# Machine-Level Programming IV: x86-64 Procedures, Data

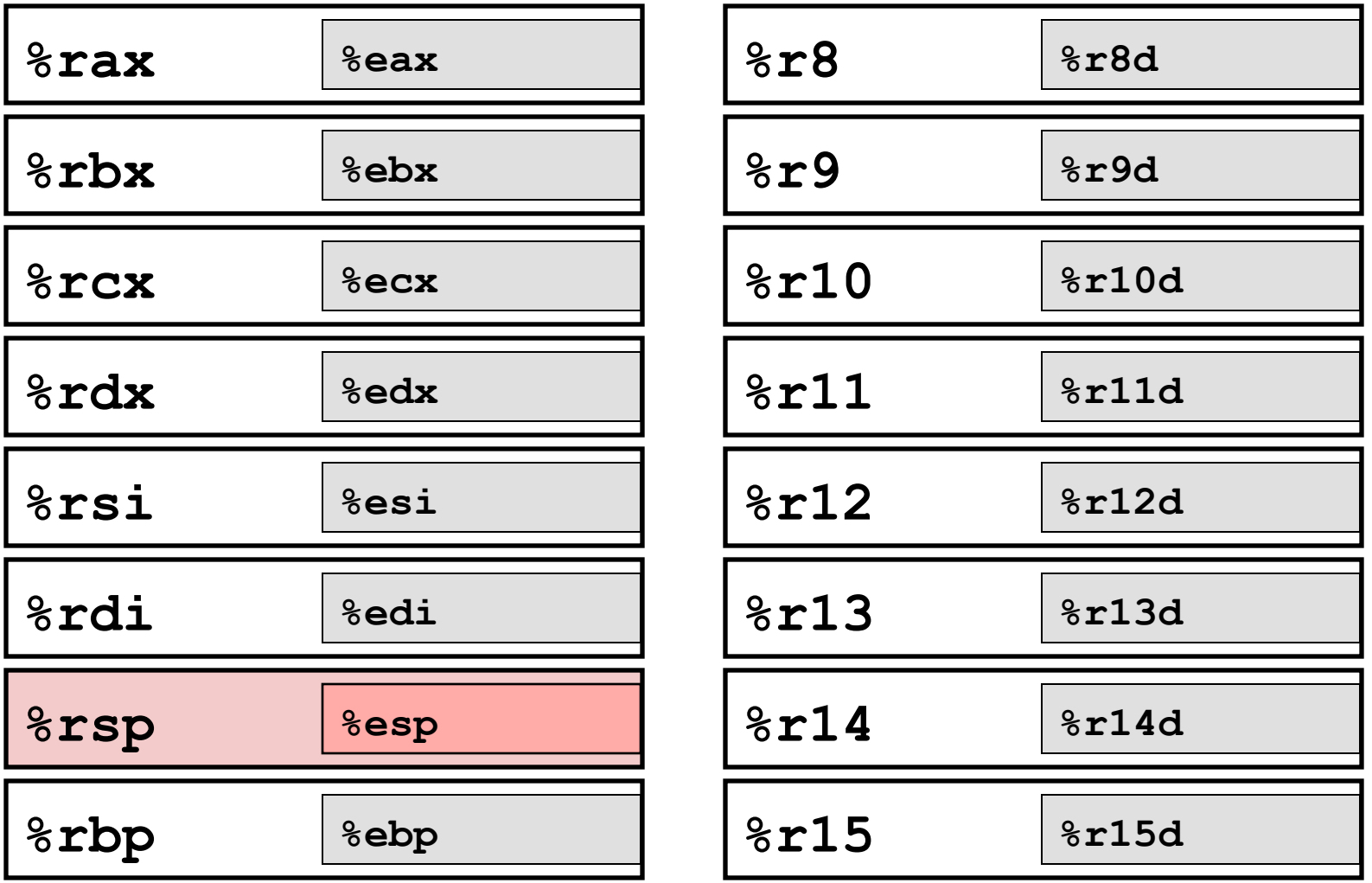
Lecture 5, March 17, 2011
Alexandre David

Credits to Randy Bryant & Dave O'Hallaron from Carnegie Mellon

# Today

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

# x86-64 Integer Registers




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

# x86-64 Integer Registers: Usage Conventions

| | |
|---|---|
| %rax | Return value |
| %rbx | Callee saved |
| %rcx | Argument #4 |
| %rdx | Argument #3 |
| %rsi | Argument #2 |
| %rdi | Argument #1 |
| %rsp | Stack pointer |
| %rbp | Callee saved |

| | |
|---|---|
| %r8 | Argument #5 |
| %r9 | Argument #6 |
| %r10 | Caller saved |
| %r11 | Caller Saved |
| %r12 | Callee saved |
| %r13 | Callee saved |
| %r14 | Callee saved |
| %r15 | Callee saved |

# x86-64 Registers

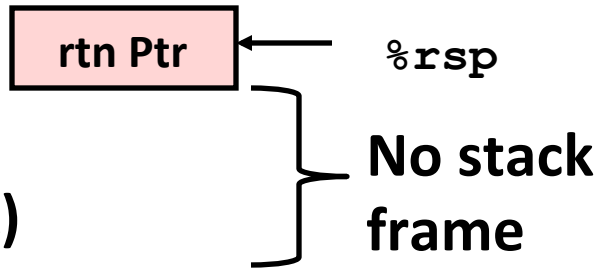
- **Arguments passed to functions via registers**
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well
- **All references to stack frame via stack pointer**
  - Eliminates need to update **`%ebp/%rbp`**
- **Other Registers**
  - 6 callee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



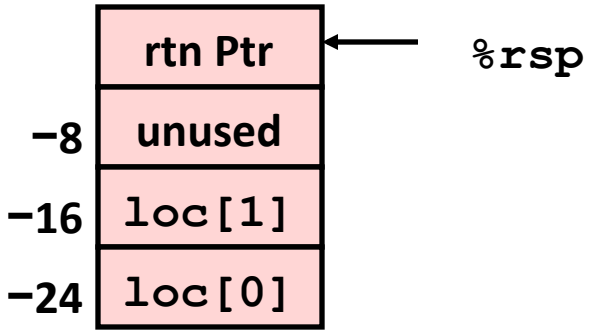
# x86-64 Locals in the Red Zone

PP3.51
PP3.52

```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
   movq   (%rdi), %rax
   movq   %rax, -24(%rsp)
   movq   (%rsi), %rax
   movq   %rax, -16(%rsp)
   movq   -16(%rsp), %rax
   movq   %rax, (%rdi)
   movq   -24(%rsp), %rax
   movq   %rax, (%rsi)
   ret
```

- **Avoiding Stack Pointer Change**
  - Can hold all information within small window beyond stack pointer



# 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                             # No-op
   ret
```

# x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
   (long a[], int i)
{
     swap(&a[i], &a[i+1]);
     sum += (a[i]*a[i+1]);
}
```

- Keeps values of `&a[i]` and `&a[i+1]` in callee save registers
- Must set up stack frame to save these registers

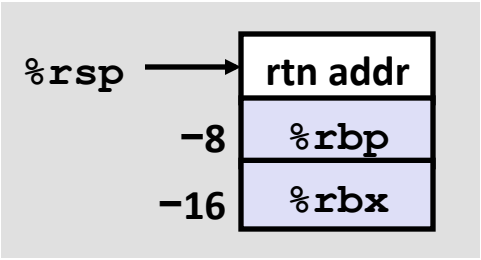
```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi,%rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

# Understanding x86-64 Stack Frame

```
swap_ele_su:
    movq    %rbx, -16(%rsp)         # Save %rbx
    movq    %rbp, -8(%rsp)          # Save %rbp
    subq    $16, %rsp               # Allocate stack frame
    movslq  %esi,%rax               # Extend i
    leaq    8(%rdi,%rax,8), %rbx    # &a[i+1] (callee save)
    leaq    (%rdi,%rax,8), %rbp     # &a[i]   (callee save)
    movq    %rbx, %rsi              # 2nd argument
    movq    %rbp, %rdi              # 1st argument
    call    swap
    movq    (%rbx), %rax            # Get a[i+1]
    imulq   (%rbp), %rax            # Multiply by a[i]
    addq    %rax, sum(%rip)         # Add to sum
    movq    (%rsp), %rbx            # Restore %rbx
    movq    8(%rsp), %rbp           # Restore %rbp
    addq    $16, %rsp               # Deallocate frame
    ret
```

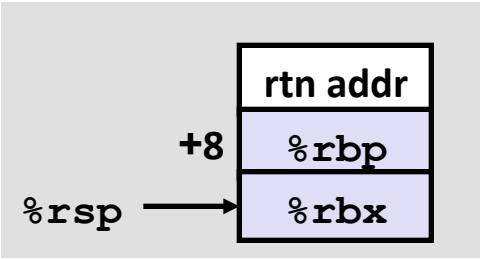
# Understanding x86-64 Stack Frame

```
movq    %rbx, -16(%rsp)        # Save %rbx
movq    %rbp, -8(%rsp)         # Save %rbp
```




```
subq    $16, %rsp              # Allocate stack frame
```

● ● ●




```
movq    (%rsp), %rbx           # Restore %rbx
movq    8(%rsp), %rbp          # Restore %rbp

addq    $16, %rsp              # Deallocate frame
```

# 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 | `l` | 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 | `l` | 8 | `double` |
| Extended | `t` | 10/12/16 | `long double` |

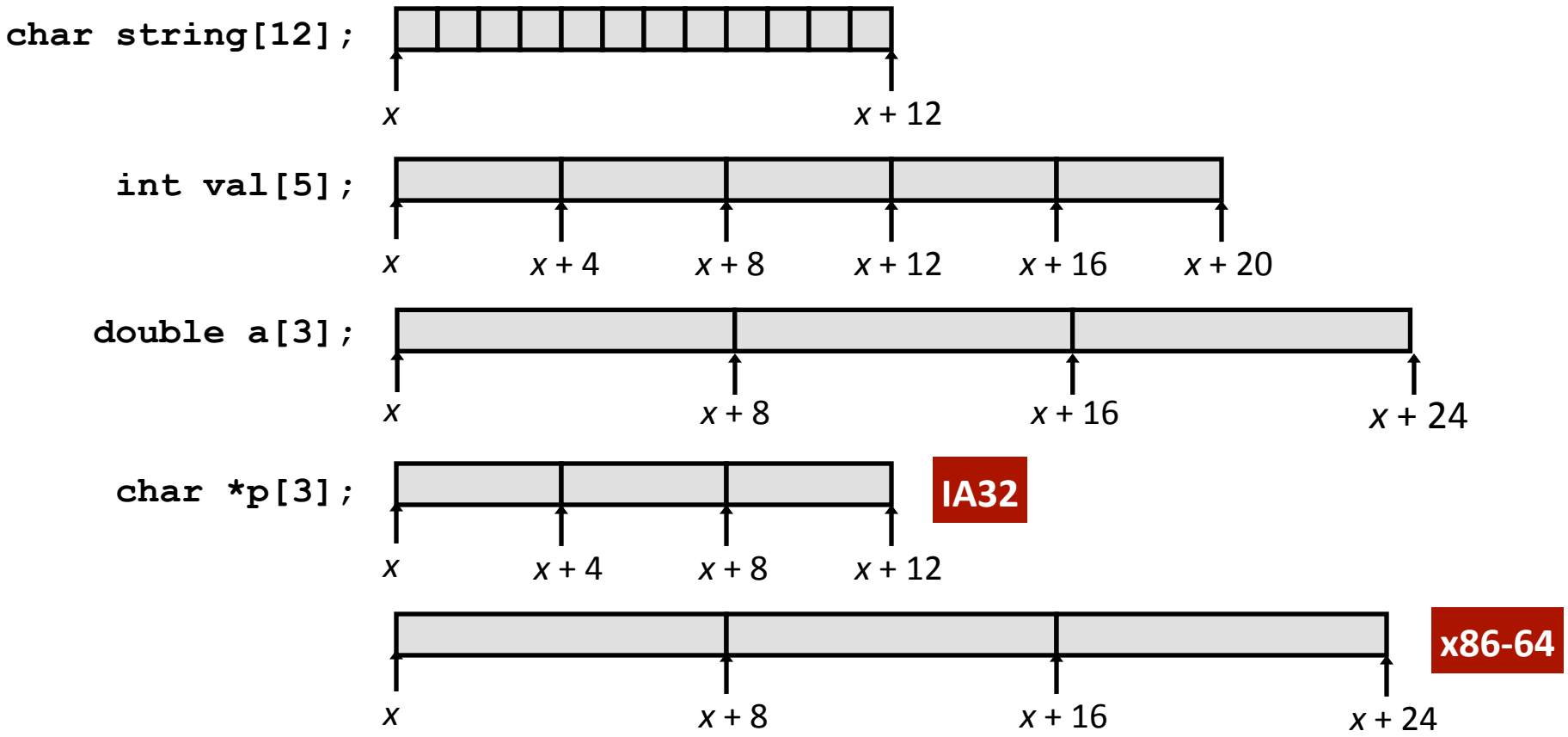
# Array Allocation

PP3.35

- **Basic Principle**

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

  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **`sizeof`**`(`*T*`)` bytes

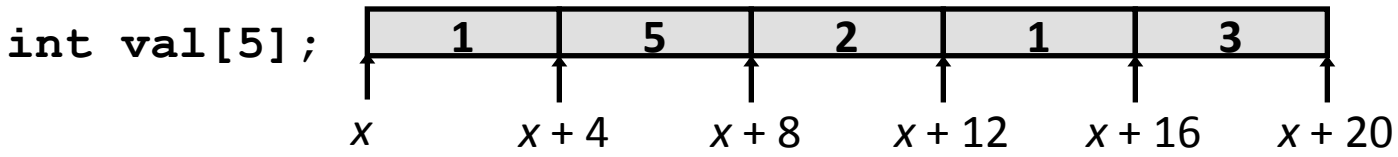


# Array Access

- **Basic Principle**

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

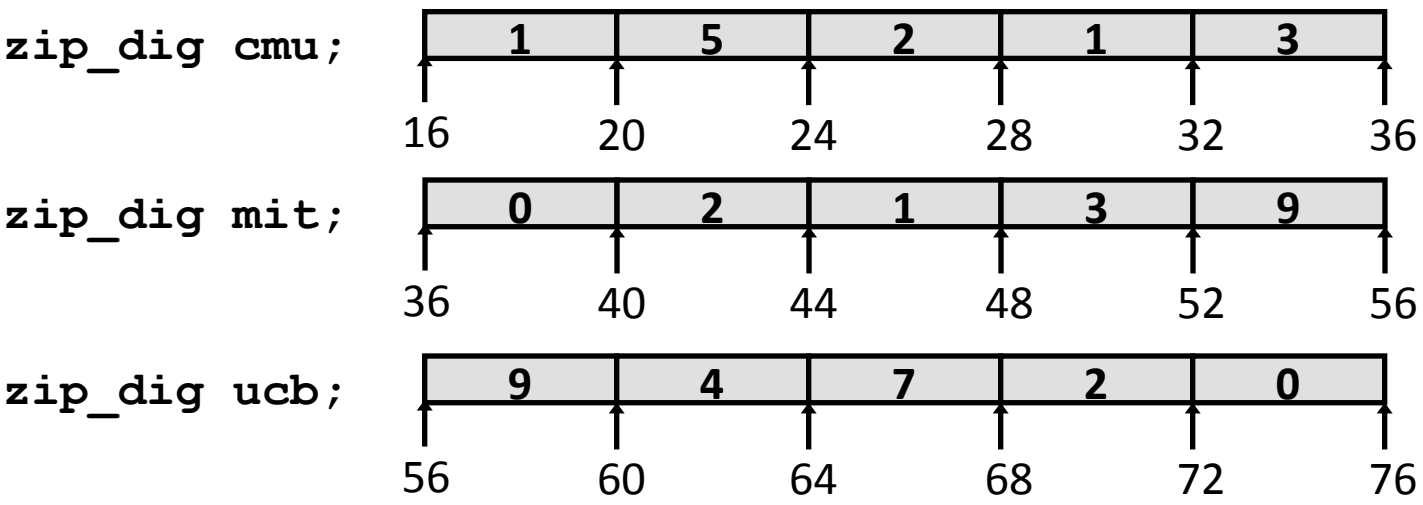
  - Array of data type *T* and length *L*
  - Identifier **`A`** can be used as a pointer to array element 0: Type *T**




| Reference | Type | Value |
|---|---|---|
| `val[4]` | `int` | 3 |
| `val` | `int *` | *x* |
| `val+1` | `int *` | *x* + 4 |
| `&val[2]` | `int *` | *x* + 8 |
| `val[5]` | `int` | ?? |
| `*(val+1)` | `int` | 5 |
| `val + i` | `int *` | *x* + 4 *i* |

# Array Example

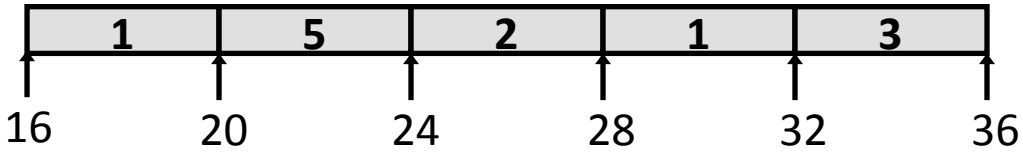
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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




- **Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"**
- **Example arrays were allocated in successive 20 byte blocks**
  - Not guaranteed to happen in general

# Array Accessing Example

```
zip_dig cmu;
```




```
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)

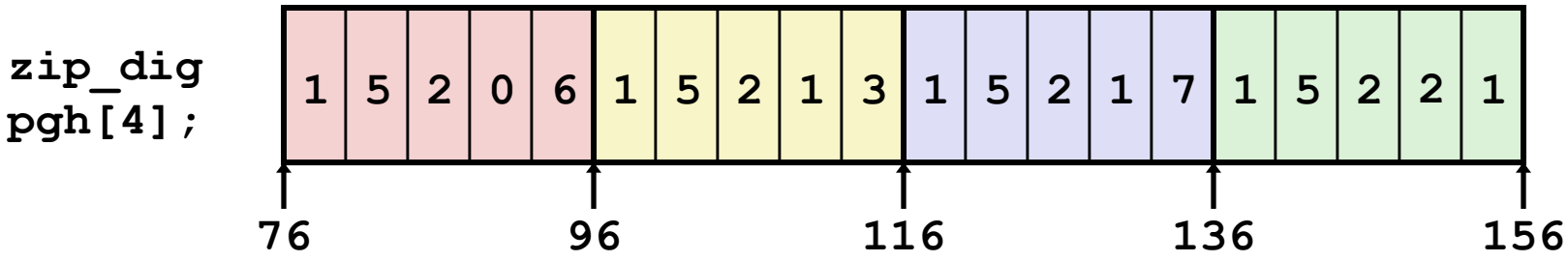
```
void zincr_p(zip_dig z) {
  int *zend = z+ZLEN;
  do {
    (*z)++;
    z++;
  } while (z != zend);
}
```

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

# Nested Array Example

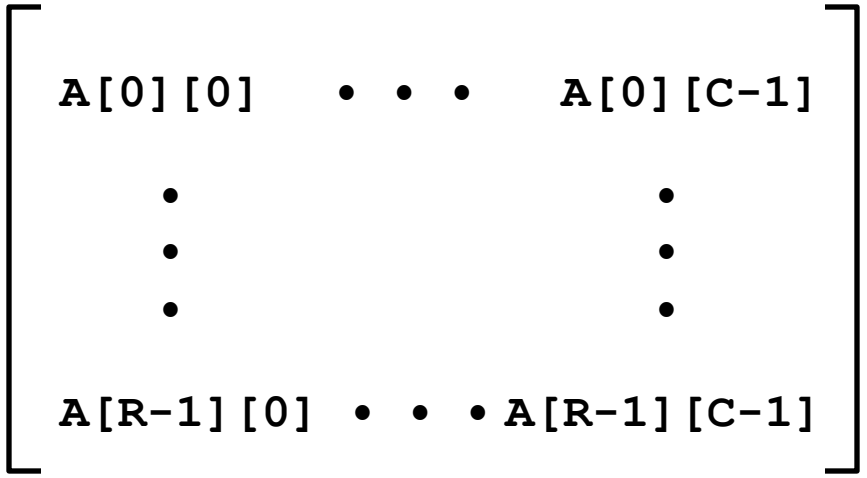
```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



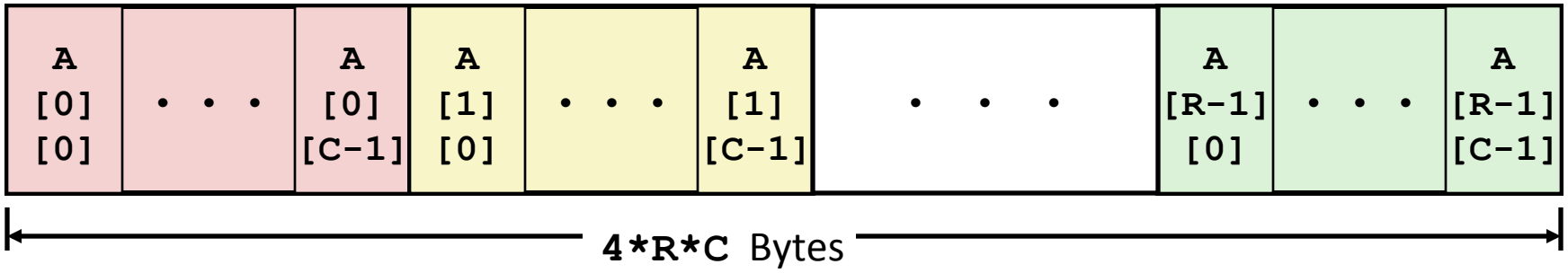

- **"`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"**
  - Variable **`pgh`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- **"Row-Major" ordering of all elements guaranteed**

# 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



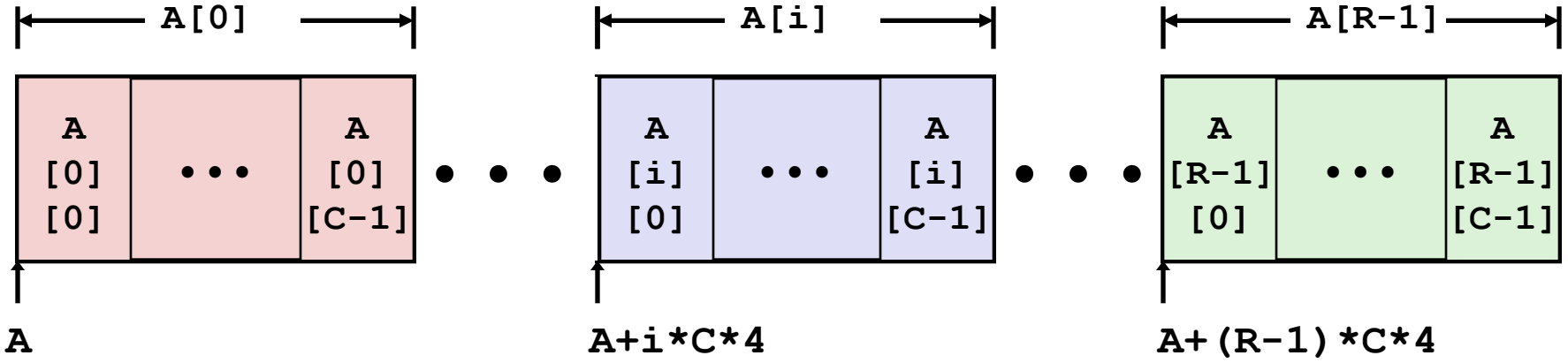

`int A[R][C];`



# Nested Array Row Access

- **Row Vectors**
  - **`A[i]`** is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Starting address **`A + `** *i* * (*C* * *K*)

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



# Nested Array Row Access Code

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

```
#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 }};
```

```
  # %eax = index
   leal (%eax,%eax,4),%eax # 5 * index
   leal pgh(,%eax,4),%eax  # pgh + (20 * index)
```

- **Row Vector**
  - **`pgh[index]`** is array of 5 **`int`**'s
  - Starting address **`pgh+20*index`**
- **IA32 Code**
  - Computes and returns address
  - Compute as **`pgh + 4*(index+4*index)`**

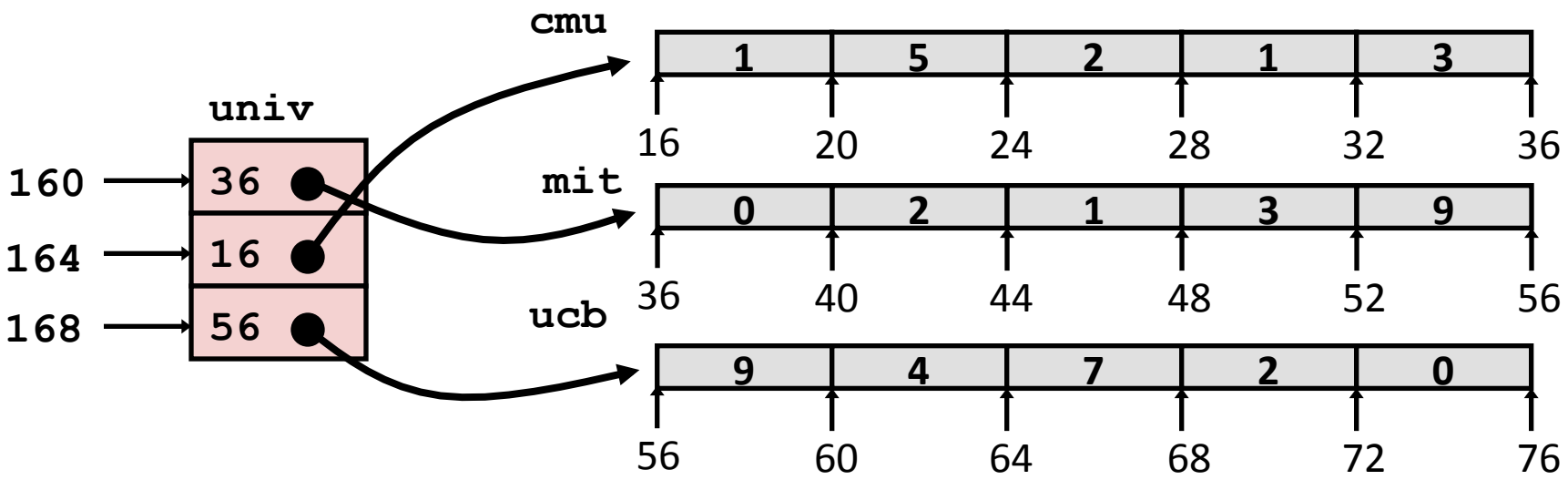
Java-like arrays

# Multi-Level Array Example

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

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of `int`'s



# Element Access in Multi-Level Array

```
int get_univ_digit
  (int index, int dig)
{
  return univ[index][dig];
}
```

```
   movl   8(%ebp), %eax              # index
   movl   univ(,%eax,4), %edx        # p = univ[index]
   movl   12(%ebp), %eax             # dig
   movl   (%edx,%eax,4), %eax        # p[dig]
```

- **Computation (IA32)**
  - Element access **`Mem[Mem[univ+4*index]+4*dig]`**
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

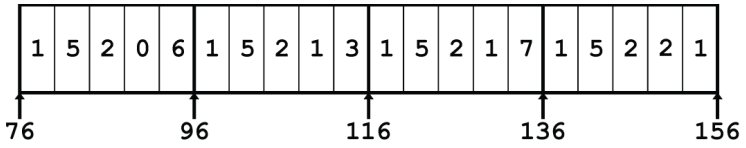
# Array Element Accesses - Comparison

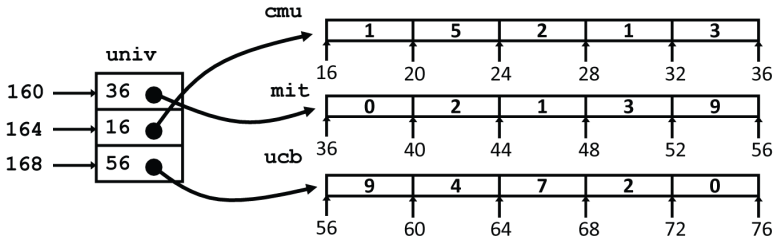
**Nested array**

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

**Multi-level array**

```
int get_univ_digit
   (int index, int dig)
{
  return univ[index][dig];
}
```




Accesses looks similar in C, but addresses very different:

**`Mem[pgh+20*index+4*dig]`**

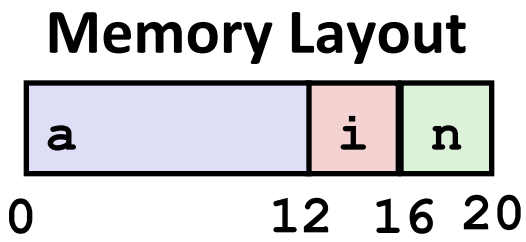
**`Mem[Mem[univ+4*index]+4*dig]`**

# 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;
};
```

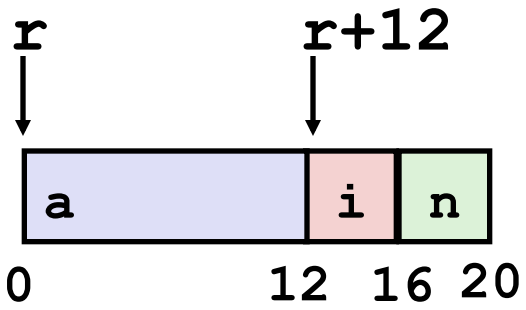



- **Concept**
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

PP3.39

# 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

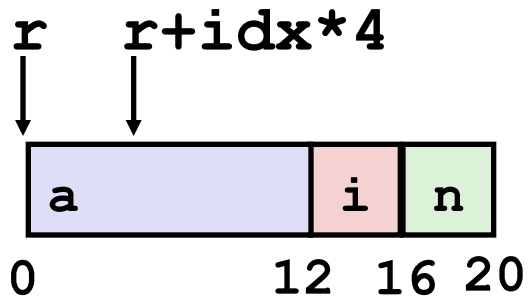
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

**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];
}
```

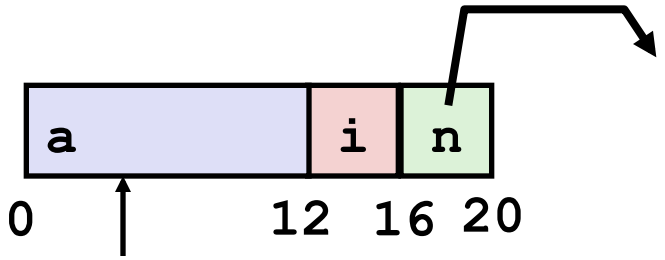
```
movl   12(%ebp), %eax   # Get idx
sall   $2, %eax         # idx*4
addl   8(%ebp), %eax    # r+idx*4
```

# Following Linked List

- C Code

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

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




| 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
  - You can count on your compiler!
- **Structures**
  - Allocation
  - Access