

Machine-Level Programming: x86-64 Procedures, Data

17: Data

April 08, 2013

Today

- **Procedures (x86-64)**
- **Arrays**
 - One-dimensional
 - Multi-dimensional (nested)
 - Multi-level
- **Structures**
 - Allocation
 - Access

x86-64 Integer Registers

<code>%rax</code>	<code>%eax</code>
<code>%rbx</code>	<code>%ebx</code>
<code>%rcx</code>	<code>%ecx</code>
<code>%rdx</code>	<code>%edx</code>
<code>%rsi</code>	<code>%esi</code>
<code>%rdi</code>	<code>%edi</code>
<code>%rsp</code>	<code>%esp</code>
<code>%rbp</code>	<code>%ebp</code>

<code>%r8</code>	<code>%r8d</code>
<code>%r9</code>	<code>%r9d</code>
<code>%r10</code>	<code>%r10d</code>
<code>%r11</code>	<code>%r11d</code>
<code>%r12</code>	<code>%r12d</code>
<code>%r13</code>	<code>%r13d</code>
<code>%r14</code>	<code>%r14d</code>
<code>%r15</code>	<code>%r15d</code>

- Twice the number of registers
- Accessible as 8, 16, 32, 64 bits

x86-64 Integer Registers: Usage Conventions

<code>%rax</code>	Return value
<code>%rbx</code>	Callee saved
<code>%rcx</code>	Argument #4
<code>%rdx</code>	Argument #3
<code>%rsi</code>	Argument #2
<code>%rdi</code>	Argument #1
<code>%rsp</code>	Stack pointer
<code>%rbp</code>	Callee saved

<code>%r8</code>	Argument #5
<code>%r9</code>	Argument #6
<code>%r10</code>	Caller saved
<code>%r11</code>	Caller Saved
<code>%r12</code>	Callee saved
<code>%r13</code>	Callee saved
<code>%r14</code>	Callee saved
<code>%r15</code>	Callee saved

x86-64 Registers

- **Arguments passed to functions via registers**
 - If more than 6 integral parameters, then pass rest on stack
 - These registers can be used as caller-saved as well
- **All references to stack frame via stack pointer**
 - Eliminates need to update `%ebp/%rbp`
- **Other Registers**
 - 6 callee saved
 - 2 caller saved
 - 1 return value (also usable as caller saved)
 - 1 special (stack pointer)

x86-64 Long Swap

```
void swap_l(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    ret
```

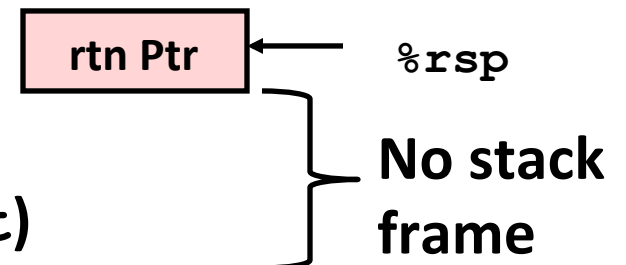
■ Operands passed in registers

- First (**xp**) in `%rdi`, second (**yp**) in `%rsi`
- 64-bit pointers

■ No stack operations required (except `ret`)

■ Avoiding stack

- Can hold all local information in registers



x86-64 NonLeaf without Stack Frame

```
/* Swap a[i] & a[i+1] */  
void swap_ele(long a[], int i)  
{  
    swap(&a[i], &a[i+1]);  
}
```

- No values held while swap being invoked
- No callee save registers needed
- `rep` instruction inserted as no-op
 - Based on recommendation from AMD

```
swap_ele:  
    movslq %esi,%rsi          # Sign extend i  
    leaq   8(%rdi,%rsi,8), %rax # &a[i+1]  
    leaq   (%rdi,%rsi,8), %rdi  # &a[i] (1st arg)  
    movq   %rax, %rsi          # (2nd arg)  
    call   swap  
    rep  
    ret                       # No-op
```

Interesting Features of Stack Frame

- **Allocate entire frame at once**
 - All stack accesses can be relative to `%rsp`
 - Do by decrementing stack pointer
 - Can delay allocation, since safe to temporarily use red zone
- **Simple deallocation**
 - Increment stack pointer
 - No base/frame pointer needed

x86-64 Procedure Summary

■ Heavy use of registers

- Parameter passing
- More temporaries since more registers

■ Minimal use of stack

- Sometimes none
- Allocate/deallocate entire block

■ Many tricky optimizations

- What kind of stack frame to use
- Various allocation techniques

Today

- Procedures (x86-64)
- **Arrays**
 - One-dimensional
 - Multi-dimensional (nested)
 - Multi-level
- Structures

Basic Data Types

■ Integral

- Stored & operated on in general (integer) registers
- Signed vs. unsigned depends on instructions used

Intel	ASM	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	d	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

■ Floating Point

- Stored & operated on in floating point registers

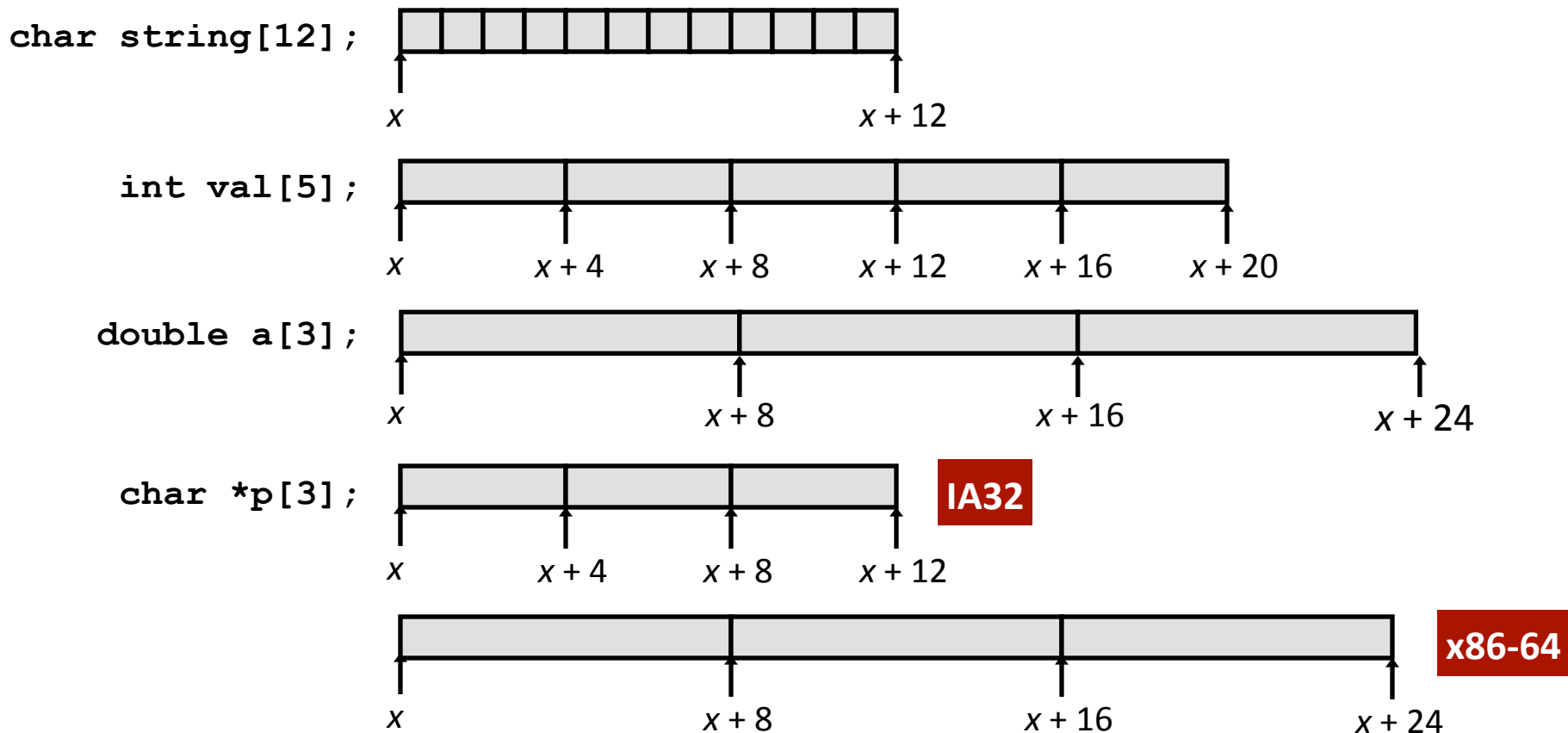
Intel	ASM	Bytes	C
Single	s	4	float
Double	d	8	double
Extended	t	10/12/16	long double

Array Allocation

■ Basic Principle

T $A[L]$;

- Array of data type T and length L
- Contiguously allocated region of $L * \text{sizeof}(T)$ bytes

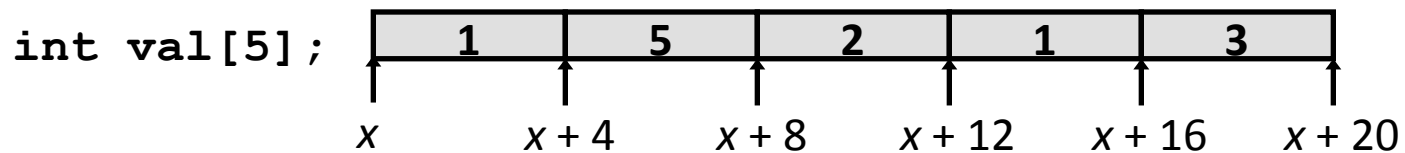


Array Access

■ Basic Principle

T **A**[L];

- Array of data type T and length L
- Identifier **A** can be used as a pointer to array element 0: Type T^*



■ Reference Type Value

`val[4]`

`val`

`val+1`

`&val[2]`

`val[5]`

`*(val+1)`

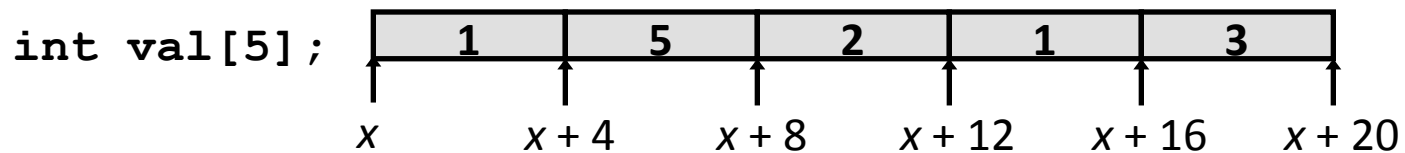
`val + i`

Array Access

■ Basic Principle

T $\mathbf{A}[L]$;

- Array of data type T and length L
- Identifier \mathbf{A} can be used as a pointer to array element 0: Type T^*

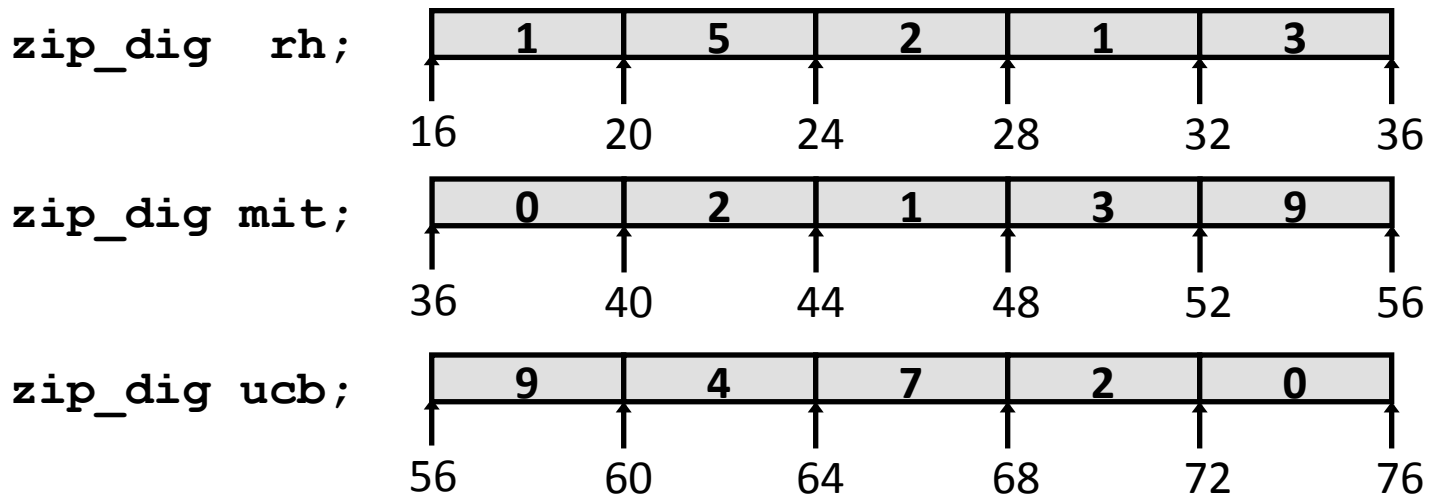


■ Reference	Type	Value
<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	x
<code>val+1</code>	<code>int *</code>	$x+4$
<code>&val[2]</code>	<code>int *</code>	$x+8$
<code>val[5]</code>	<code>int</code>	??
<code>*(val+1)</code>	<code>int</code>	5
<code>val + i</code>	<code>int *</code>	$x+4i$

Array Example

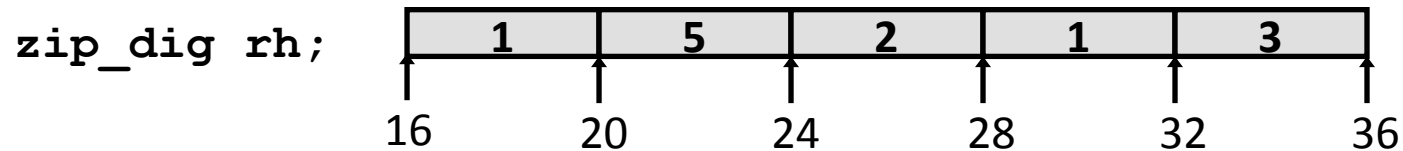
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig rh = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- Declaration “`zip_dig rh`” equivalent to “`int rh[5]`”
- Example arrays were allocated in successive 20 byte blocks
 - Not guaranteed to happen in general

Array Accessing Example



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

IA32

```
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax # z[dig]
```

- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at $4 * \%eax + \%edx$
- Use memory reference $(\%edx, \%eax, 4)$

Array Loop Example (IA32)

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

```
# edx = z
movl  $0, %eax          # %eax = i
.L4:                    # loop:
addl  $1, (%edx,%eax,4) # z[i]++
addl  $1, %eax          # i++
cmpl  $5, %eax         # i:5
jne   .L4              # if !=, goto loop
```

Pointer Loop Example (IA32)

```
void zincr_p(zip_dig z) {
    int *zend = z+ZLEN;
    do {
        (*z)++;
        z++;
    } while (z != zend);
}
```

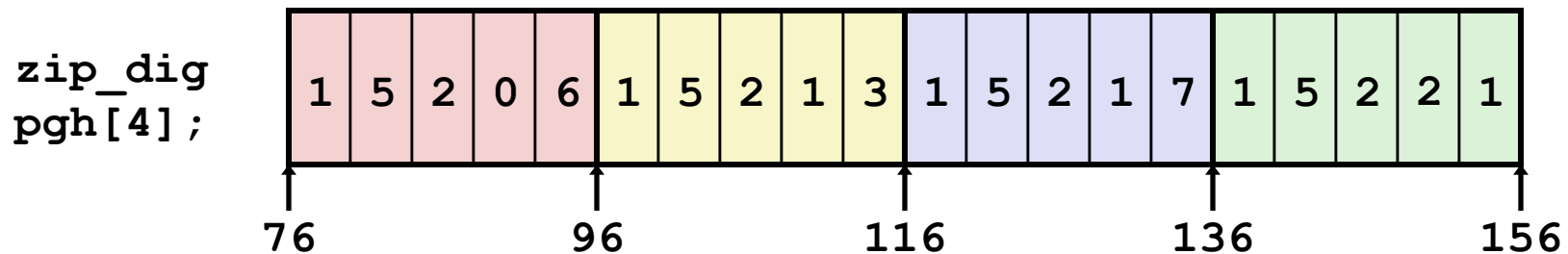


```
void zincr_v(zip_dig z) {
    void *vz = z;
    int i = 0;
    do {
        (*((int *) (vz+i)))++;
        i += ISIZE;
    } while (i != ISIZE*ZLEN);
}
```

```
# edx = z = vz
movl  $0, %eax
.L8:
addl  $1, (%edx,%eax)
addl  $4, %eax
cmpl  $20, %eax
jne   .L8
# i = 0
# loop:
# Increment vz+i
# i += 4
# Compare i:20
# if !=, goto loop
```

Nested Array Example

```
#define PCOUNT 4
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]” equivalent to “int pgh[4][5]”**
 - Variable `pgh`: array of 4 elements, allocated contiguously
 - Each element is an array of 5 `int`'s, allocated contiguously
- **“Row-Major” ordering of all elements guaranteed**

Multidimensional (Nested) Arrays

■ Declaration

`T A[R][C];`

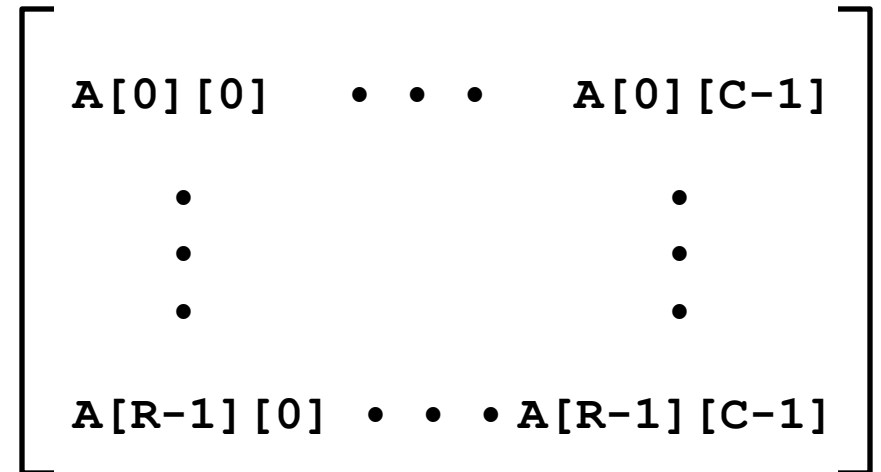
- 2D array of data type T
- R rows, C columns
- Type T element requires K bytes

■ Array Size

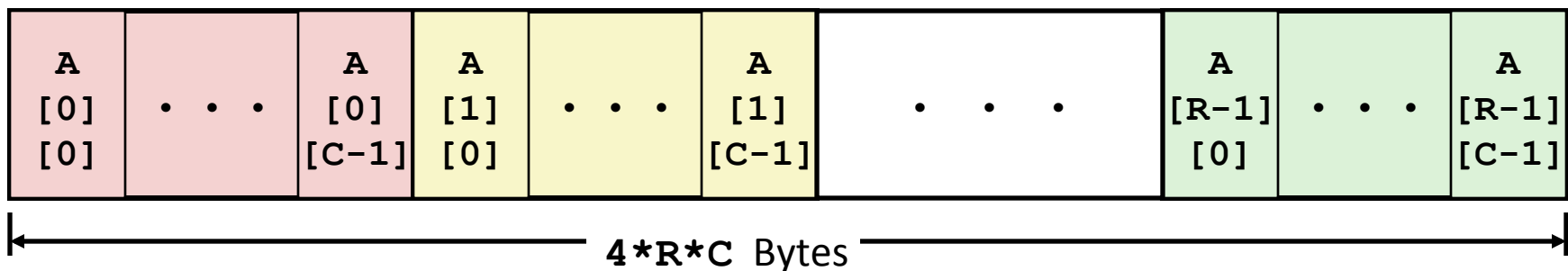
- $R * C * K$ bytes

■ Arrangement

- Row-Major Ordering



`int A[R][C];`

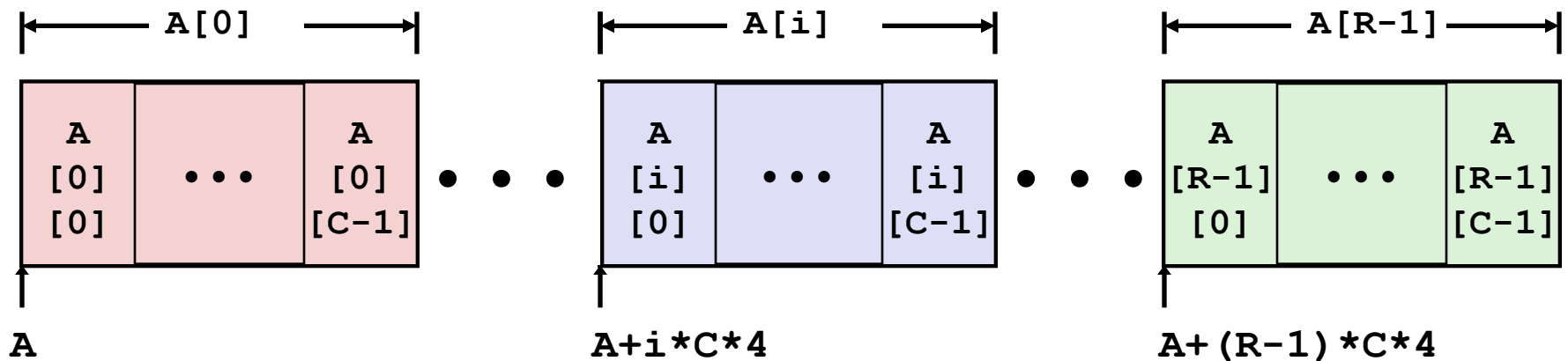


Nested Array Row Access

■ Row Vectors

- $\mathbf{A}[i]$ is array of C elements
- Each element of type T requires K bytes
- Starting address $\mathbf{A} + i * (C * K)$

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

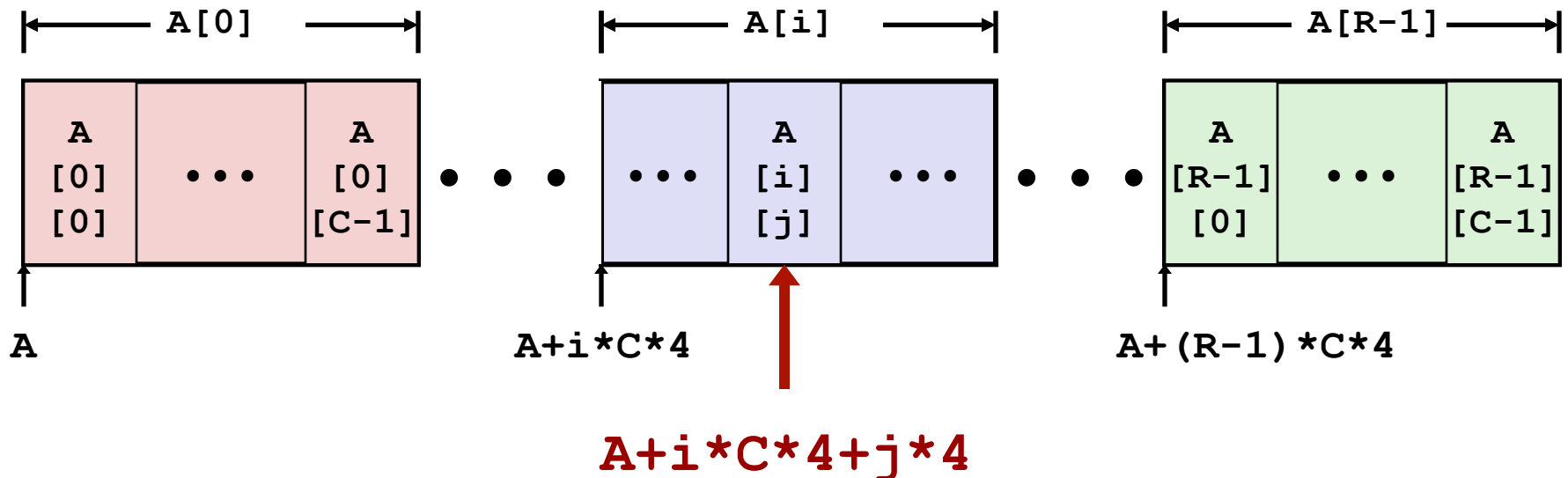


Nested Array Row Access

■ Array Elements

- $A[i][j]$ is element of type T , which requires K bytes
- Address $A + i * (C * K) + j * K = A + (i * C + j) * K$

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



Nested Array Element Access Code

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

```
movl    8(%ebp), %eax        # index
leal    (%eax,%eax,4), %eax  # 5*index
addl    12(%ebp), %eax       # 5*index+dig
movl    pgh(,%eax,4), %eax   # offset 4*(5*index+dig)
```

■ Array Elements

- `pgh[index][dig]` is `int`
- Address: `pgh + 20*index + 4*dig`
 - = `pgh + 4*(5*index + dig)`

■ IA32 Code

- Computes address `pgh + 4*((index+4*index)+dig)`

16 X 16 Matrix Access

■ Array Elements

- Address $A + i * (C * K) + j * K$
- $C = 16, K = 4$

```
/* 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)
```


n X n Matrix Access

■ Array Elements

- Address $\mathbf{A} + i * (\mathbf{C} * \mathbf{K}) + j * \mathbf{K}$
- $\mathbf{C} = \mathbf{n}, \mathbf{K} = 4$

```
/* 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)
```

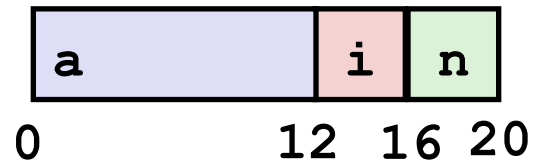
Today

- **Procedures (x86-64)**
- **Arrays**
 - One-dimensional
 - Multi-dimensional (nested)
 - Multi-level
- **Structures**
 - Allocation
 - Access

Structure Allocation

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

Memory Layout

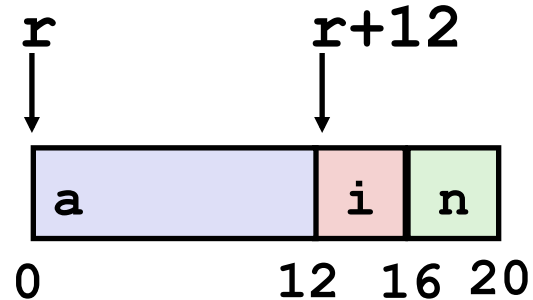


■ Concept

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

Structure Access

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



■ Accessing Structure Member

- Pointer indicates first byte of structure
- Access elements with offsets

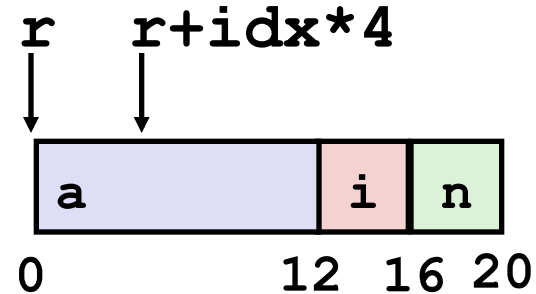
```
void  
set_i(struct rec *r,  
      int val)  
{  
    r->i = val;  
}
```

IA32 Assembly

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

Generating Pointer to Structure Member

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



■ Generating Pointer to Array Element

- Offset of each structure member determined at compile time

▪ Arguments

- Mem[%ebp+8]: **r**
- Mem[%ebp+12]: **idx**

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

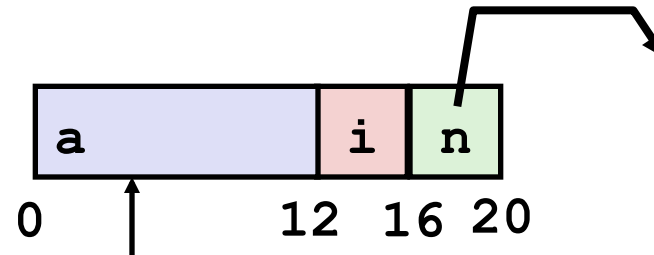
```
movl    12(%ebp), %eax    # Get idx  
sall    $2, %eax         # idx*4  
addl    8(%ebp), %eax    # r+idx*4
```

Following Linked List

■ C Code

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

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



Element i

Register	Value
%edx	r
%ecx	val

```
.L17:                # loop:
    movl    12(%edx), %eax    # r->i
    movl    %ecx, (%edx,%eax,4) # r->a[i] = val
    movl    16(%edx), %edx    # r = r->n
    testl   %edx, %edx       # Test r
    jne     .L17             # If != 0 goto loop
```

Summary

- **Procedures in x86-64**
 - Stack frame is relative to stack pointer
 - Parameters passed in registers
- **Arrays**
 - One-dimensional
 - Multi-dimensional (nested)
 - Multi-level
- **Structures**
 - Allocation
 - Access