程序的机器级表示

- 分以下五个部分介绍
  - 第一讲： x86-64指令系统
    - 高级语言程序转换为机器代码的过程
    - 机器指令和汇编指令
    - x86-64指令系统
  - 第二讲： C语言程序的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
    - 过程调用的机器级表示
  - 第三讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第四讲：越界访问和缓冲区溢出
  - 第五讲：浮点指令和代码

围绕C语言中的语句和**复杂数据类型**，解释其在底层机器级的实现方法

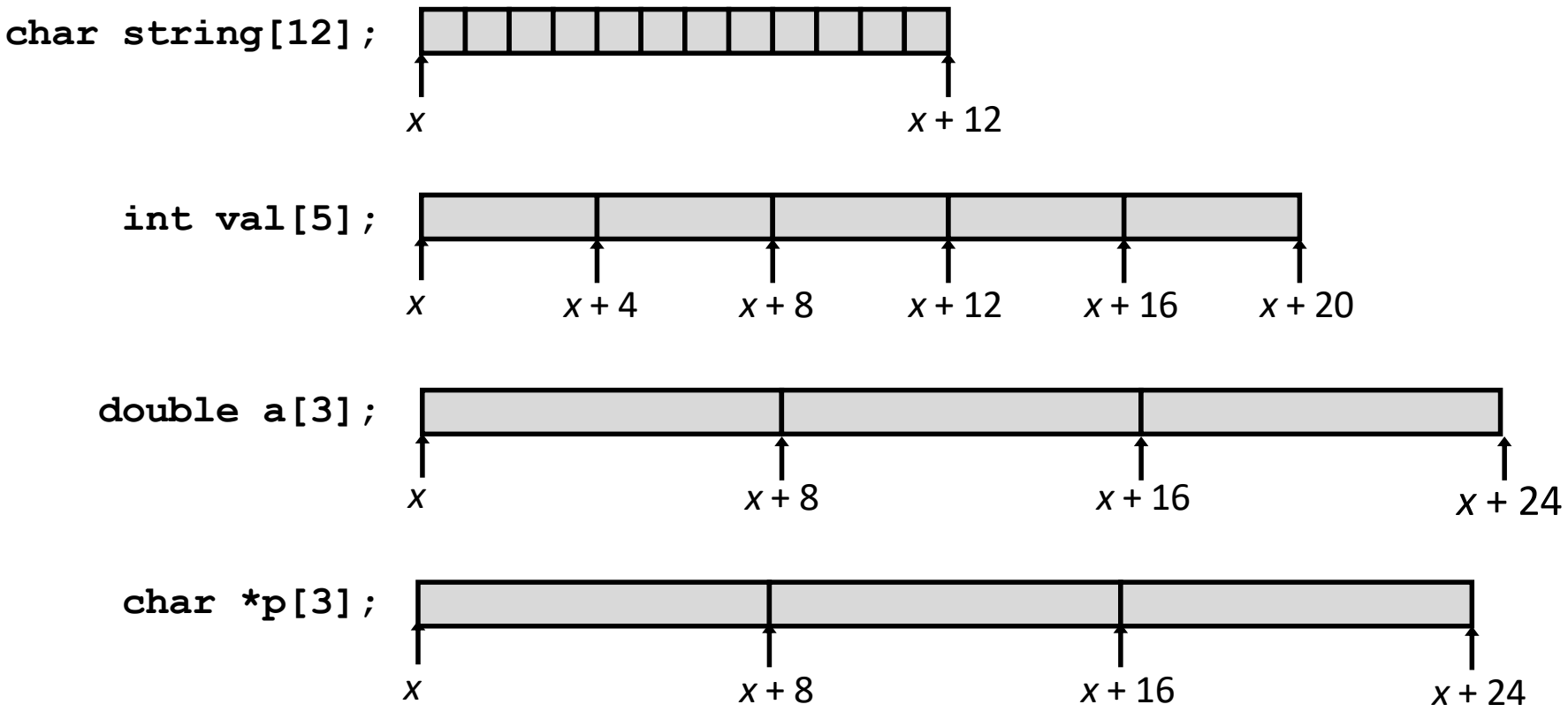
数组的分配和访问

- 数组元素在内存的存放和访问 基本原则：

  T **A[***L***];**

  – 数组的数据类型 *T*，长度 *L*

  – 内存中分配 *L* * **sizeof**(*T*) 字节的连续区域



数组的分配和访问

T **A[***L***];**

- 用标识符A来作为指向数组开头的指针：T* 类型
- 用0~L-1的整数索引来访问该数组元素
- 第 i（0≤i≤L-1）个元素的地址计算公式：&A[0]+sizeof(T)*i
- 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

A=&A[0]

T *p=A;

&A[i]=A+i=p+i

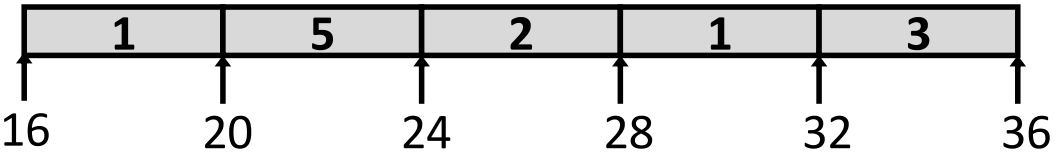
A[i]=*(A+i)=*(p+i)

计算出来的值会根据该指针引用的数据类型的大小进行伸缩。

数组的分配和访问


```
#define ZLEN 5
typedef int zip_dig[ZLEN];
zip_dig cmu = { 1, 5, 2, 1, 3 };
```

zip_dig cmu;




```
int get_digit
(zip_dig z, int digit)
{
  return z[digit];
}
```

汇编指令

```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

- 寄存器 %rdi存放是数组的起始地址
- 寄存器 %rsi 存放的是数组索引

有效地址%rdi + 4*%rsi

- 存储器寻址:
  (%rdi,%rsi,4)

数组的分配和访问


```
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax            #   i = 0
  jmp     .L3                 #   goto middle
.L4:                          # loop:
  addl    $1, (%rdi,%rax,4)   #   z[i]++
  addq    $1, %rax            #   i++
.L3:                          # middle
  cmpq    $4, %rax            #   i:4
  jbe     .L4                 #   if <=, goto loop
  rep; ret
```

数组的分配和访问

问题： 假定数组A的首址SA在rcx中，i在rdx中，表达式结果在eax或rax中，各表达式的计算方式以及汇编代码各是什么?

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

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”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。

假设A首址SA在rcx，i 在rdx，结果在eax或rax

序号	表达式	类型	值的计算方法	汇编代码
1	A	int *	SA	leaq (%rcx),%rax
2	A[0]	int	M[SA]	movl (%rcx),%eax
3	A[i]	int	M[SA+4*i]	movl (%rcx,%rdx,4),%eax
4	&A[3]	int *	SA+12	leaq 12(%rcx),%rax
6	*(A+i)	int	M[SA+4*i]	movl (%rcx,%rdx,4),%eax
7	*(&A[0]+i-1)	int	M[SA+4*i-4]	movl -4(%rcx,%rdx,4),%eax
8	A+i	int *	SA+4*i	leaq (%rcx,%rdx,4),%rax

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”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 。

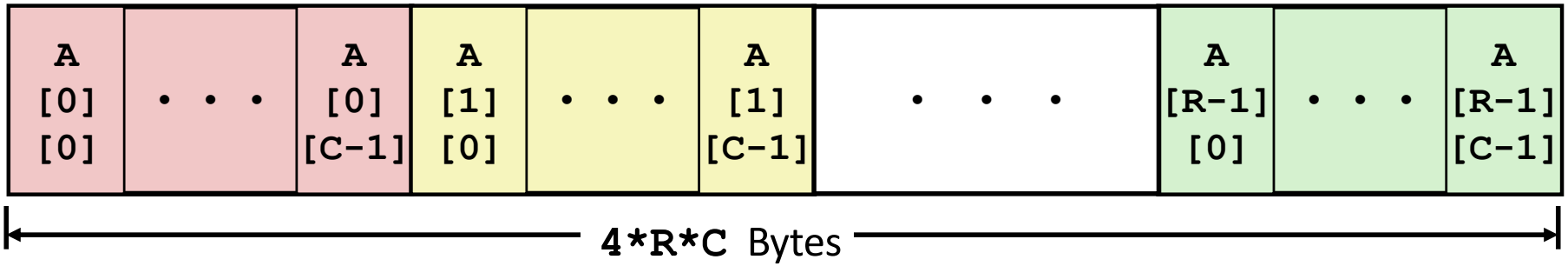
嵌套数组的分配和访问

- *T* **A**[*R*][*C*];
  - 2D 数组，数据类型 *T*
  - *R* 行, *C* 列
  - Type *T* 每个元素 *K* bytes

$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

- 数组大小
  - *R* * *C* * *K* bytes

```
int A[R][C];
```

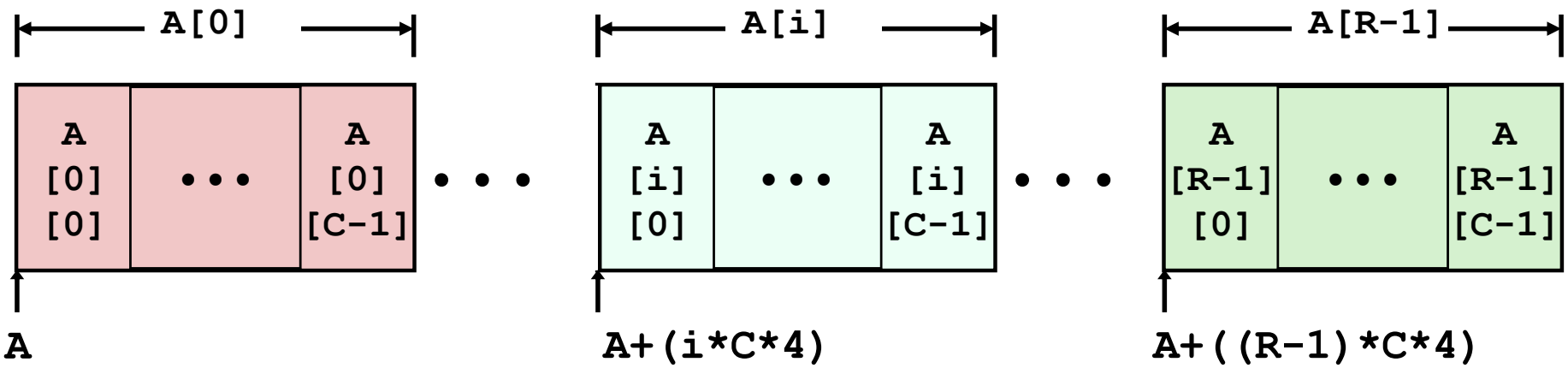


存放规律：行优先

嵌套数组的分配和访问

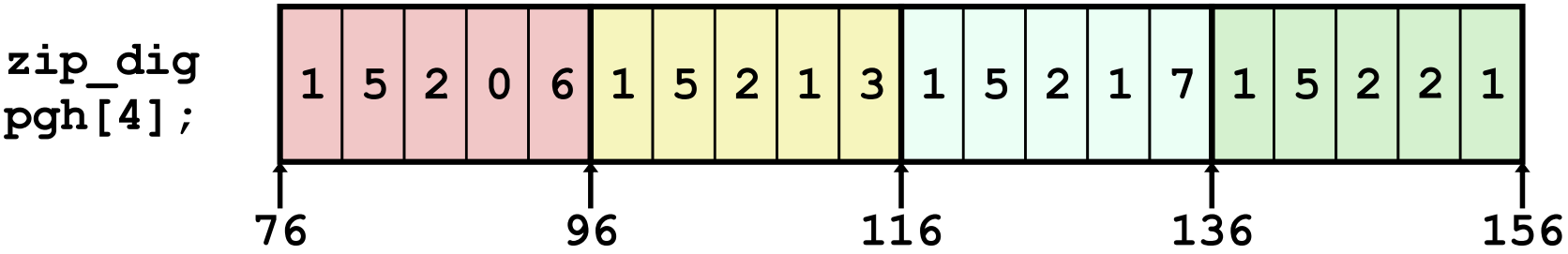
- 行向量
  - **A[i]** 是 *C* 个元素组成的数组
  - 每个元素类型为 *T*， *K* 字节
  - 起始地址： **A** + *i* * (*C* * *K*)

```
int A[R][C];
```



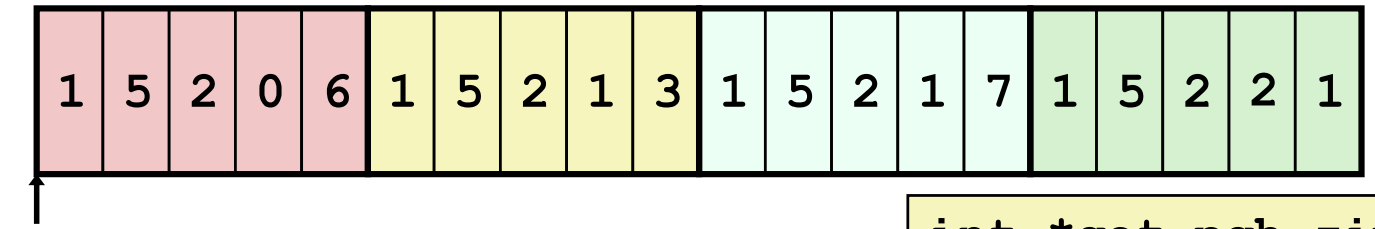
嵌套数组的分配和访问


```
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



- “zip_dig pgh[4]”等价“int pgh[4][5]”
  - 变量 **pgh**: 4个元素的数组
  - 每个元素是一个5个整数的数组
  - **存放规律：行优先**

嵌套数组的分配和访问




```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

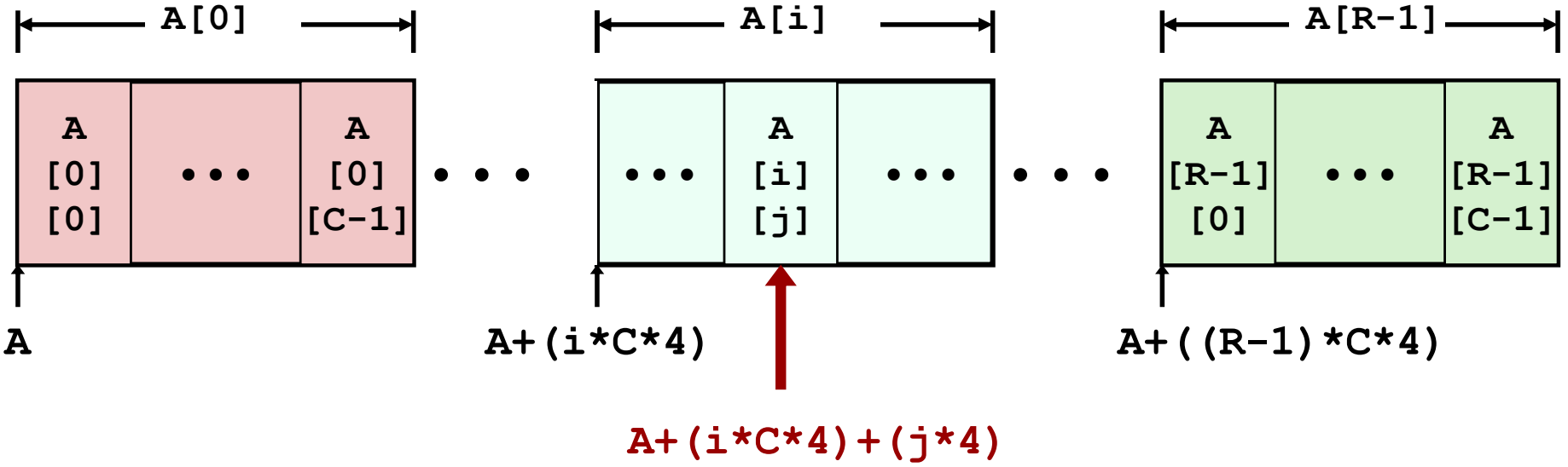
```
# %rdi = index
 leaq (%rdi,%rdi,4),%rax # 5 * index
 leaq pgh(,%rax,4),%rax  # pgh + (20 * index)
```

- 行向量
  - **pgh[index]** 是5个int类型数据组成的数组
  - 地址 **pgh+20*index**
- 机器代码
  - 计算并返回地址： **pgh + 4*(index+4*index)**

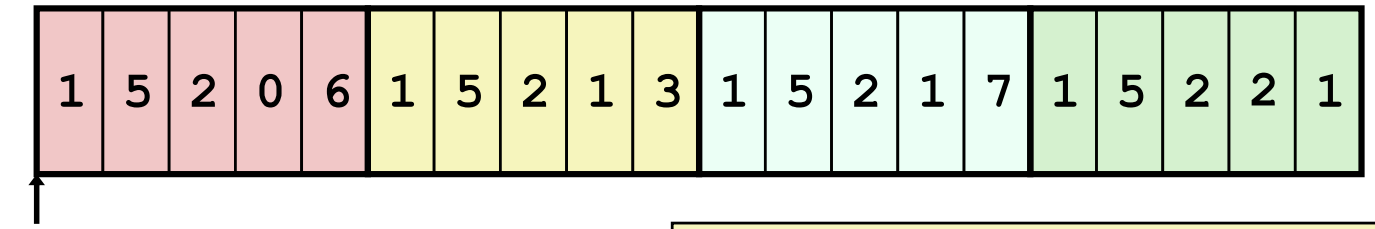
嵌套数组的分配和访问

- 数组元素
  - **A[i][j]**是类型为 *T*，*K* 字节的元素
  - 元素地址 **A** + *i* * (*C* * *K*) + *j* * *K* = *A* + (*i* * *C* + *j*) * *K*

```
int A[R][C];
```



嵌套数组的分配和访问




```
int get_pgh_digit
(int index, int dig)
{
  return pgh[index][dig];
}
```

```
leaq   (%rdi,%rdi,4), %rax    # 5*index
addl   %rax, %rsi             # 5*index+dig
movl   pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

- 数组元素
  - **pgh[index][dig]** is **int**
  - Address: **pgh + 20*index + 4*dig**

    = **pgh + 4*(5*index + dig)**

n X n 矩阵

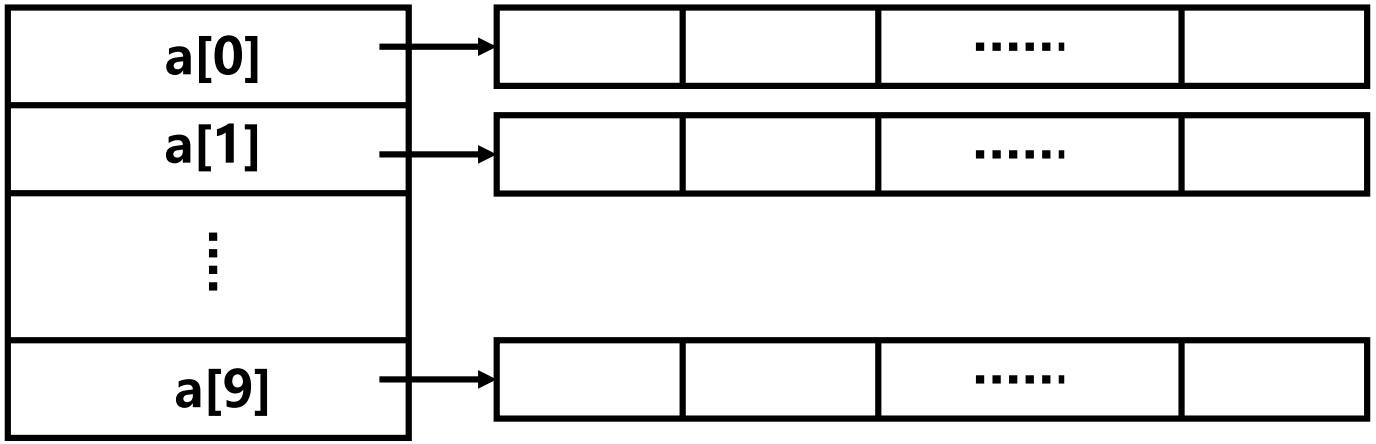
- 数组元素
  - 地址 **A** + *i* * (*C* * *K*) + *j* * *K*
  - C = n, K = 4
  - Must perform integer multiplication

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

```
  # n in %rdi, a in %rsi, i in %rdx, j in %rcx
  imulq   %rdx, %rdi              # n*i
  leaq    (%rsi,%rdi,4), %rax     # a + 4*n*i
  movl    (%rax,%rcx,4), %eax     # a + 4*n*i + 4*j
  ret
```

- 指针数组
  - 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：

    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。



- 指针数组
  - 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+8*i]=M[0x8049318]=0x8049308

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

```
movq   pn(,%rcx,8), %rdx
addw   (%rdx), %ax
addq   $2, pn(, %rcx, 8)
```

pn[i]+1→pn[i]

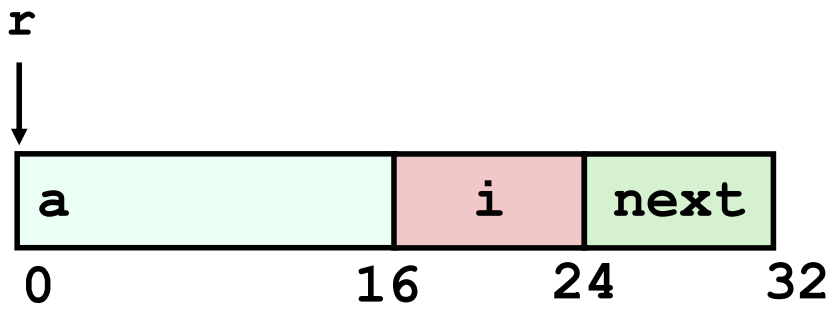
若num=0x8049300,则num、pn和s在存储区中如何存放?

```
08049300 <num>:   num=num[0]=&num[0][0]=0x8049300
08049300:  02 00 09 00 ff ff 05 00 03 00 08 00 02 00 fa ff
08049310 <pn>:
08049310:  00 93 04 08 00 00 00 00 08 93 04 08 00 00 00 00
08049320<s>:
08049320:  00 00 00 00
```

pn=&pn[0]=0x8049310
pn[0]=num[0]=0x8049300
pn[1]=num[1]=0x8049308

结构体的分配和访问

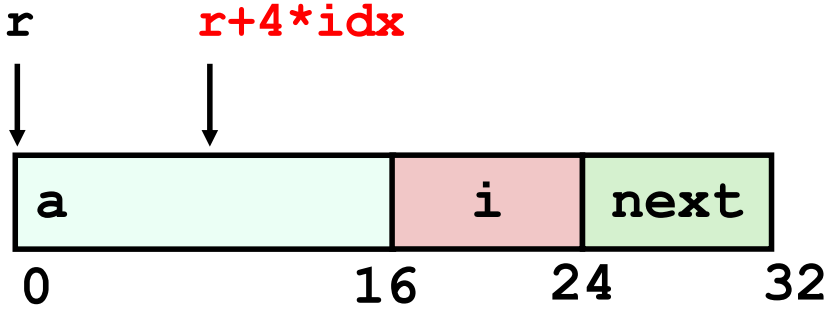

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
```



- 结构体的所有字段都存放在内存中一段连续的区域内
- 各字段存放顺序按照声明顺序
- 编译器决定各字段的大小和位置

结构体的分配和访问

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
```



- 指向结构体的指针就是结构体第一个字节的地址

```
int *get_ap
(struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```

```
  # r in %rdi, idx in %rsi
  leaq   (%rdi,%rsi,4), %rax
  ret
```

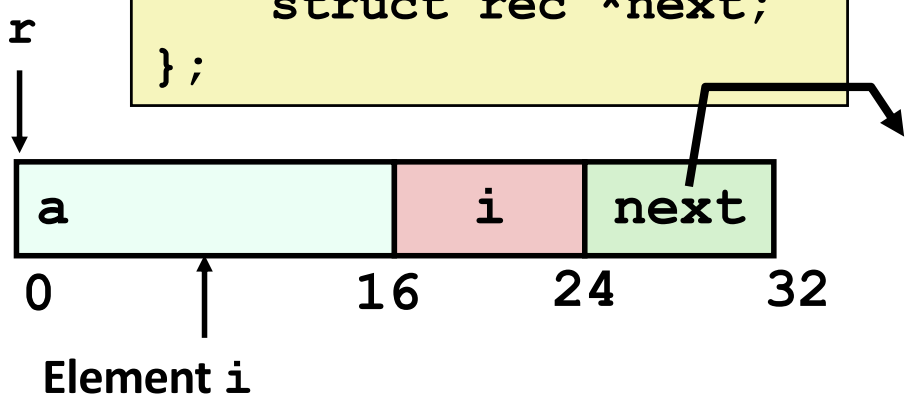
机器代码不包含关于字段声明或字段名字的信息。

结构体的分配和访问

- C Code

```
void set_val
(struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
```



Register	Value
%rdi	r
%rsi	val

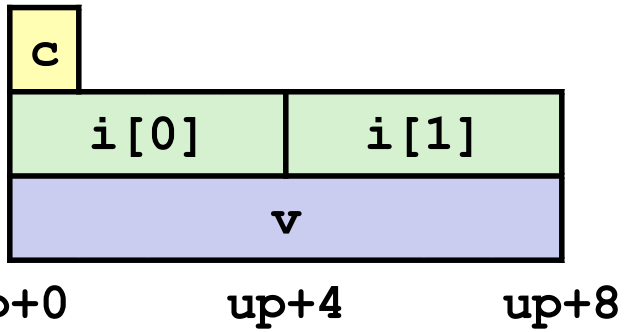
```
      jmp     .L22                  # loop:
.L11  movq    16(%rdi), %rax        #   i = M[r+16]
      movl    %esi, (%rdi,%rax,4)   #   M[r+4*i] = val
      movq    24(%rdi), %rdi        #   r = M[r+24]
.L22: testq   %rdi, %rdi            #   Test r
      jne     .L11                  #   if !=0 goto loop
```

联合体数据的分配和访问

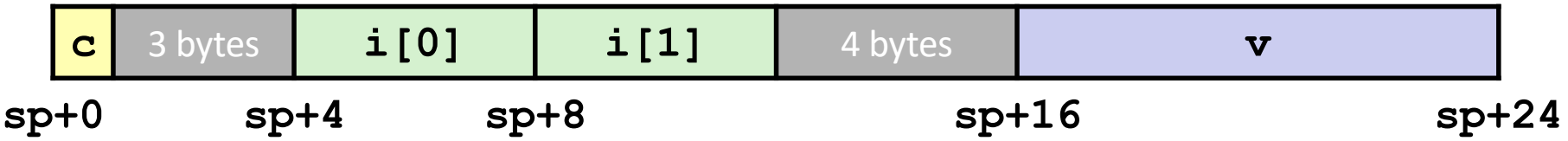
按不同类型引用同一个对象。

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 U1 {
  char c;
  int i[2];
  double v;
} *up;
```




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



联合体数据的分配和访问


```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

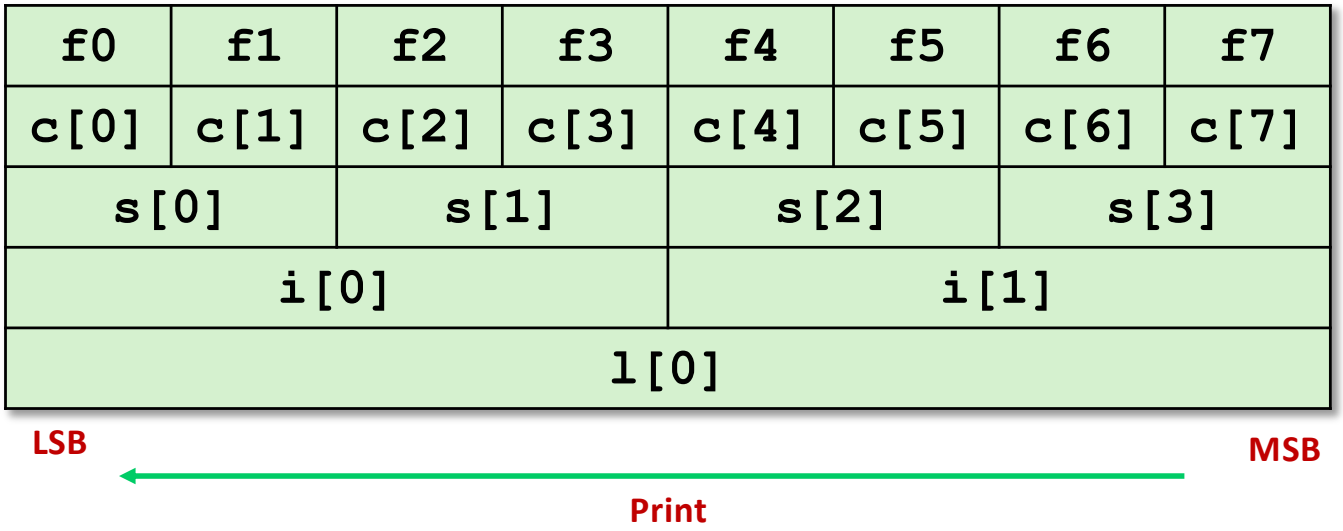
64-bit

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

联合体数据的分配和访问

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

小端法

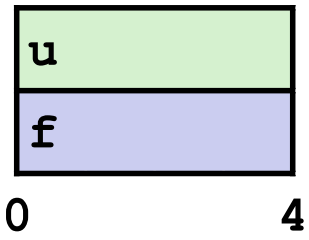


Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

联合体数据的分配和访问

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```



位拷贝

```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as (float)u?　数值转换　Same as (unsigned)f?

联合体数据的分配和访问

- **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。**

例：实现一个二叉树的数据结构

```
struct node_s {
  struct node_s *left;
  struct node_s *right;
  double data[2];
} *n;
```

32个字节

```
union node_u {
  struct {
    union node_u *left;
    union node_u *right;
  } internal;
  double data[2];
} *n;
```

16个字节

叶子节点：

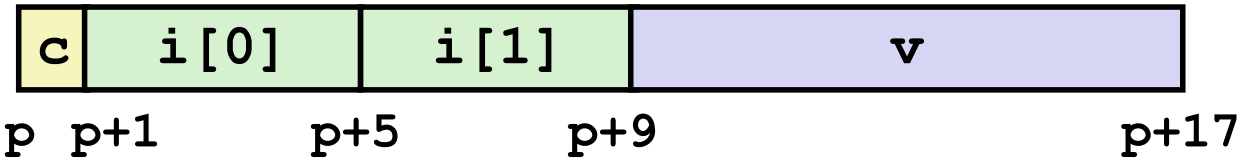
存储数据 n->data[0] 和 n->data[1]

内部节点：

存储孩子 n->internal.left 和 n-> internal.right

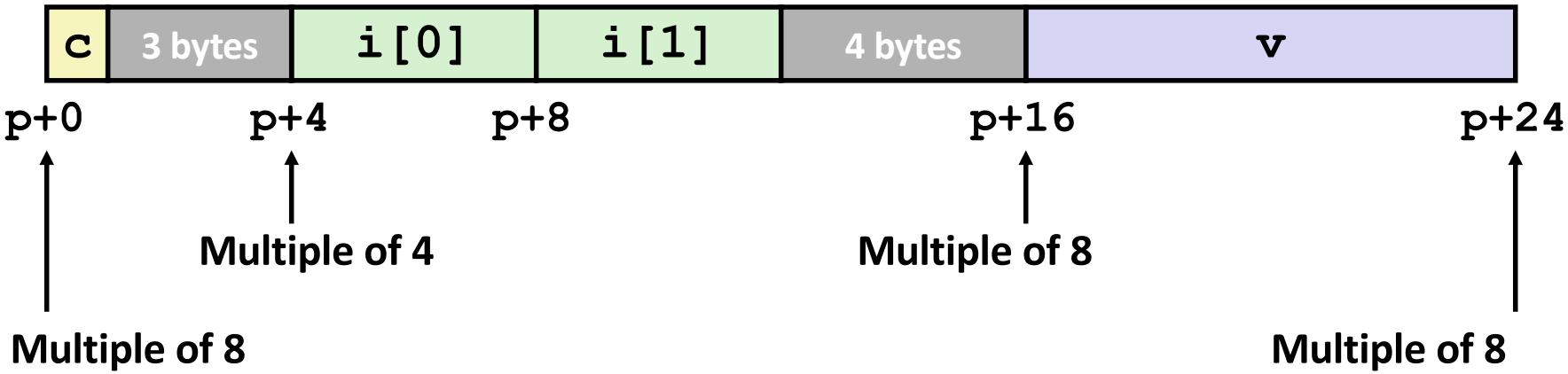
数据对齐

- 数据未对齐




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

- 数据对齐
  - 某种类型的数据的存储地址必须是K的倍数
  - K是数据类型存储字节数



数据的对齐

对齐限制简化了处理器和内存系统之间接口的硬件设计

- CPU访问主存时只需一次读取或写入若干特定位
  - 例如，若每次读写64位，则第0-7字节可同时读写，第8-15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,...) 单元内

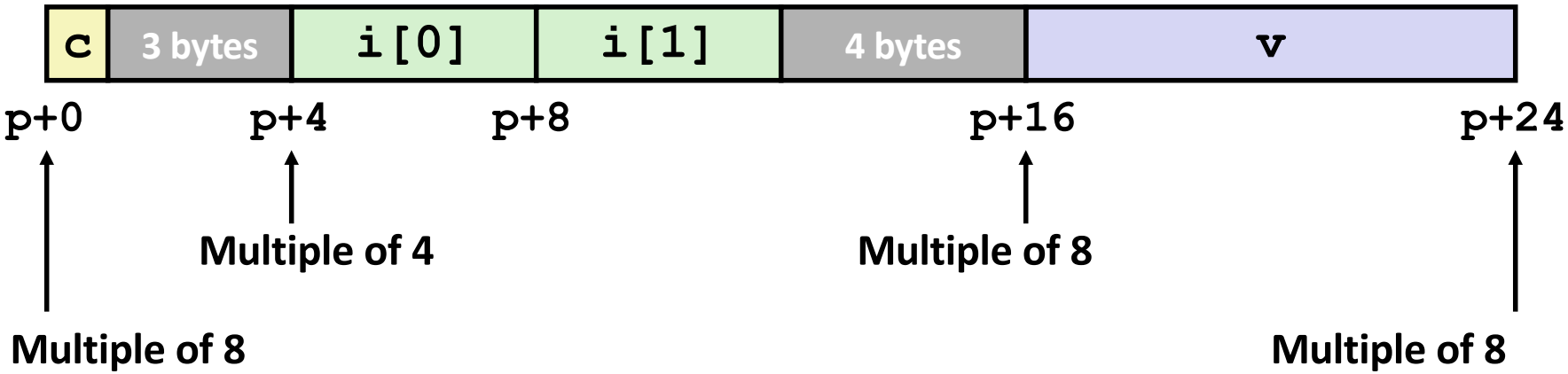
反例：int a存在[63,64,65,66]单元，读写a需要两次总线操作

K	类型T
1	char
2	short
4	int, float
8	long, double, T *

数据的对齐

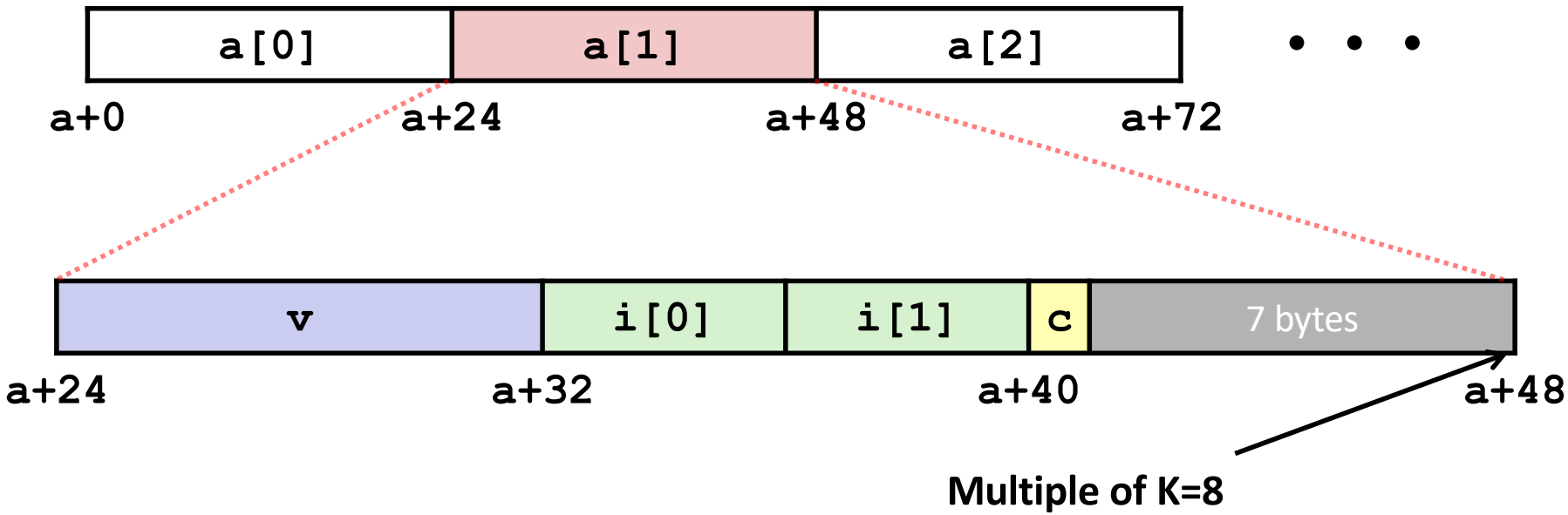
- 结构体内元素的对齐
  - 保证每个结构体的字段都满足对齐要求。
- 整个结构的对齐
  - 存放的地址和长度必须是K的倍数
    - K = Largest alignment of any element
- Example:
  - K = 8, due to **double** element

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



数据的对齐

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

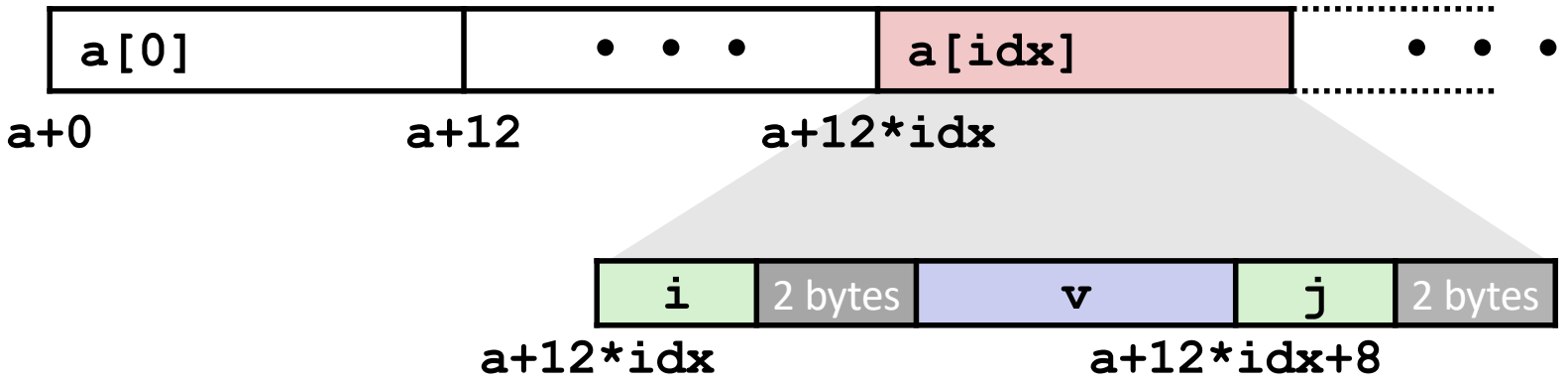


数据的对齐


```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

如果不填充最后2字节，即a[0]占a+0～a+9共10字节，那么&a[1].v=a+10+4=a+14，不是4的整数倍

所以，**需要保证a[i]占用字节数是其中最大字段所需字节数的整数倍**，最少填充2字节到共12字节




```
short get_j(int idx)  {
  return a[idx].j;
}
```

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

全局/静态数据区，基址a链接时确定

程序的机器级表示

- 分以下五个部分介绍
  - 第一讲： x86-64指令系统
    - 高级语言程序转换为机器代码的过程
    - 机器指令和汇编指令
    - x86-64指令系统
  - 第二讲： C语言程序的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
    - 过程调用的机器级表示
  - 第三讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第四讲：内存越界引用和缓冲区溢出
  - 第五讲：浮点指令和代码

内存越界引用和缓冲区溢出

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

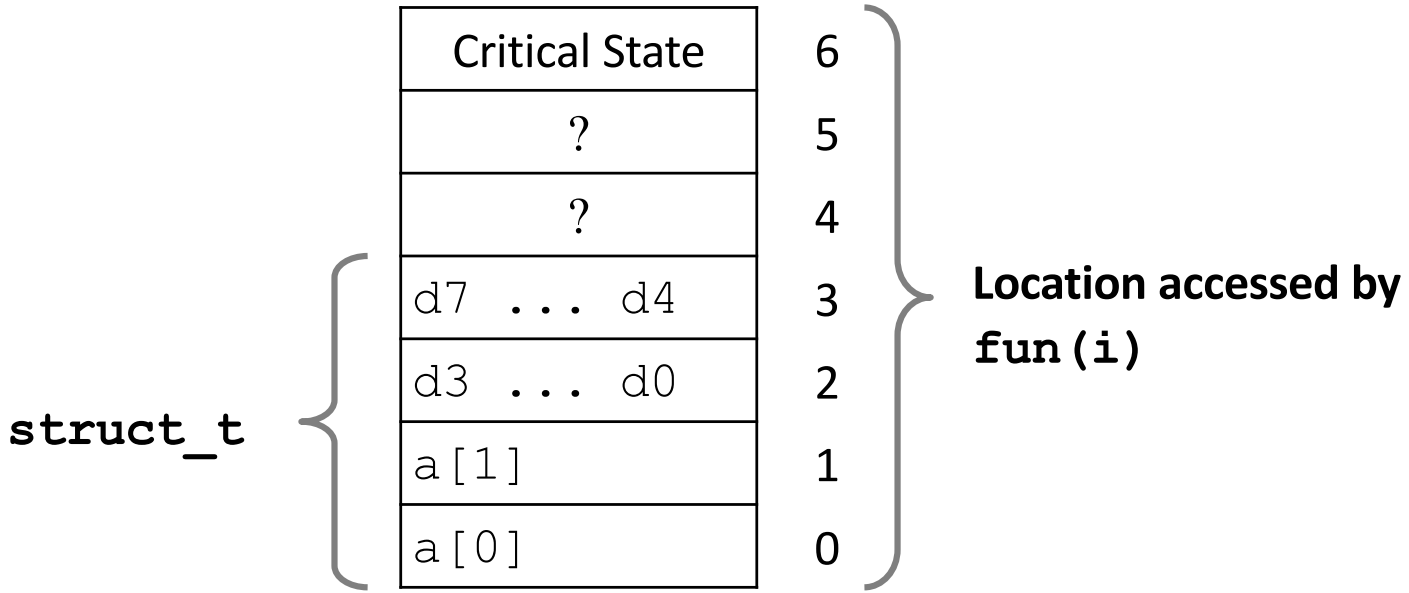
```
fun(0)   →   3.14
fun(1)   →   3.14
fun(2)   →   3.1399998664856
fun(3)   →   2.00000061035156
fun(4)   →   3.14
fun(6)   →   Segmentation fault
```

– Result is system specific

内存越界引用和缓冲区溢出

```
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0)  →  3.14
fun(1)  →  3.14
fun(2)  →  3.1399998664856
fun(3)  →  2.0000006103515625
fun(4)  →  3.14
fun(5)  →  3.14
fun(6)  →  Segmentation fault
```



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。

内存越界引用和缓冲区溢出

- C语言中的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，也即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
- C标准规定, 数组越界访问属于未定义行为, 访问结果是不可预知的
- 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
- 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行动。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

- Implementation of Unix function `gets()`

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

内存越界引用和缓冲区溢出

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901 2
0123456789012345678901 2
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123 4
Segmentation Fault
```

Buffer Overflow Disassembly

echo:

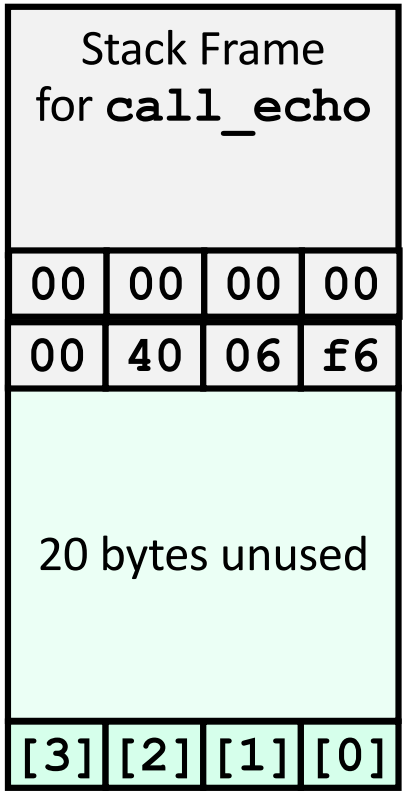
```
0000000004006cf <echo>:
 4006cf:  48 83 ec 18           sub    $0x18,%rsp
 4006d3:  48 89 e7              mov    %rsp,%rdi
 4006d6:  e8 a5 ff ff ff        callq  400680 <gets>
 4006db:  48 89 e7              mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff        callq  400520 <puts@plt>
 4006e3:  48 83 c4 18           add    $0x18,%rsp
 4006e7:  c3                    retq
```

call_echo:

```
 4006e8:  48 83 ec 08           sub    $0x8,%rsp
 4006ec:  b8 00 00 00 00        mov    $0x0,%eax
 4006f1:  e8 d9 ff ff ff        callq  4006cf <echo>
 4006f6:  48 83 c4 08           add    $0x8,%rsp
 4006fa:  c3                    retq
```

内存越界引用和缓冲区溢出

Before call to gets




```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006f1:   callq   4006cf <echo>
   4006f6:   add     $0x8,%rsp
   . . .
```

内存越界引用和缓冲区溢出

After call to gets

Stack Frame for call_echo			
00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ⟵ %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006f1:   callq   4006cf <echo>
   4006f6:   add     $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

缓冲区溢出，但没有破坏返回地址

内存越界引用和缓冲区溢出

After call to gets

Stack Frame for call_echo			
00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ⟵ %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006f1:   callq   4006cf <echo>
   4006f6:   add     $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

缓冲区溢出，破坏返回地址

Buffer Overflow Stack Example #3 Explained

After call to gets

Stack Frame for `call_echo`			
00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

`buf` ⟵ `%rsp`

register_tm_clones:

```
   . . .
400600:   mov     %rsp,%rbp
400603:   mov     %rax,%rdx
400606:   shr     $0x3f,%rdx
40060a:   add     %rdx,%rax
40060d:   sar     %rax
400610:   jne     400614
400612:   pop     %rbp
400613:   retq
```

返回到另外的代码段，利用进行缓冲区溢出攻击

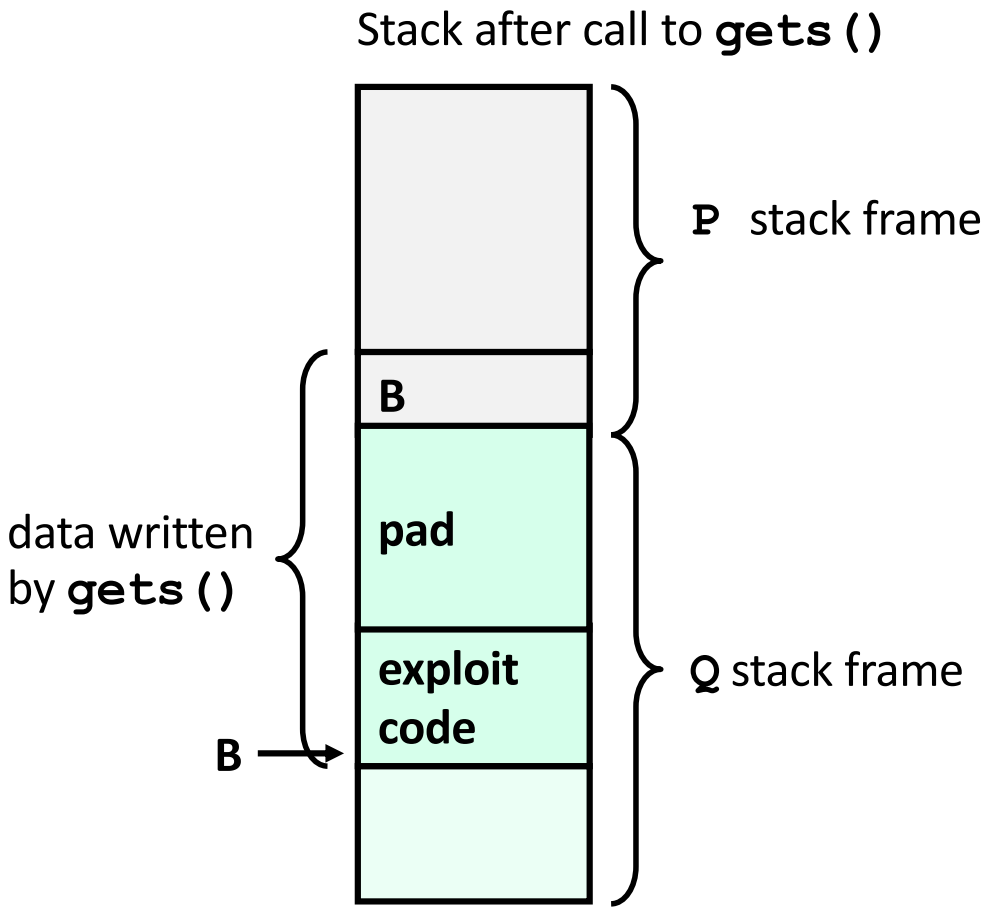
内存越界引用和缓冲区溢出

代码注入攻击Code Injection Attacks

```
void P(){
   Q();
   ...
}
```

return address A

```
int Q() {
   char buf[64];
   gets(buf);
   ...
   return ...;
}
```



- 输入给程序一个字符串，这个字符串包含一些可执行代码的字节编码，称为攻击代码(exploit code)，
- 攻击代码的地址覆盖原返回地址。
- 执行ret 指令跳转到攻击代码

对抗缓冲区溢出攻击

1. 程序员角度防范—在代码中避免

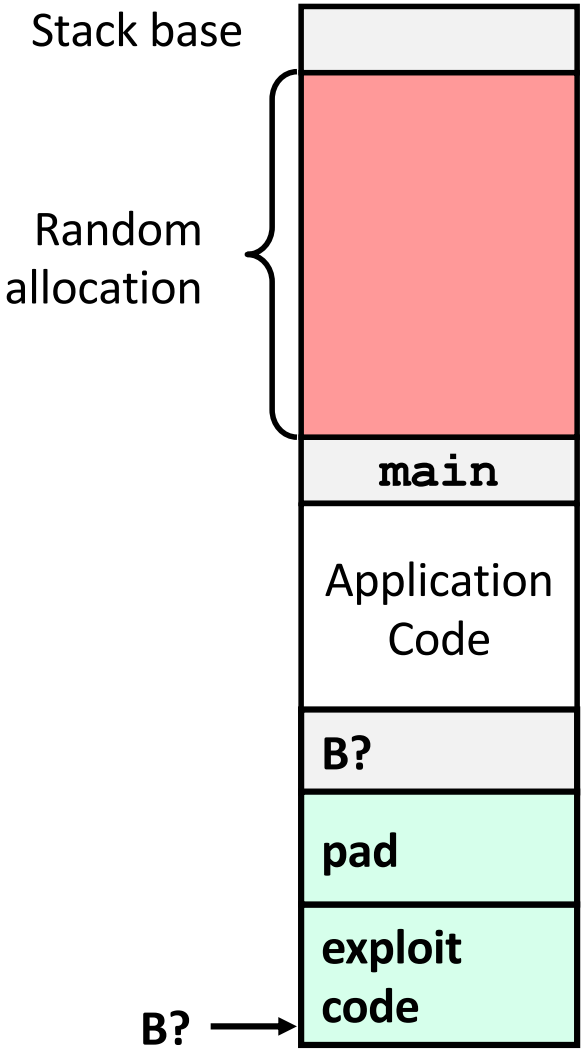
```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- 使用带限制长度的库函数
  - **fgets** instead of **gets**
  - **strncpy** instead of **strcpy**
  - Don't use **scanf** with **%s** conversion specification
    - Use **fgets** to read the string
    - Or use **%ns** where **n** is a suitable integer

对抗缓冲区溢出攻击

2、系统级别的防范

- 栈随机化
  - 栈的位置在程序每次运行时都有变化

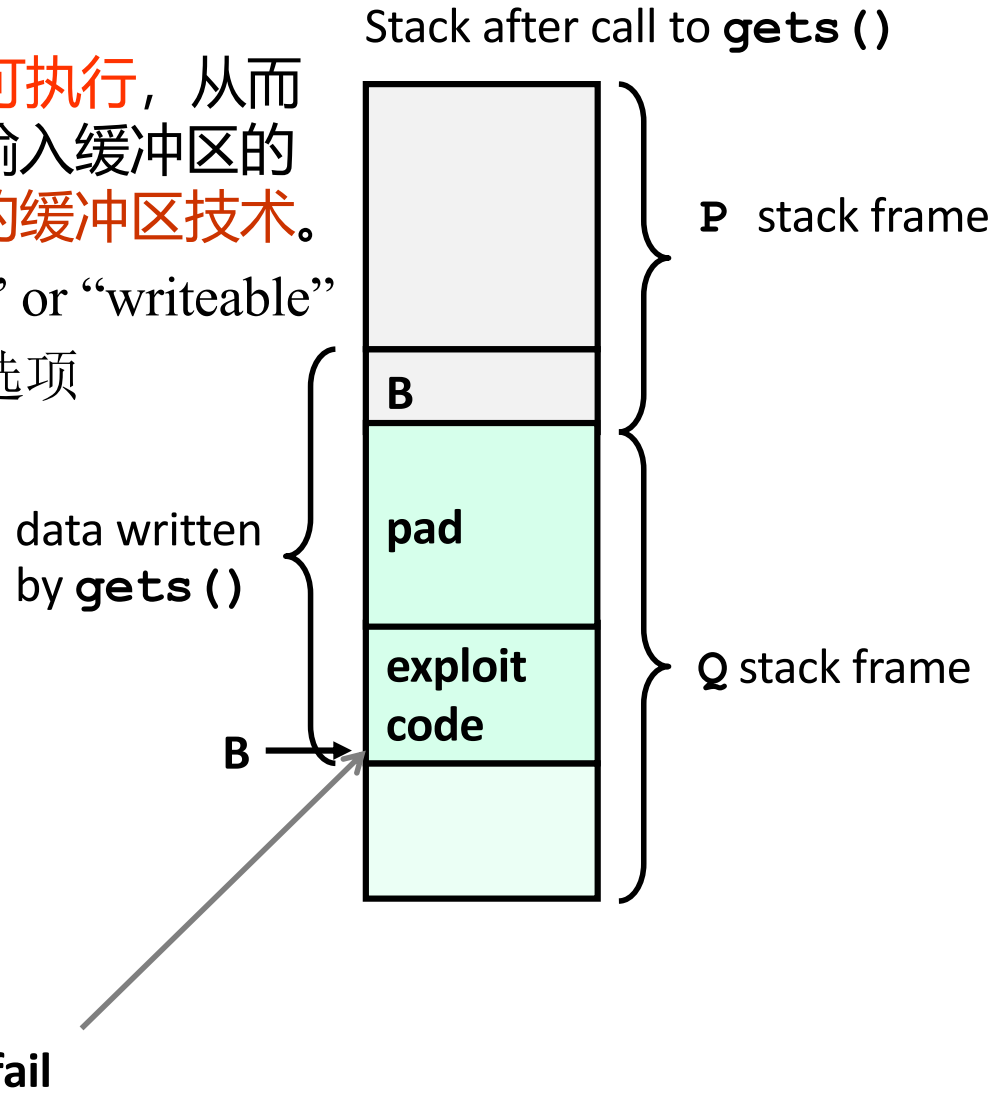


对抗缓冲区溢出攻击

- 限制可执行代码区域

通过将程序栈区和堆区设置为不可执行，从而使得攻击者不可能执行被植入在输入缓冲区的代码，这种技术也被称为非执行的缓冲区技术。

- 传统 x86, 内存分为 “read-only” or “writeable”
- x86-64 增加 可执行“execute” 选项
- 栈标记为非执行non-executable



缓冲区溢出攻击的防范

- 栈破坏检测
  - 若在程序跳转到攻击代码前能检测出程序栈已被破坏，就可避免受到严重攻击
  - 新GCC版本在代码中加入了一种栈保护者（stack canary）机制

    -fstack-protector，用于检测缓冲区是否越界
  - 主要思想：在函数准备阶段，在其栈帧中缓冲区底部与保存寄存器之间加入一个随机生成的特定值；在函数恢复阶段，在恢复寄存器并返回到调用函数前，先检查该值是否被改变。若改变则程序异常中止。因为插入在栈帧中的特定值是随机生成的，所以攻击者很难猜测出它是什么

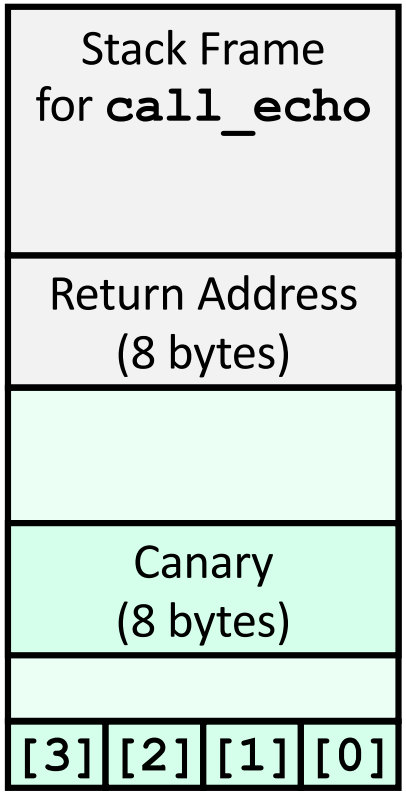
Protected Buffer Disassembly

echo:

```
40072f:   sub     $0x18,%rsp
400733:   mov     %fs:0x28,%rax          # 段寄存器：偏移
40073c:   mov     %rax,0x8(%rsp)
400741:   xor     %eax,%eax
400743:   mov     %rsp,%rdi
400746:   callq   4006e0 <gets>
40074b:   mov     %rsp,%rdi
40074e:   callq   400570 <puts@plt>
400753:   mov     0x8(%rsp),%rax
400758:   xor     %fs:0x28,%rax
400761:   je      400768 <echo+0x39>
400763:   callq   400580 <__stack_chk_fail@plt>
400768:   add     $0x18,%rsp
40076c:   retq
```

Setting Up Canary

Before call to gets



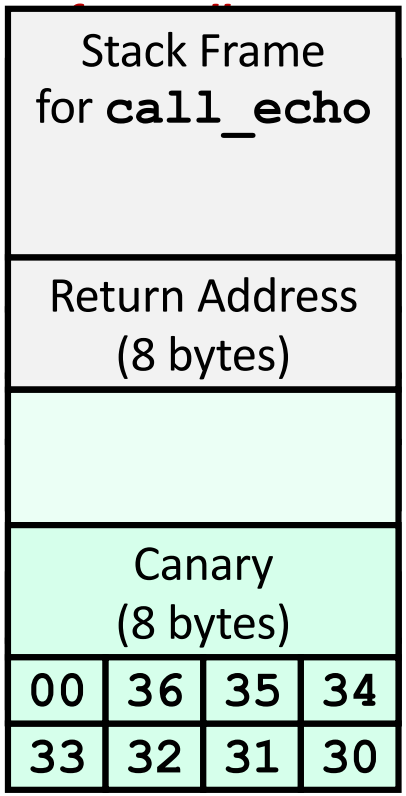
8字节对齐

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %rax   # Get canary
    movq    %rax, 8(%rsp)  # Place on stack
    xorl    %eax, %eax     # Erase canary
    . . .
```

Checking Canary

After call to gets




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Input: *0123456*

```
echo:
    . . .
    movq    8(%rsp), %rax     # Retrieve from stack
    xorq    %fs:40, %rax      # Compare to canary
    je      .L6               # If same, OK
    call    __stack_chk_fail  # FAIL
.L6:        . . .
```

程序的机器级表示

- 分以下五个部分介绍
  - 第一讲： x86-64指令系统
    - 高级语言程序转换为机器代码的过程
    - 机器指令和汇编指令
    - x86-64指令系统
  - 第二讲： C语言程序的机器级表示
    - 选择语句的机器级表示
    - 循环结构的机器级表示
    - 过程调用的机器级表示
  - 第三讲：复杂数据类型的分配和访问
    - 数组的分配和访问
    - 结构体数据的分配和访问
    - 联合体数据的分配和访问
    - 数据的对齐
  - 第四讲：越界引用和缓冲区溢出
  - 第五讲：浮点指令和代码

浮点指令与代码

- 发展历史
  - x87 指令集
    - *x86配套的浮点协处理器x87FPU架构
    - *80位浮点寄存器栈
  - SSE（流水式SIMD扩展）指令集
    - *由MMX发展而来的SSE指令集架构，采用的是单指令多数据
    - *128位的XMM寄存器
  - AVX （高级向量扩展）指令集
    - *最新版本
    - *256位的YMM寄存器

YMM 寄存器

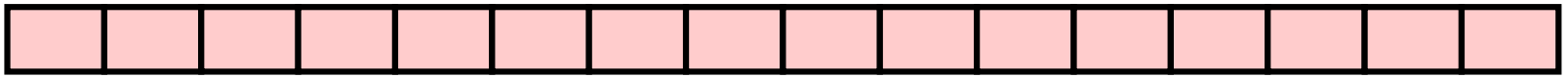
%ymm0	%xmm0
%ymm1	%xmm1
%ymm2	%xmm2
%ymm3	%xmm3
%ymm4	%xmm4
%ymm5	%xmm5
%ymm6	%xmm6
%ymm7	%xmm7

%ymm8	%xmm8
%ymm9	%xmm9
%ymm10	%xmm10
%ymm11	%xmm11
%ymm12	%xmm12
%ymm13	%xmm13
%ymm14	%xmm14
%ymm15	%xmm15

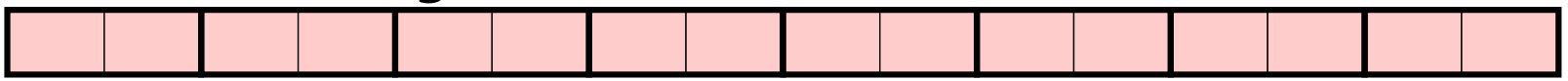
XMM 寄存器

- 16 total, each 16 bytes

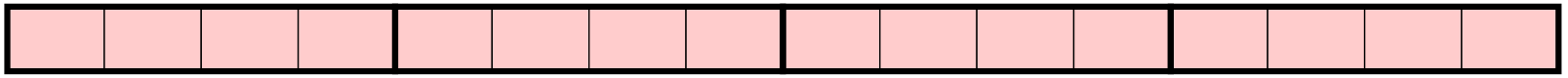
- 16 single-byte integers (char)



- 8 16-bit integers (short)



- 4 32-bit integers (int)



- 4 single-precision floats (float)

- 2 double-precision floats (double)

浮点传送指令

指令	源	目的	描述
vmovss	*M32*	*X*	传送单精度数
vmovss	*X*	*M32*	传送单精度数
vmovsd	*M64*	*X*	传送双精度数
vmovsd	*X*	*M64*	传送双精度数
vmovaps	*X*	*X*	传送对齐的封装好的单精度数
vmovapd	*X*	*X*	传送对齐的封装好的双精度数

浮点指令与代码

例:

```
float float_mov(float v1, float *src, float *dst) {
  float v2 = *src;
  *dst = v1;
  return v2;
}
```

与它相关联的x86-64 汇编代码为:

```
float float_mov(float v1 , float *src , float *dst)
v1 in %xmm0, src in %rdi, dst in %rsi
ret in %xmm0
float_mov:
  vmovaps %xmm0, %xmm1     # v1
  vmovss (%rdi), %xmm0     # return *src
  vmovss %xmm1, (%rsi)     # *dst=v1
  ret
```

过程中的浮点代码

在x86-64 中， XMM 寄存器用来向函数传递浮点参数，以及从函数返回浮点值。具体规则:

（1）XMM 寄存器%xmm0~%xmm7最多可以传递8个浮点参数。按照参数列出的顺序使用这些寄存器。通过栈传递额外的浮点参数。

（2）函数使用寄存器%xmm0来返回浮点值。

（3）当函数包含指针、整数和浮点数混合的参数时，指针和整数通过通用寄存器传递，而浮点值通过XMM寄存器传递

例：double fl(int x , double y , long z);

x 存放在%edi中， y存放在%xmm0中，而z存放在%rsi中

double fl(float x , double *y , long *z);

x放在%xmm0中， y放在%rdi中，而z放在%rsi中

浮点指令与代码

浮点运算指令

单精度	双精度	效果	描述
vaddss	vaddsd	D←S2+S1	浮点数加
vsubss	vsubsd	D←S2-S1	浮点数减
vmulss	vmulsd	D←S2*S1	浮点数乘
vdivss	vdivsd	D←S2/S1	浮点数除
sqrtss	sqrtsd	$D \leftarrow \sqrt{S1}$	浮点数平方根

*每条指令有一个(S1)或两个(S1,S2) 源操作数，和一个目的操作数D
*第一个源操作数S1可以是一个XMM 寄存器或一个内存位置
*第二个源操作数和目的操作数都必须是XMM寄存器

浮点指令与代码


```
float fadd(float x, float y) {
    return x + y;
}
```

```
# x in %xmm0, y in %xmm1
vaddss   %xmm1, %xmm0
ret
```

```
double dadd(double x, double y) {
    return x + y;
}
```

```
# x in %xmm0, y in %xmm1
vaddsd   %xmm1, %xmm0
ret
```

定义和使用浮点常数

```
double ce12fabr (double temp)
{
   return 1.8*temp+32.0
}
```

x86-64 汇编代码部分:

```
cel2fahr:
   vmulsd .LC2(%rip), %xmm0, %xmm0     Multiply by 1.8
   vaddsd .LC3(%rip), %xmm0, %xmm0     Add 32.0
   ret    编译器生成的常量标签
.LC2:     RIP 相对寻址，表示从当前指令地址到标签的偏移量
   .long 3435973837          Low-order 4 bytes of 1.8
   .long 1073532108          High-order 4 bytes of 1.8
.LC3:
   .long 0                   Low-order 4 bytes of 32.0
   .long 1077936128          High-order 4 bytes of 32.0
```

在数据段中直接存储原始4字节数据

其它浮点指令

- 浮点转换指令
  - 浮点数和整数数据类型之间转换
  - 不同浮点格式之间转换
- 浮点比较指令
  - 指令 **`ucomiss`** and **`ucomisd`**
  - Set condition codes CF, ZF, and PF
- 浮点位操作指令
  - 指令vxorps vxorpd vandps vandpd

Parity Flag，奇偶标志位：检查数据的最低有效字节（LSB）中 1 的个数是否为偶数，即至少一个操作数是 NaN