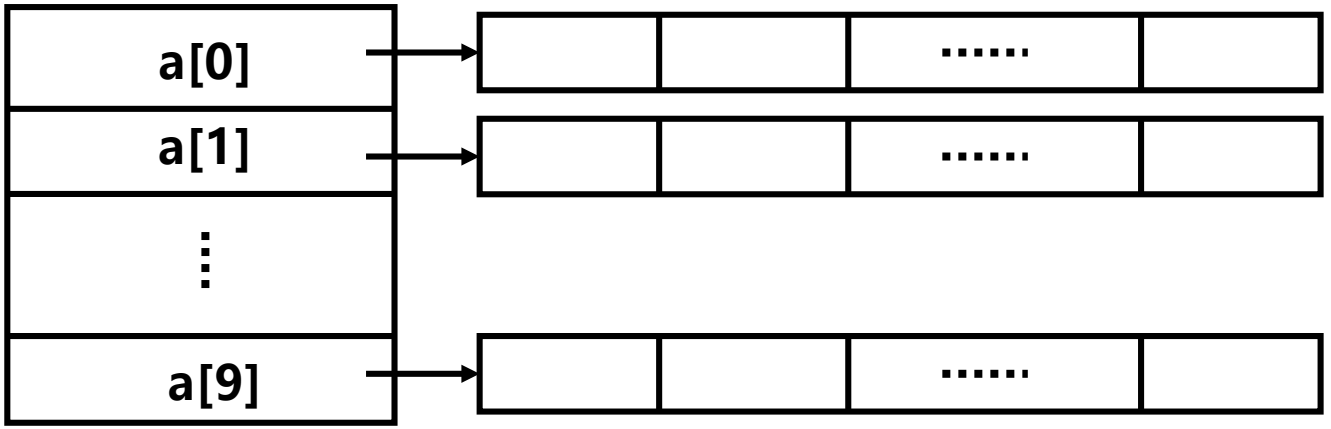
序号	表达式	类型	值的计算方式	汇编代码
1	A	int *	*SA*	leal (%ecx), %eax
2	A[0]	int	M[*SA*]	movl (%ecx), %eax
3	A[i]	int	M[*SA*+4**i*]	movl (%ecx, %edx, 4), %eax
4	&A[3]	int *	*SA*+12	leal 12(%ecx), %eax
5	&A[i]-A	int	(*SA*+4**i*-*SA*)/4=*i*	movl %edx, %eax
6	*(A+i)	int	M[*SA*+4**i*]	movl (%ecx, %edx, 4), %eax
7	*(&A[0]+i-1)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

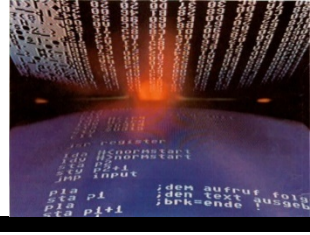
若num=0x8049300,则num、pn和s在存储区中如何存放?

```
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]=0x8049300
pn[1]=num[1]=0x8049308

N*N矩阵



```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele
   (fix_matrix a, int i, int j){
   return a[i][j];
}
```

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele
  (int n, int *a, int i, int j){
   return a[IDX(n,i,j)];
}
```

```
/* Get element a[i][j] */
int var_ele
  (int n, int a[n][n], int i, int j){
   return a[i][j];
}
```

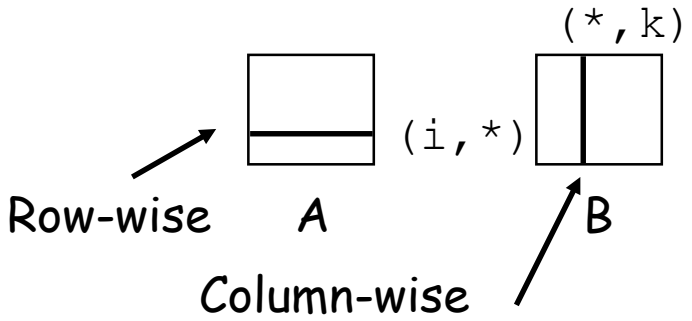
- 固定维数
  - 在编译的时N数值确定
- 可变维数,直接寻址
  - 传统方法是利用动态数组
- 可变维数,间接寻址
  - gcc支持

16*16矩阵

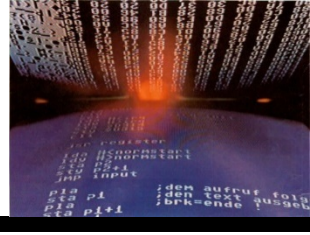
```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, int i, int j) {
  return a[i][j];
}
```

```
movl    12(%ebp), %edx   # i
sall    $6, %edx         # i*64
movl    16(%ebp), %eax   # j
sall    $2, %eax         # j*4
addl    8(%ebp), %eax    # a + j*4
movl    (%eax,%edx), %eax     # *(a + j*4 + i*64)
```

- **数组元素**
  - 地址：**A** + $i * (C * K) + j * K$
  - C = 16, K = 4



n X n Matrix Access



```
/* Get element a[i][j] */
int var_ele(int n, int a[n][n], int i, int j) {
  return a[i][j];
}
```

```
  movl   8(%ebp), %eax      # n
  sall   $2, %eax           # n*4
  movl   %eax, %edx         # n*4
  imull  16(%ebp), %edx     # i*n*4
  movl   20(%ebp), %eax     # j
  sall   $2, %eax           # j*4
  addl   12(%ebp), %eax     # a + j*4
  movl   (%eax,%edx), %eax  # *(a + j*4 + i*n*4)
```

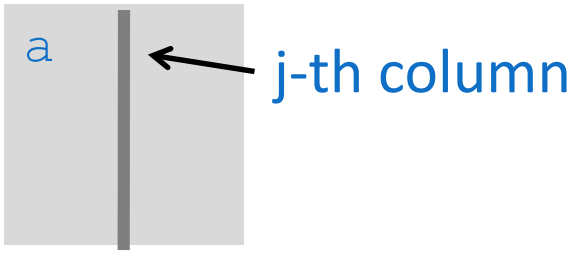
- **Array Elements**
  - Address **A** + $i * (C * K) + j * K$
  - C = n, K = 4

固定维数数组优化

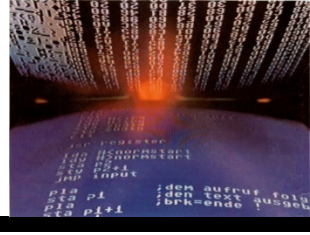
```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

- 所有元素排列在1行
- 优化：从1行中搜索元素



固定维数数组优化



- 优化
  - 计算 ajp = &a[i][j]
    - 初始化= a + 4*j
    - 增量 4*N

Register	Value
%ecx	ajp
%ebx	dest
%edx	i

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

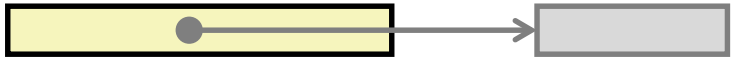
在下标运算中避免了乘法

```
.L8:                              # loop:
   movl  (%ecx), %eax             #   Read *ajp
   movl  %eax, (%ebx,%edx,4)      #   Save in dest[i]
   addl  $1, %edx                 #   i++
   addl  $64, %ecx                #   ajp += 4*N
   cmpl  $16, %edx                #   i:N
   jne   .L8                      #   if !=, goto loop
```

指针

- 指针是C语言的一个重要特征，帮助程序员避免寻址错误。
- 以统一的方式对不同数据结构中的元素产生引用。

```
int *ip;char *cpp
```

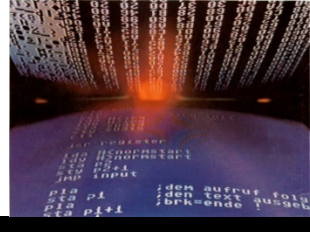
IP 

- 每个指针都对应一个类型，每个指针都有一个值，是某个指定类型对象的地址。特殊NULL(0)表示指针没有指向任何地址。
- 指针用&运算符创建，操作符用于指针的间接引用
- 数组和指针可以紧密联系

```
int a[10]; int *ptr=&a[0];
```

```
&a[i]=ptr+i
a[i]=*(ptr+i)
```

指针代码



- Add3函数建立指针，并通过incrk函数赋值 incrk

Generating Pointer

```
/* Compute x + 3 */
int add3(int x) {
   int localx = x;
   incrk(&localx, 3);
   return localx;
}
```

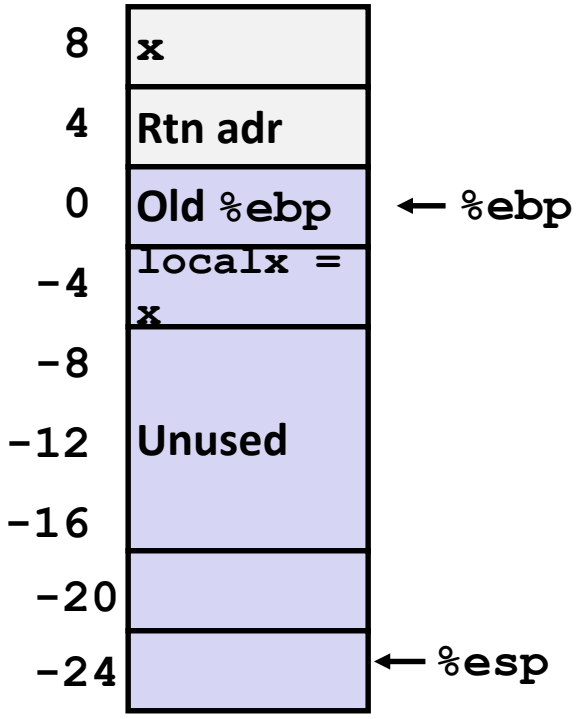
Referencing Pointer

```
/* Increment value by
k */
void incrk(int *ip,
int k) {
   *ip += k;
}
```

创建 & 初始化指针

Add3函数代码初始部分

```
add3:
   pushl  %ebp
   movl   %esp, %ebp
   subl   $24, %esp        # Alloc. 24 bytes
   movl   8(%ebp), %eax
   movl   %eax, -4(%ebp) # Set localx to x
```



- 用栈来保存局部变量
  - 变量 val必须要存于栈
    - 需要创建指针指向它
  - 计算指针-4(%ebp)
  - 作为第二个参数进栈

```
int add3(int x) {
   int localx = x;
   incrk(&localx, 3);
   return localx;
}
```

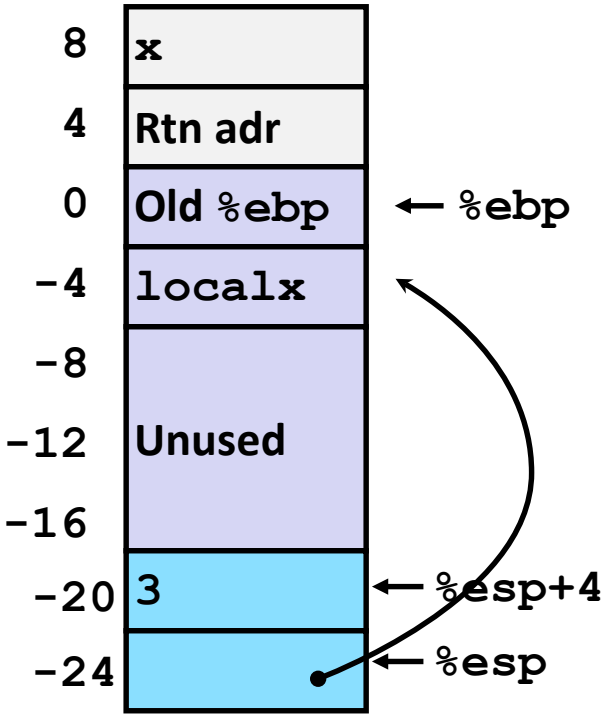
建立指针

Add3函数代码中间部分

```
movl $3, 4(%esp)   # 2nd arg = 3
leal -4(%ebp),%eax # &localx
movl %eax, (%esp)  # 1st arg = &localx
call incrk
```

- 通过leal指令变量localx的地址

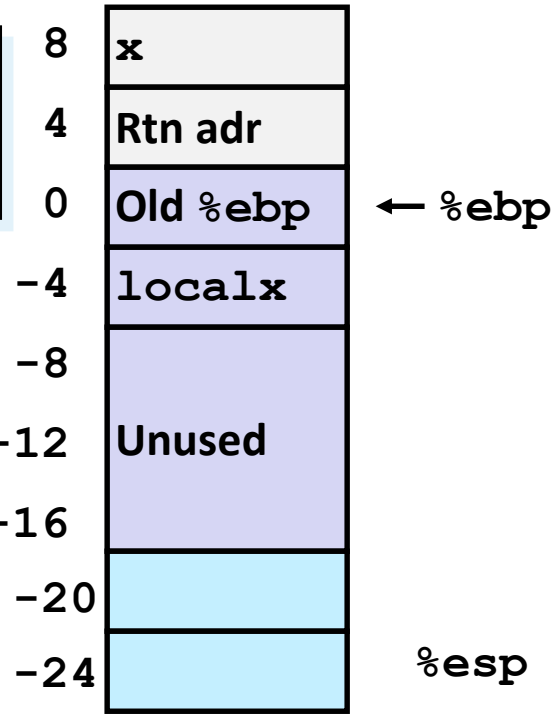
```
int add3(int x) {
   int localx = x;
   incrk(&localx, 3);
   return localx;
}
```



恢复局部变量

Add3函数代码最后部分

```
movl -4(%ebp), %eax # Return val= localx
leave
ret
```

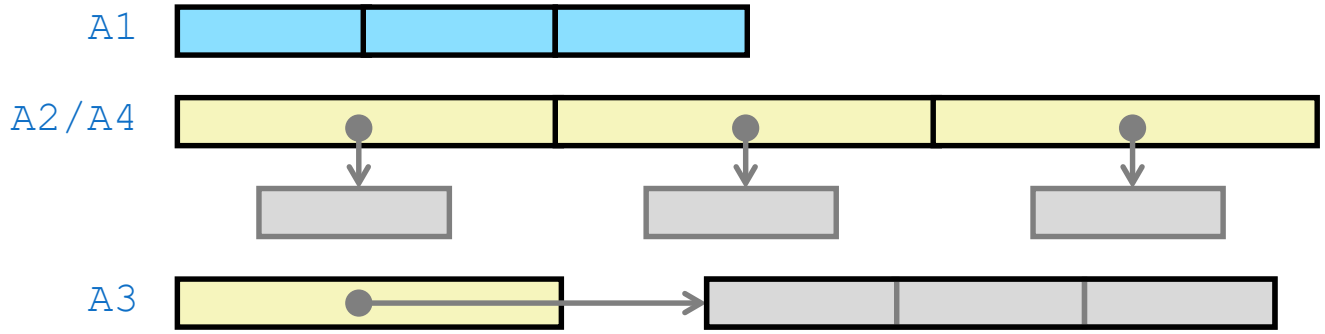


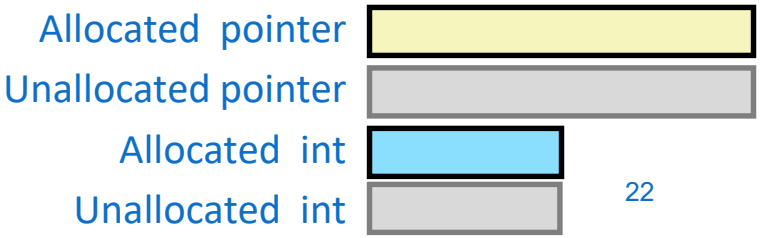
- 从堆栈中返回localx数值

```
int add3(int x) {
   int localx = x;
   incrk(&localx, 3);
   return localx;
}
```

指针与数组

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3]	Y	N	12	Y	N	4	N	–	–
int *A2[3]	Y	N	24	Y	N	8	Y	Y	4
int (*A3)[3]	Y	N	8	Y	Y	12	Y	Y	4
int (*A4[3])	Y	N	24	Y	N	8	Y	Y	4

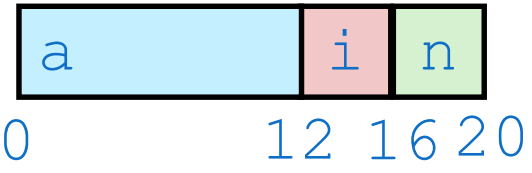




结构

```
struct rec {
   int a[3];
   int i;
   struct rec *n;
};
```

存储布局



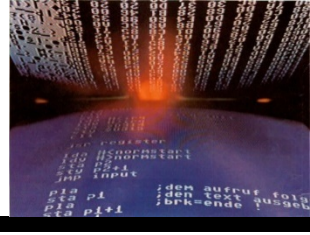
- 概念
  - 连续分配的存储空间
  - 通过名字引用结构成员
  - 成员可能是不同的类型
- 访问结构元素
  - 指针指向结构的第1个字节
  - 通过地址偏移来访问不同元素

```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

IA32汇编代码

```
# %edx = val
# %eax = r
movl %edx, 12(%eax)  # Mem[r+12] = val
```

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];
};
```

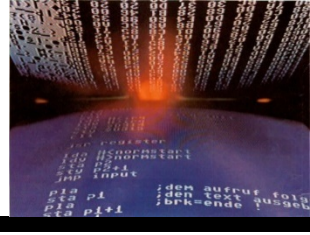
若变量x分配在地址0x8049200开始的区域，那么
x=&(x.id)=0x8049200 (若x在EDX中)
&(x.name)= 0x8049200+8=0x8049208
&(x.post)= 0x8049200+8+12=0x8049214
&(x.address)=0x8049200+8+12+4=0x8049218
&(x.phone)=0x8049200+8+12+4+100=0x804927C

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

按地址调用 **stu_phone1(&x)**

```
void stu_phone1 ( struct cont_info *s_info_ptr)
{
   printf (“%s phone number: %s”, (*s_info_ptr).name, (*s_info_ptr).phone);
}
```

按值调用 **stu_phone1(x)**

```
void stu_phone2 ( struct cont_info s_info)
{
   printf (“%s phone number: %s”, s_info.name, s_info.phone);
}
```

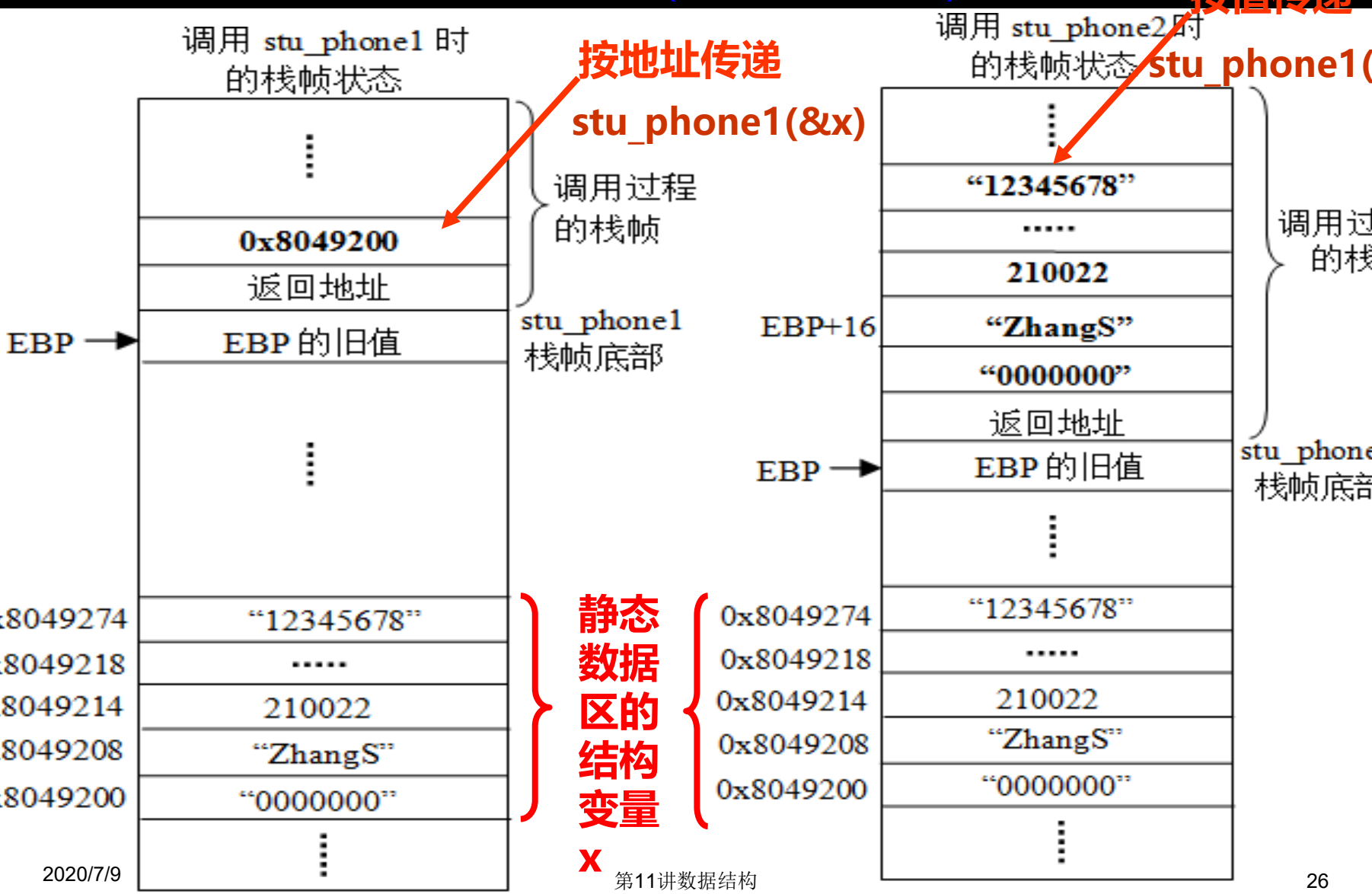
- 当结构体变量需要作为一个函数的形参时，形参和调用函数中的实参应具有相同结构

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

- 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(如前面的swap(a, b)例子)
- 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

结构体数据的分配和访问

- 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

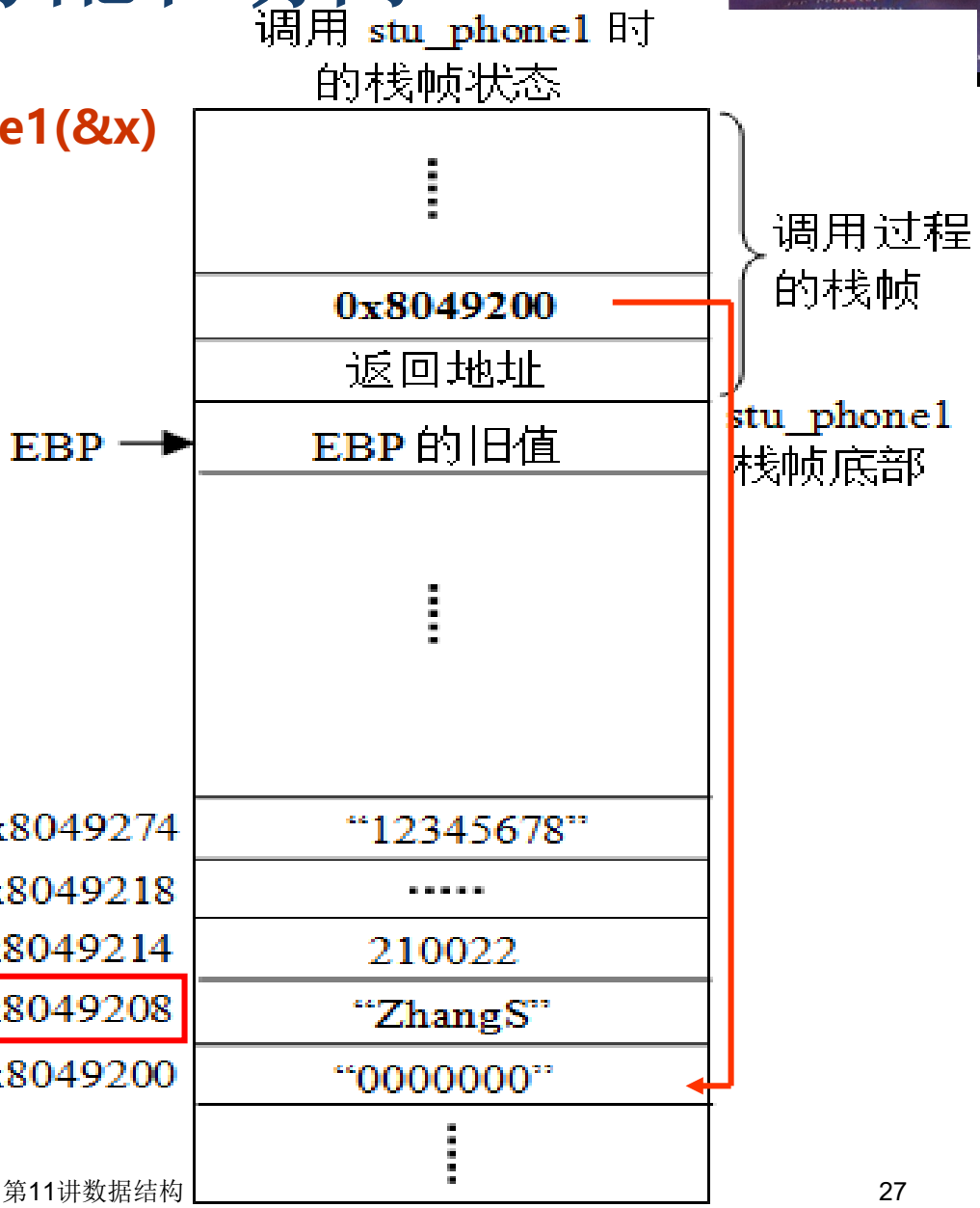
- 按地址传递参数 **stu_phone1(&x)**

(*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



结构体数据的分配和访问

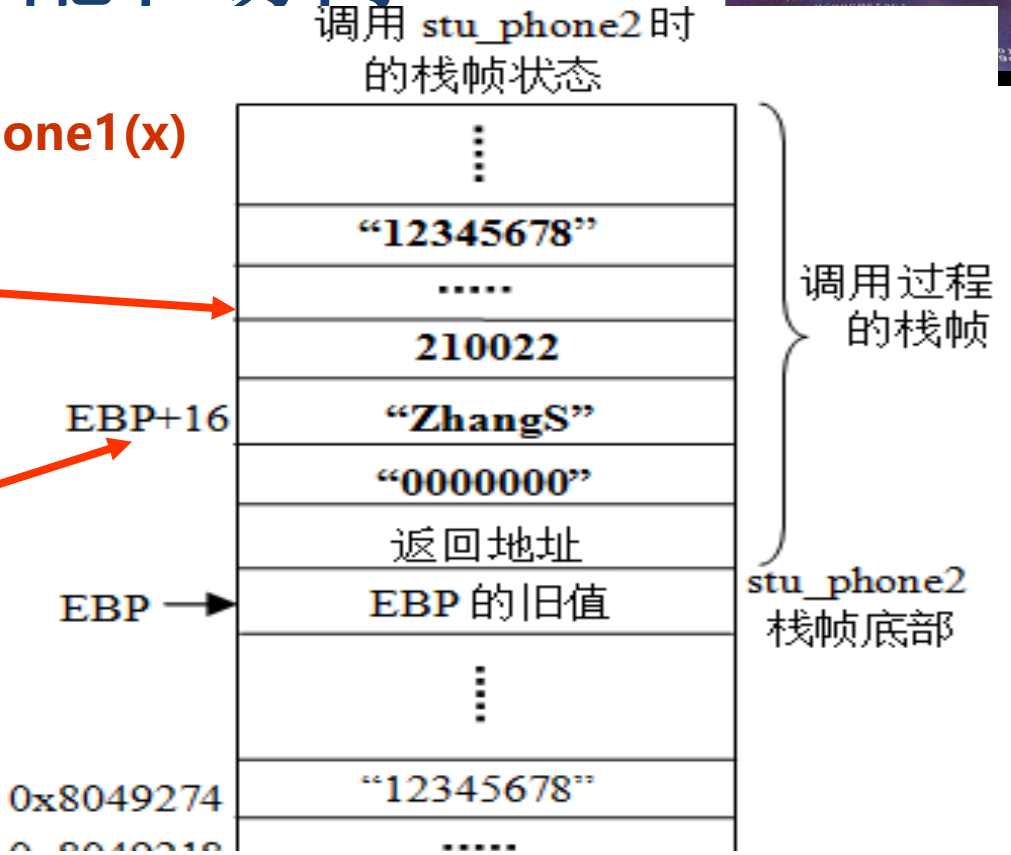
- 按值传递参数

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

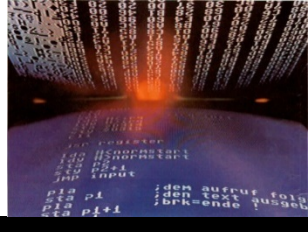
leal 8(%edx), eax

EAX中存放的是“ZhangS”的栈内参数区首址 。



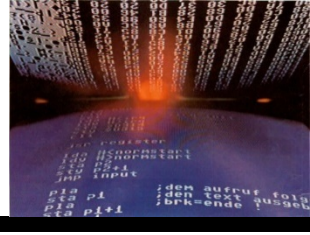
- **stu_phone1和stu_phone2功能相同，但后者开销大，因为它需对结构体成员整体从静态区复制到栈中，需要很多条mov或其他指令，从而执行时间更长，并占更多栈空间和代码空间**
- **特别是，按值传递时，无法获得更新后的结果**

数据对齐



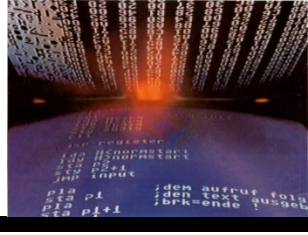
- 数据对齐：目的提升系统效率
  - 基本数据类型需要K字节，则数据对应的地址必须是从K的倍数开始
- 对齐数据的目的
  - 存储器访问都是通过对齐的双字或者四字
    - 如果数据跨越了四字边界存取效率低下
    - 虚存中数据跨越两页访问情况会很复杂
- 编译器
  - 在结构中插入一些空隙来确保正确的域对齐
- IA32 Linux 、X86-64 Linux与 Windows处理不同!

IA-32中数据对齐



- 1个字节的数据类型 (如char)：在初始地址上无限制
- 2个字节的数据类型 (如 short):地址的末位必须是0_2，即是2的倍数。
- 4个字节的数据类型 (如 int, float, char *, etc.)
  - 地址的末2位必须是00_2，即是4的倍数。
- 8个字节的数据类型 (如 double)
  - Windows要求：**最低3位地址必须是000_2**，即是8的倍数
  - Linux要求:**最低2位地址必须是00_2，与4字节同样处理**
- 12个字节的数据类型 (long double)
  - Windows,Linux:**最低2位地址必须是00_2，与4字节基本数据类型同样处理**

X86-64中数据对齐



- 1 byte: `char, ...`
  - 在地址上无限制
- 2 bytes: `short, ...`
  - 最低1位地址必须是0_2
- 4 bytes : `int, float, char *, ...`
  - 最低2位地址必须是00_2
- 8 bytes: `double,char*, ...`
  - Windows,Linux:
    - **最低3位地址必须是000_2**
- 16 bytes :`long double, ...`
  - Linux:
    - **最低3位地址必须是000_2**
    - **i.e.,与8字节基本数据类型同样处理**

数据的对齐

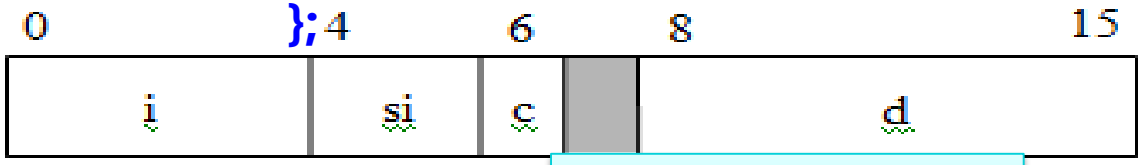
- CPU访问主存时只能一次读取或写入若干特定位。例如，若每次最多读写64位，则第0字节到第7字节可同时读写，第8字节到第15字节可同时读写，……，以此类推。
- 按边界对齐，可使读写数据位于8i~8i+7(i=0,1,2,…) 单元 。
- 最简单的对齐策略是，按其数据长度进行对齐，例如，int型地址是4的倍数，short型地址是2的倍数，double和long long型的是8的倍数，float型的是4的倍数，char不对齐。Windows采用该策略。Linux策略更宽松：short是2的倍数，其他如int、double和指针等都是4的倍数。

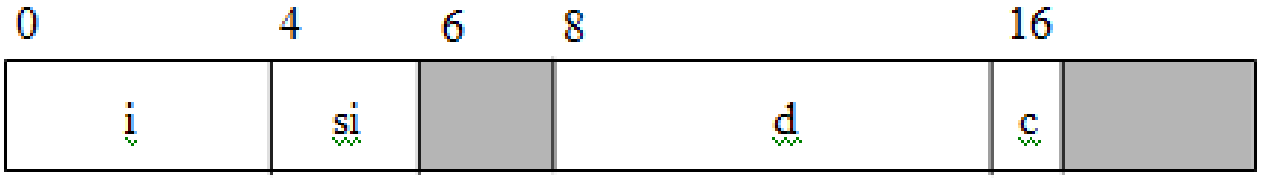
```
struct SDT {
    int     i;
    short   si;
    double  d;
    char    c;
} sa[10];
```

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;
};
```

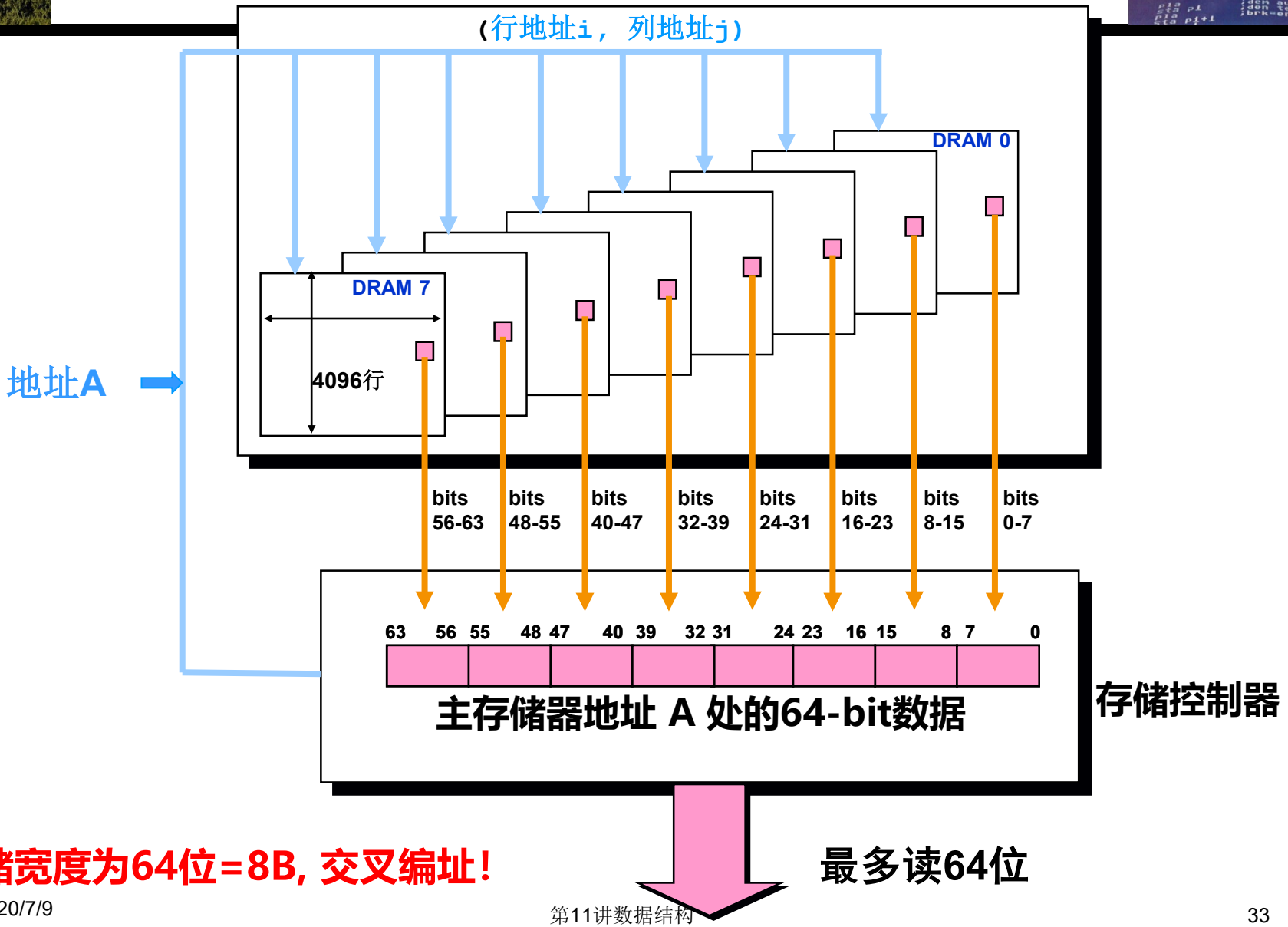
结构变量首地址按4字节边界对齐





只要SD首址按4B边界对齐，所有字段都能按要求对齐

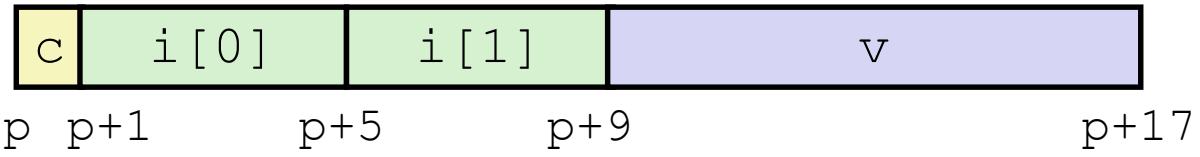
主存储器的结构



存储宽度为64位=8B，交叉编址！

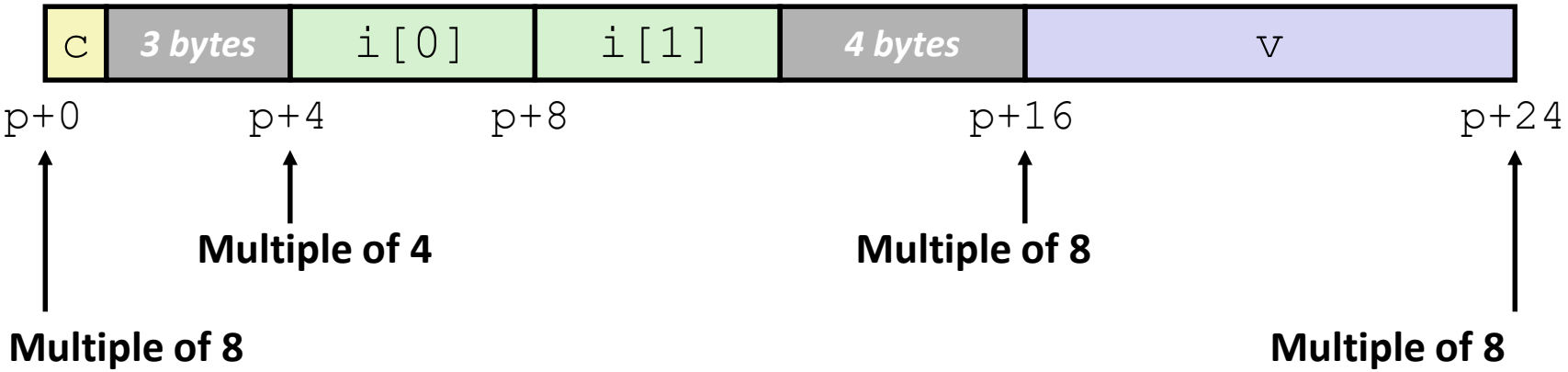
结构与数据对齐

- 不对齐数据分配




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

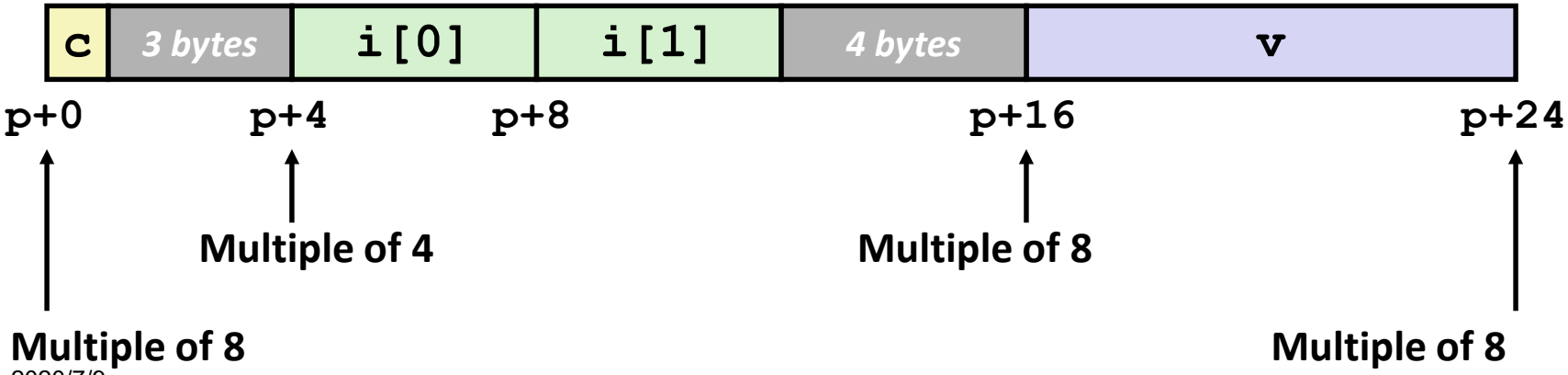
- 对齐数据分配
  - 原始数据类型需要***K*** 字节
  - 地址必须分配 ***K*** 的倍数开始



在结构中满足对齐

- 结构中的偏移量
  - 必须满足元素的对齐要求
- 整个结构放置
  - 每个结构有对齐要求K
    - 成员中最大的对齐要求
  - 起始地址& 结构长度必须是K的倍数
- 例子 (在Windows或x86-64下):
  - K = 8, 由于double 元素

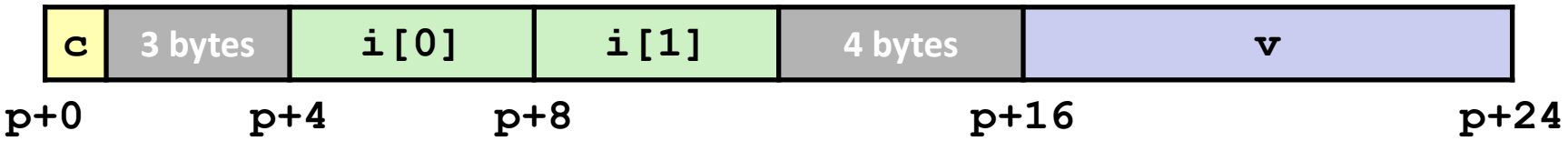
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



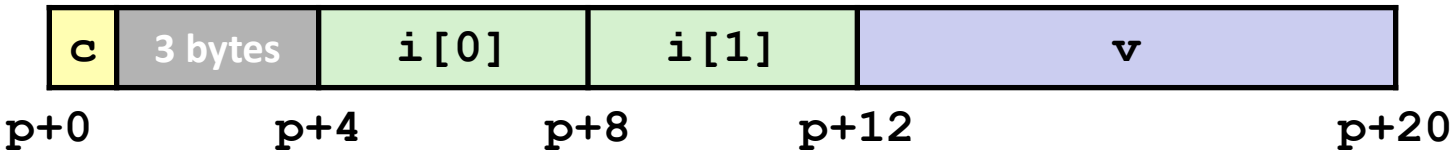
Linux vs. Windows

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- x86-64 or IA32 Windows :
  - K = 8, 由于double 元素



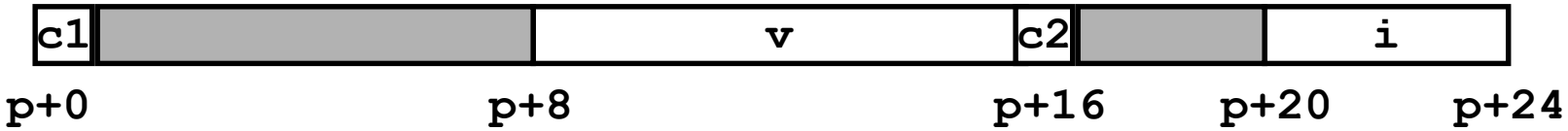
- IA32 Linux:
  - K = 4; double与4字节数据类型同样对待



结构内的元素顺序

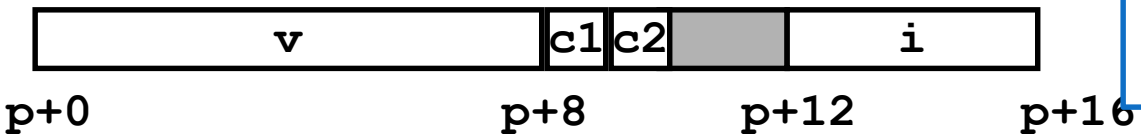
```
struct S4 {
   char c1;
   double v;
   char c2;
   int i;
} *p;
```

在Windows中10个字节浪费掉了




```
struct S5 {
   double v;
   char c1;
   char c2;
   int i;
} *p;
```

2个字节浪费掉了



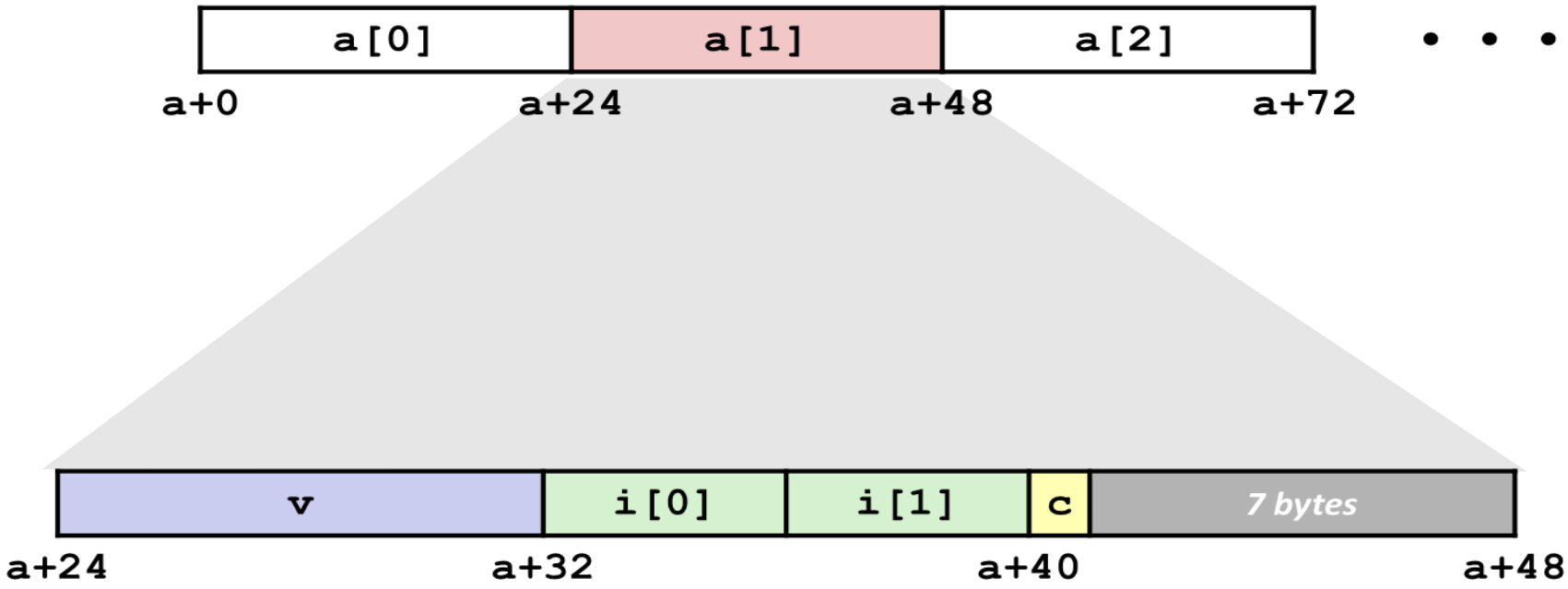
有没有不浪费字节的方法？

按类型长度大小排序，从大到小顺序定义。

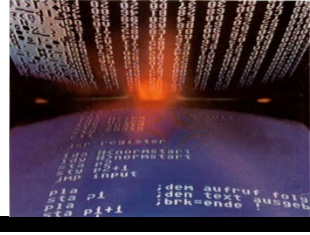
结构数组

- 全部结构长度是 K 倍数
- 每一个元素满足对齐要求

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



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）

问题：float2unsign(10.0)=? $2^{30}+2^{24}+2^{21}=1092616192$?

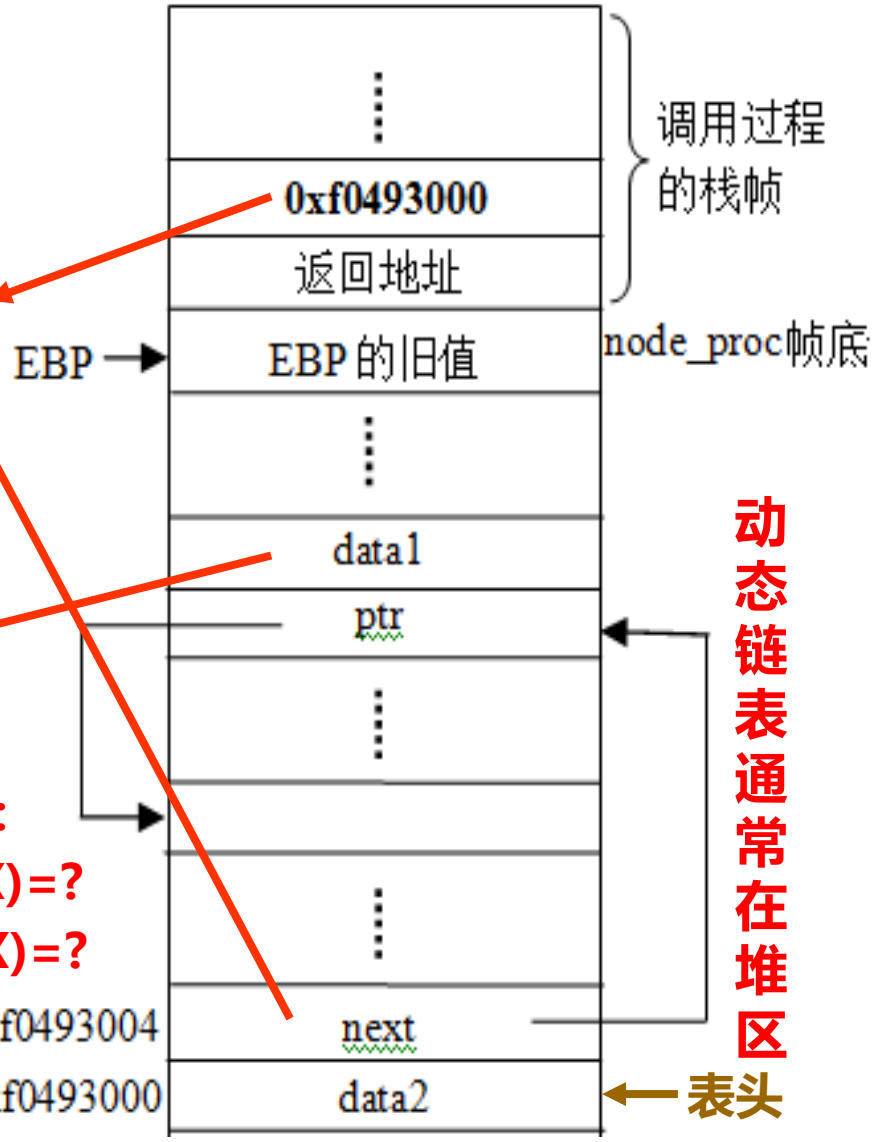
从该例可看出：机器级代码并不区分所处理对象的数据类型，不管高级语言中将其说明成float型还是int型或unsigned型，都把它当成一个0/1序列来处理。

联合体数据的分配

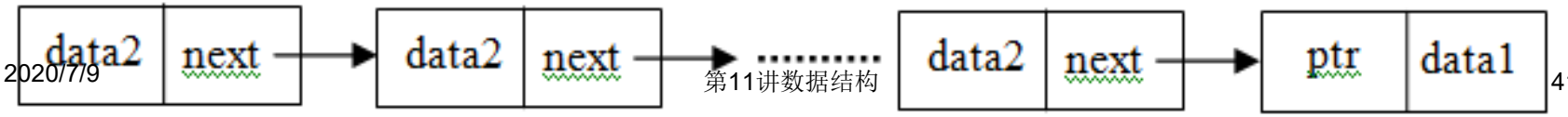
- 利用嵌套可定义链表结构

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

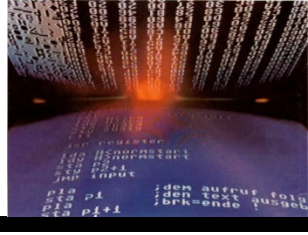
```
movl  8(%ebp), %ecx
movl  4(%ecx), %edx
movl  (%edx), %eax
movl  (%eax), %eax
addl  (%ecx), %eax
movl  %eax, 4(%edx)
```




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



总结



- C中数组
  - 连续的存储空间
  - 指向最开始元素
  - 无边界检测
- 结构
  - 按照声明顺序分配字节
  - 为了满足对齐在中间和最后填充空隙
- 联合
  - 叠加声明
  - 绕过类型系统的一种方法