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

15-213 / 18-213 / 15-513: Introduction to Computer Systems
8th Lecture, February 5, 2015

Franz Franchetti, Seth Copen Goldstein, Greg Kesden

Today

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

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- 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

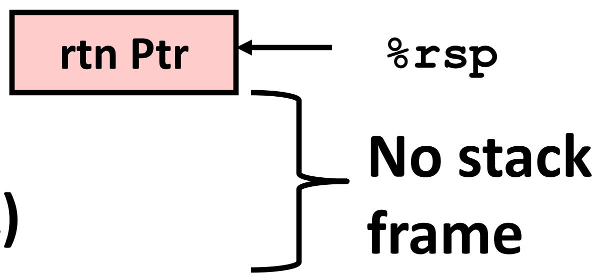
- **Arguments passed to functions via registers**
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well
- **All references to stack frame via stack pointer**
  - Eliminates need to update `%ebp/%rbp`
- **Other Registers**
  - 6 callee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



More on x86-64 vs. IA32 stack use

- **Same**
  - Push/Pop to save/restore register values (e.g., callee saved)
  - Sub/Add to create/delete space for local variables of function
    - when not all fit in registers
  - May allocate extra/unused space to ensure 16-byte alignment of every stack frame

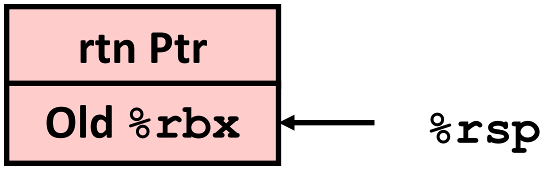
- **Different**
  - x86-64 does all stack references relative to %**rsp**
    - eliminates need to use **%ebp/%rbp** as base pointer
  - x86-64 allocates entire stack frame (if any) at once, not little-by-little
  - x86-64 has concept of usable "red zone" beyond **%rsp**

x86-64 Stack Frame Example #1

```
/* Swap a[i] and a[j]
   Compute difference */
void swap_ele_diff(long a[],
                   long i, long j) {
  long diff = a[j] - a[i];
  swap(&a[i], &a[j]);
  return diff;
}
```

```
swap_ele_diff:
  pushq   %rbx
  leaq    (%rdi,%rdx,8), %rdx
  leaq    (%rdi,%rsi,8), %rdi
  movq    (%rdx), %rbx
  subq    (%rdi), %rbx
  movq    %rdx, %rsi
  call    swap
  movq    %rbx, %rax
  popq    %rbx
  ret
```

- Keeps `diff` in callee saved register
- Uses push & pop to save/restore

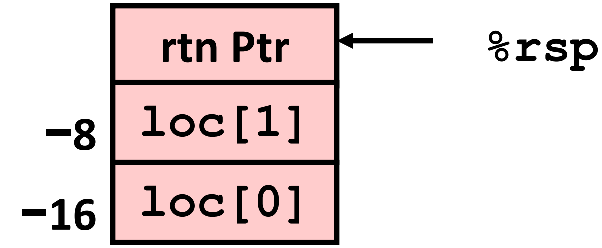


x86-64 Locals in the Red Zone

```
/* 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, -16(%rsp)
  movq   (%rsi), %rax
  movq   %rax, -8(%rsp)
  movq   -8(%rsp), %rax
  movq   %rax, (%rdi)
  movq   -16(%rsp), %rax
  movq   %rax, (%rsi)
  ret
```

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



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

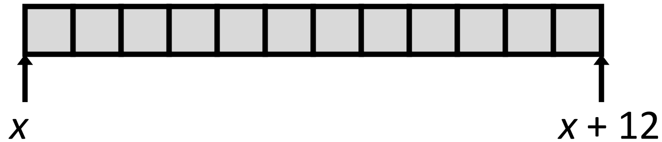
Array Allocation

- **Basic Principle**

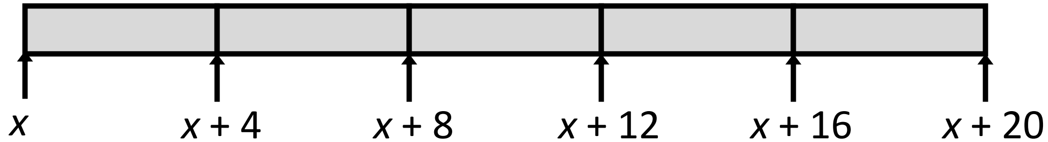
  T **`A[`***L***`];`**

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

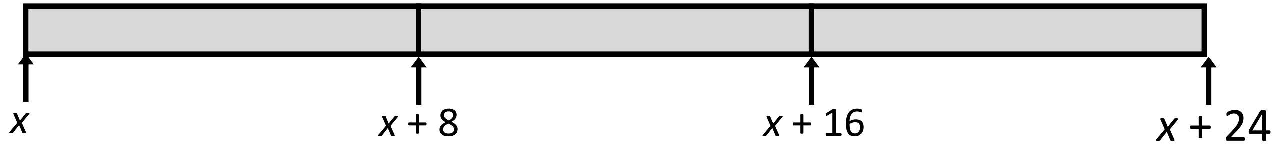
`char string[12];`



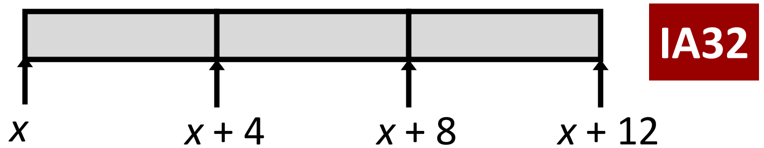
`int val[5];`

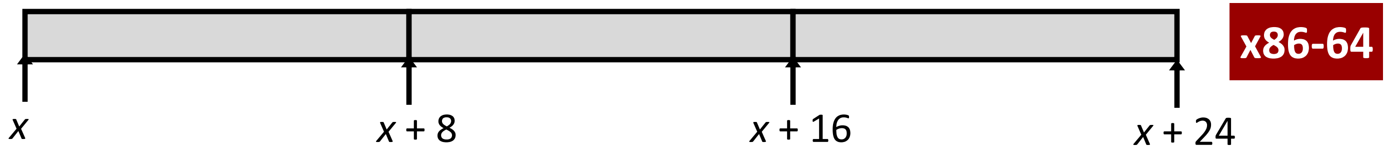


`double a[3];`



`char *p[3];`





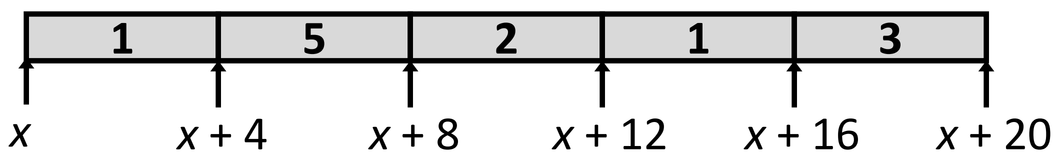
Array Access

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**

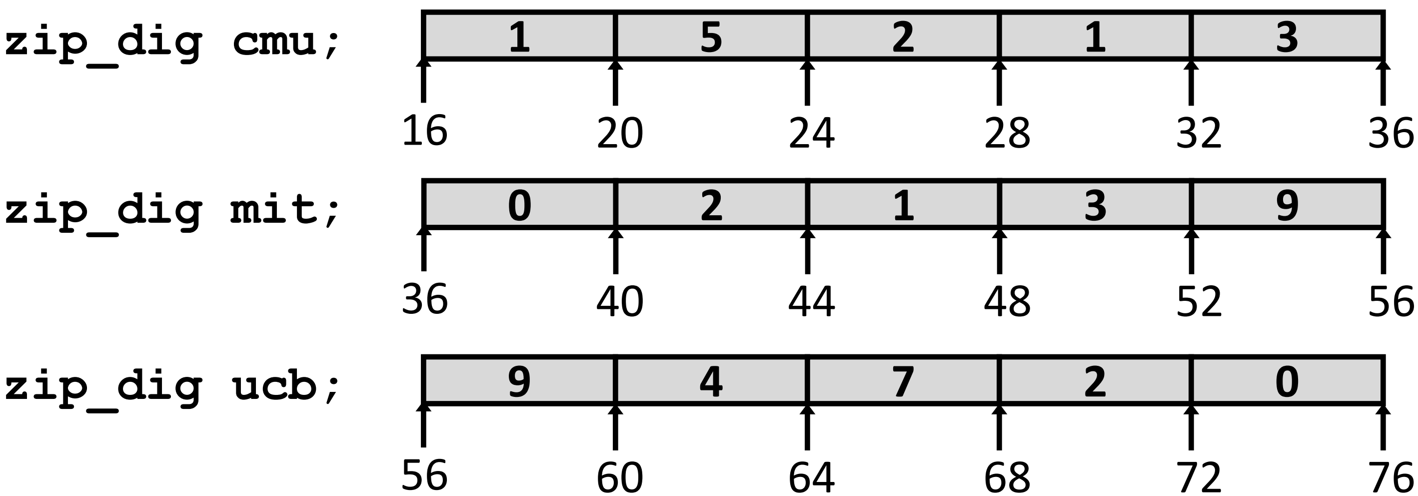
`int val[5];`



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

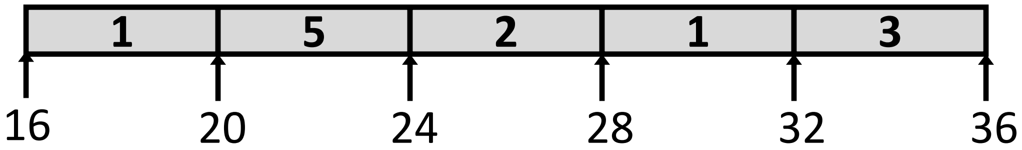
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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 digit)
{
  return z[digit];
}
```

IA32

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

- 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
```

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

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

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

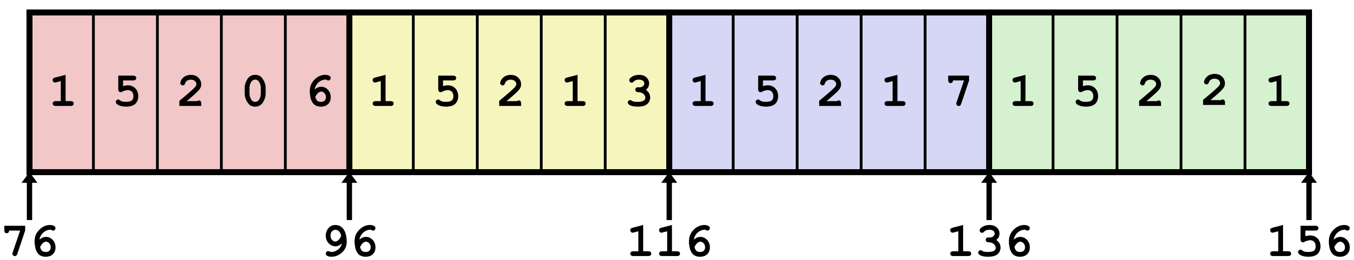
A [0] [0]	• • •	A [0] [C-1]	A [1] [0]	• • •	A [1] [C-1]	• • •	A [R-1] [0]	• • •	A [R-1] [C-1]

4*R*C Bytes

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];
```

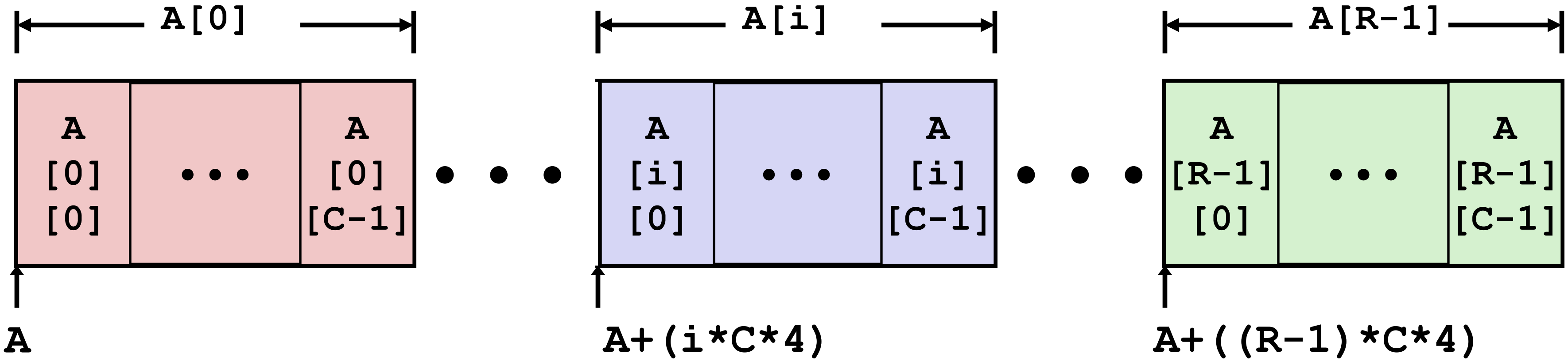


- **"`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"**
  - Variable **`pgh`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- **"Row-Major" ordering of all elements in memory**

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*)

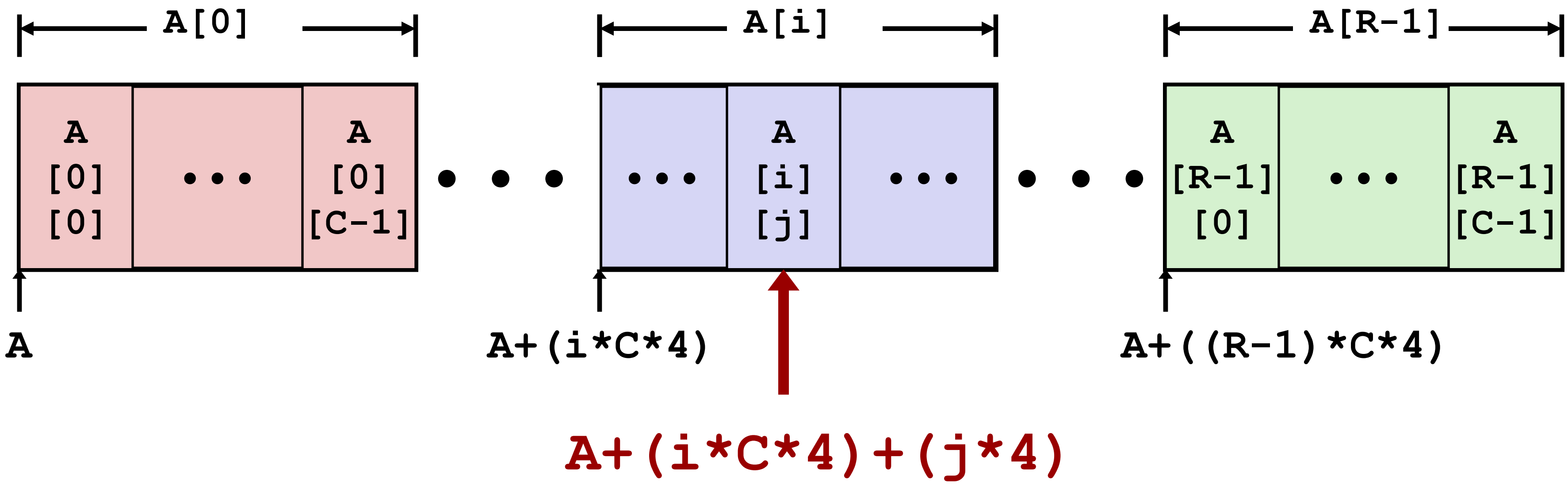
```
int A[R][C];
```



Nested Array Element 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];
```

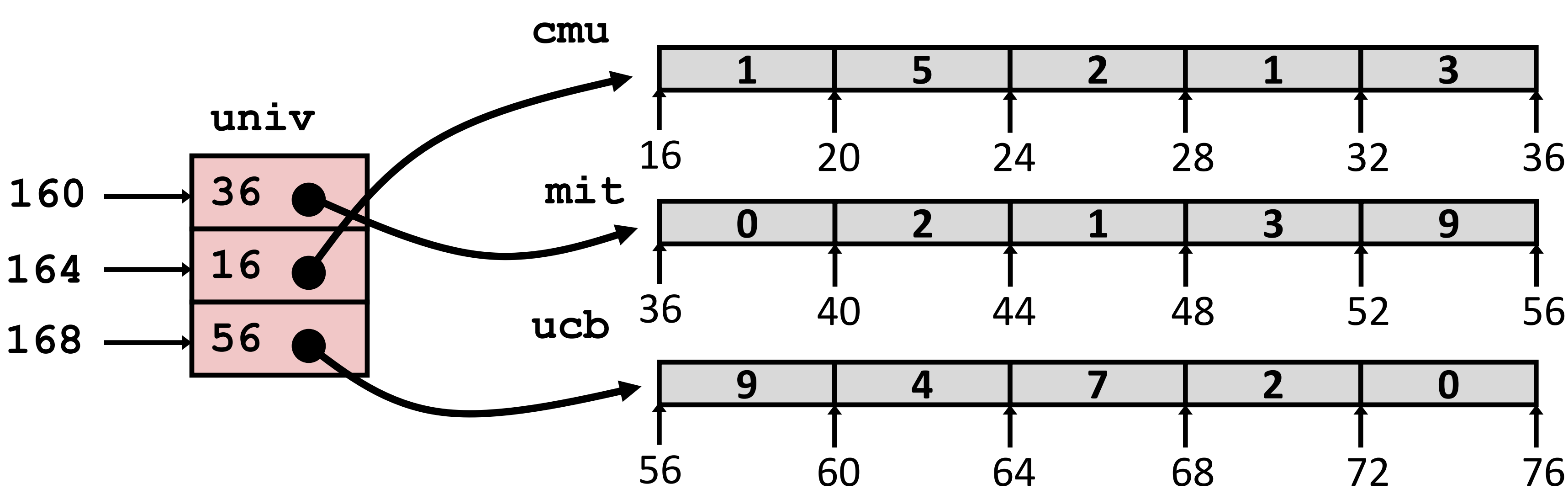


Multi-Level Array Example

```
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 digit)
{
  return univ[index][digit];
}
```

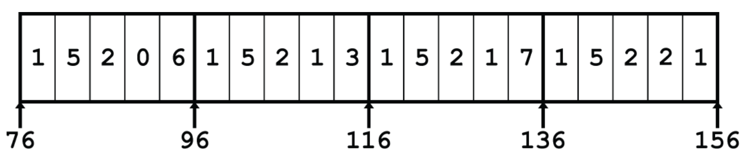
```
  movl   8(%ebp), %eax              # index
  movl   univ(,%eax,4), %edx        # p = univ[index]
  movl   12(%ebp), %eax             # digit
  movl   (%edx,%eax,4), %eax        # p[digit]
```

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

Array Element Accesses

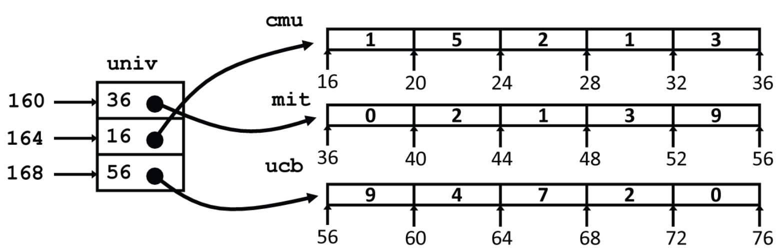
Nested array

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



Multi-level array

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



Accesses looks similar in C, but addresses very different:

`Mem[pgh+20*index+4*digit]` **`Mem[Mem[univ+4*index]+4*digit]`**

N X N Matrix Code

- **Fixed dimensions**
  - Know value of N at compile time
- **Variable dimensions, explicit indexing**
  - Traditional way to implement dynamic arrays
- **Variable dimensions, implicit indexing**
  - Now supported by gcc

```
#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];
}
```

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

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

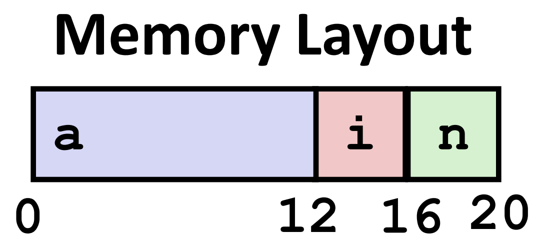
```
   movl   8(%ebp), %eax       # n
   sall   $2, %eax            # n*4
   movl   %eax, %edx          # n*4
   imull  16(%ebp), %edx      # i*n*4
   movl   20(%ebp), %eax      # j
   sall   $2, %eax            # j*4
   addl   12(%ebp), %eax      # a + j*4
   movl   (%eax,%edx), %eax # *(a + j*4 + i*n*4)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

Today

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

Structure Allocation

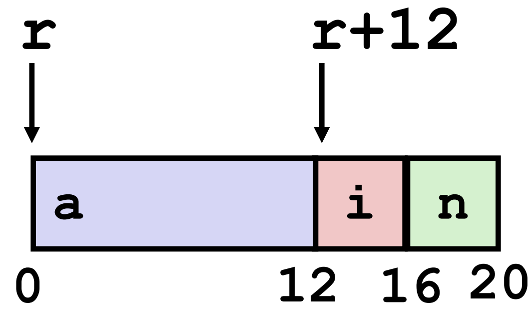
```
struct rec {
  int a[3];
  int i;
  struct rec *n;
};
```



- **Concept of structures in C**
  - 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 to structure is memory address of first byte of structure
  - Access elements with offsets

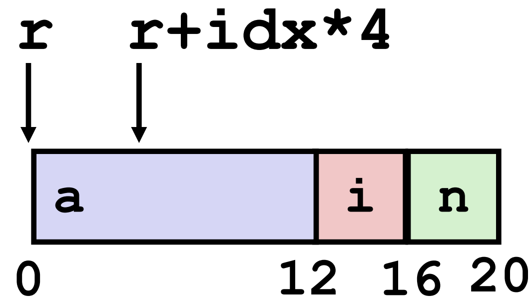
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

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];
}
```

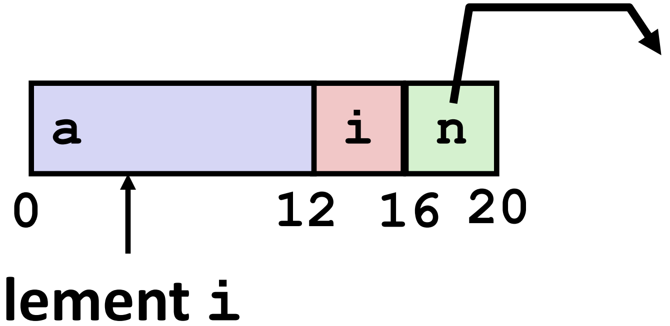
```
movl   12(%ebp), %eax   # Get idx
sall   $2, %eax         # idx*4
addl   8(%ebp), %eax    # r+idx*4
```

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;
  }
}
```

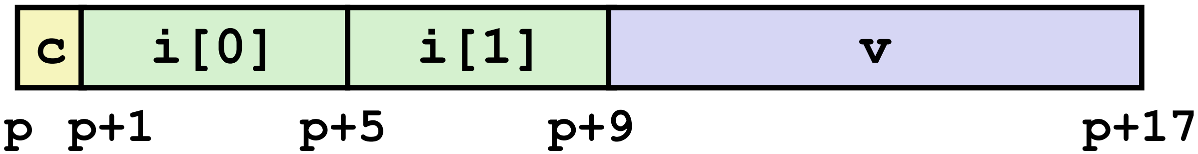


Register	Value
%edx	r
%ecx	val

```
.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
```

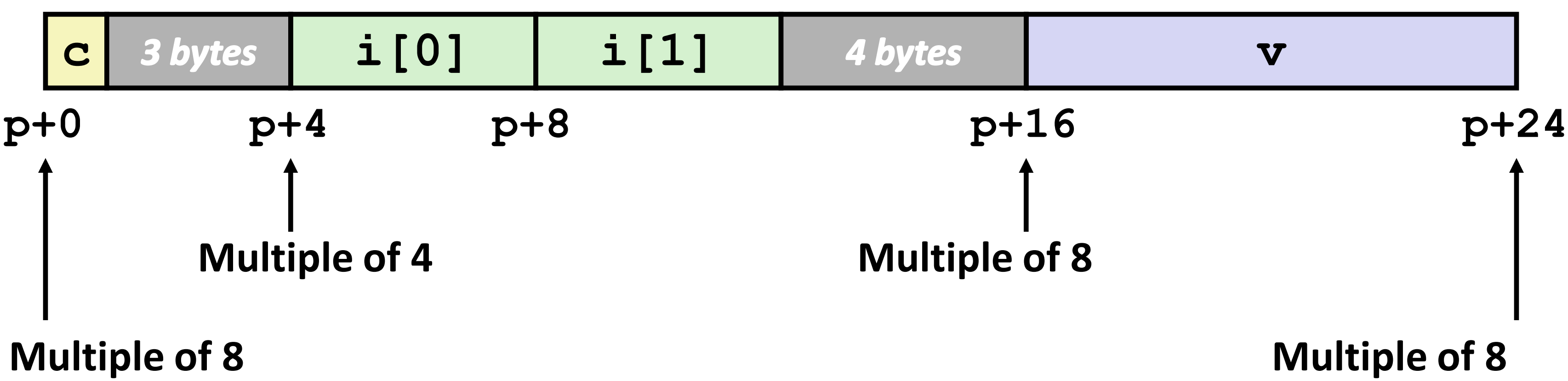
Structures & Alignment

- **Unaligned Data**




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

- **Aligned Data**
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***



Alignment Principles

- **Aligned Data**
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***
  - Required on some machines; advised on IA32
    - treated differently by IA32 Linux, x86-64 Linux, and Windows!
- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory trickier when datum spans 2 pages
- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

Specific Cases of Alignment (IA32)

- **1 byte: `char`, …**
  - no restrictions on address
- **2 bytes: `short`, …**
  - lowest 1 bit of address must be 0_2
- **4 bytes: `int`, `float`, `char *`, …**
  - lowest 2 bits of address must be 00_2
- **8 bytes: `double`, …**
  - Windows (and most other OS's & instruction sets):
    - lowest 3 bits of address must be 000_2
  - Linux:
    - lowest 2 bits of address must be 00_2
    - i.e., treated the same as a 4-byte primitive data type
- **12 bytes: `long double`**
  - Windows (GCC), Linux:
    - lowest 2 bits of address must be 00_2
    - i.e., treated the same as a 4-byte primitive data type

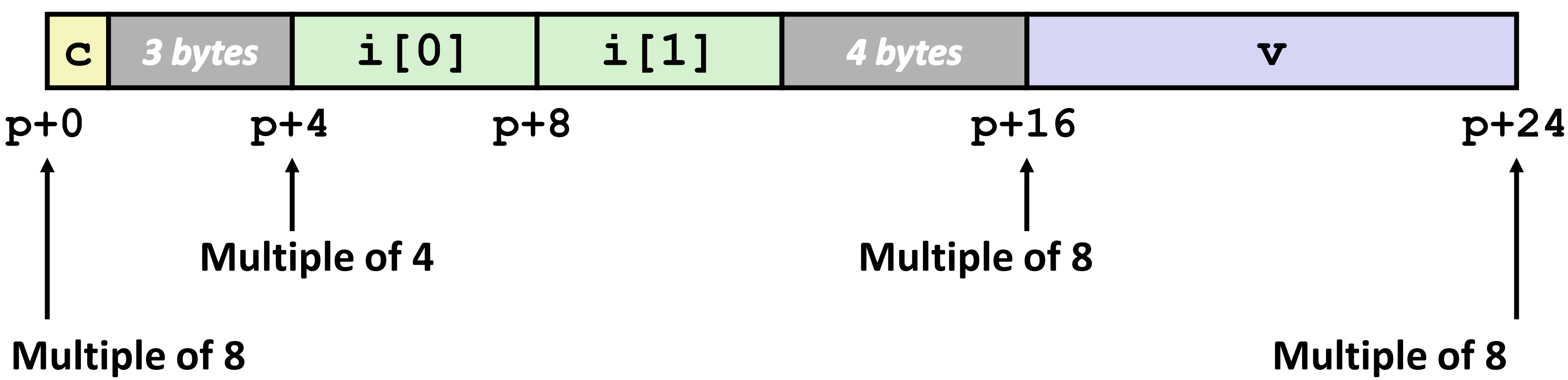
Specific Cases of Alignment (x86-64)

- **1 byte: `char`, …**
  - no restrictions on address
- **2 bytes: `short`, …**
  - lowest 1 bit of address must be 0_2
- **4 bytes: `int`, `float`, …**
  - lowest 2 bits of address must be 00_2
- **8 bytes: `double`, `long`, `char *`, …**
  - lowest 3 bits of address must be 000_2
- **16 bytes: `long double`** (GCC on Linux or Windows)
  - lowest 4 bits of address must be 0000_2

Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each element's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement **K**
    - **K** = Largest alignment of any element
  - Initial address & structure length must be multiples of **K**
- **Example (under Windows or x86-64):**
  - K = 8, due to **`double`** element

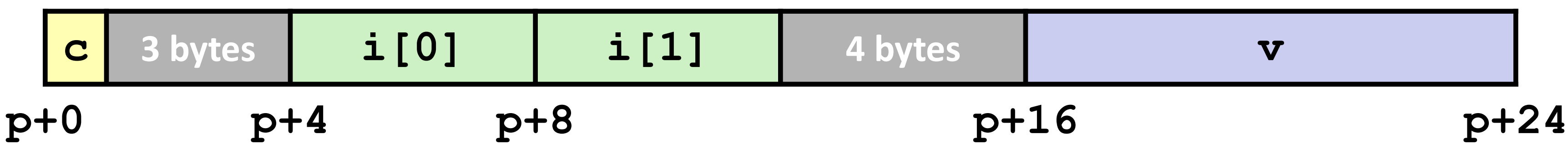
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



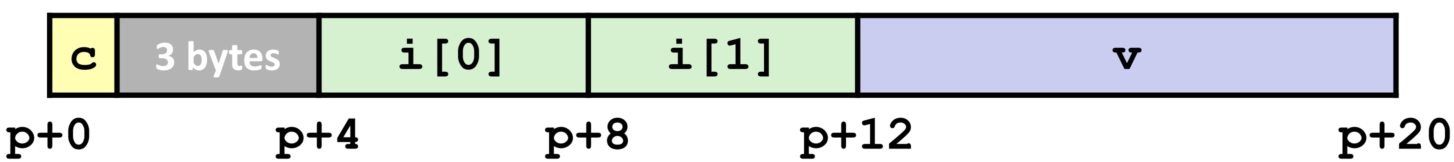
Different Alignment Conventions

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

- **Windows, x86-64**
  - K = 8, due to **`double`** element



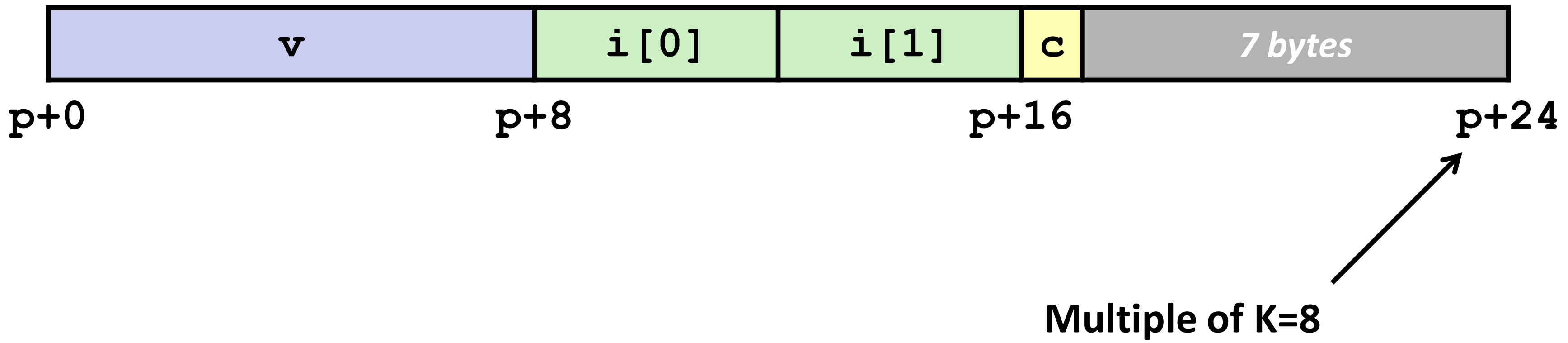
- **IA32 Linux**
  - K = 4; **`double`** treated like a 4-byte data type



Meeting Overall Alignment Requirement (Windows, x86-64)

- For largest alignment requirement K
- Overall structure must be multiple of K

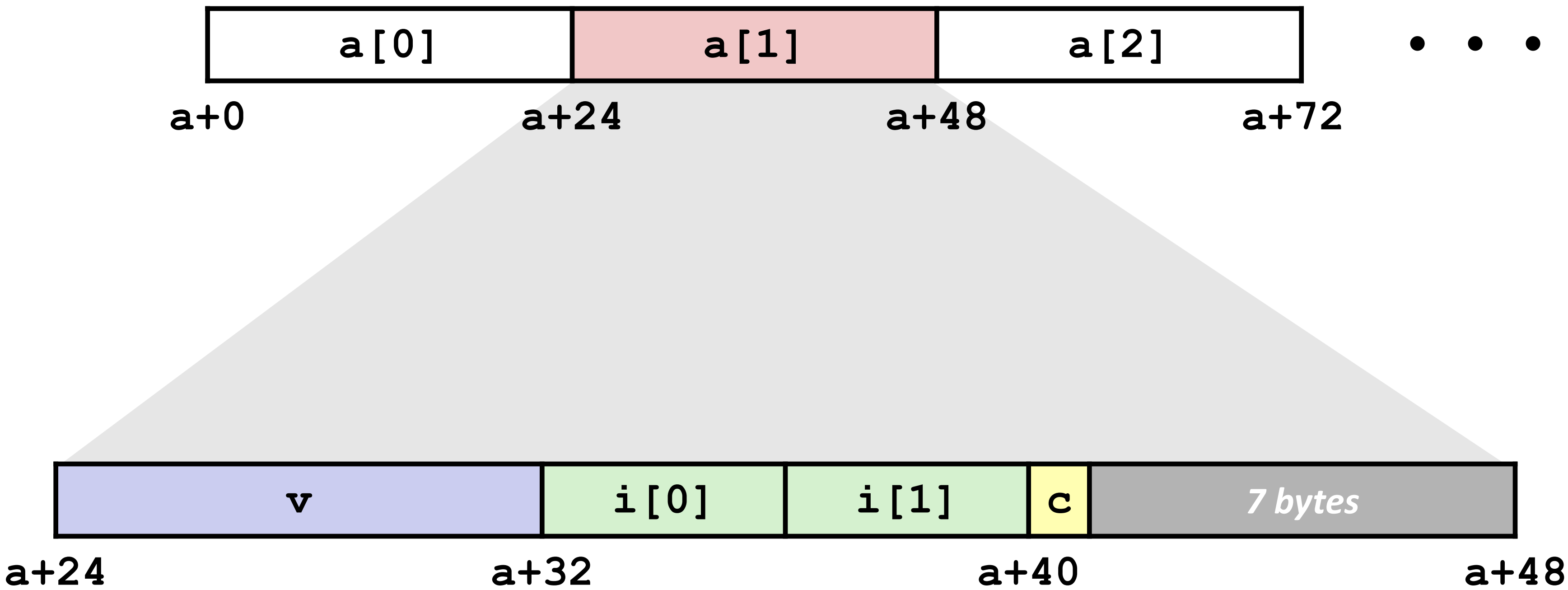
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures (Windows, x86-64)

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

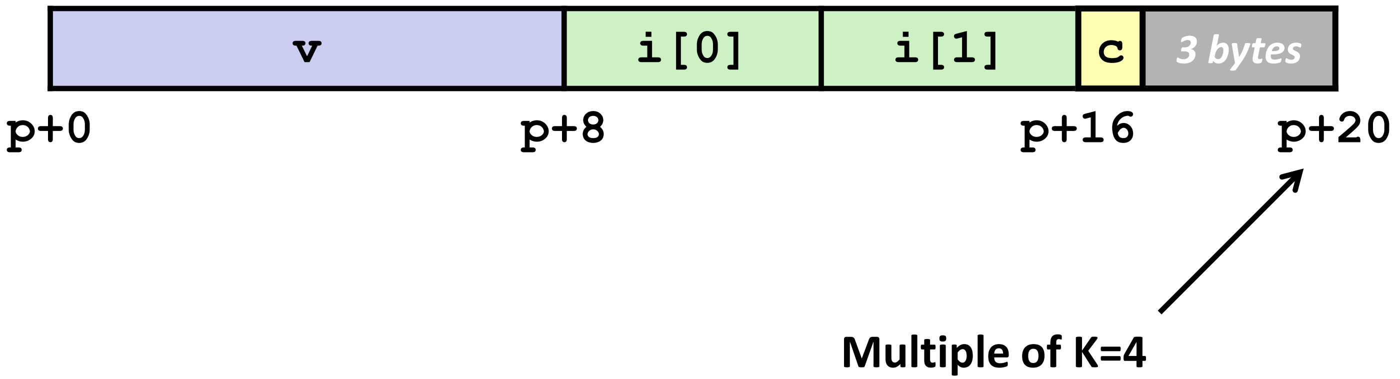
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Meeting Overall Alignment Requirement (IA32 Linux)

- **For largest alignment requirement K**
- **Overall structure must be multiple of K**
  - Up to maximum of K=4

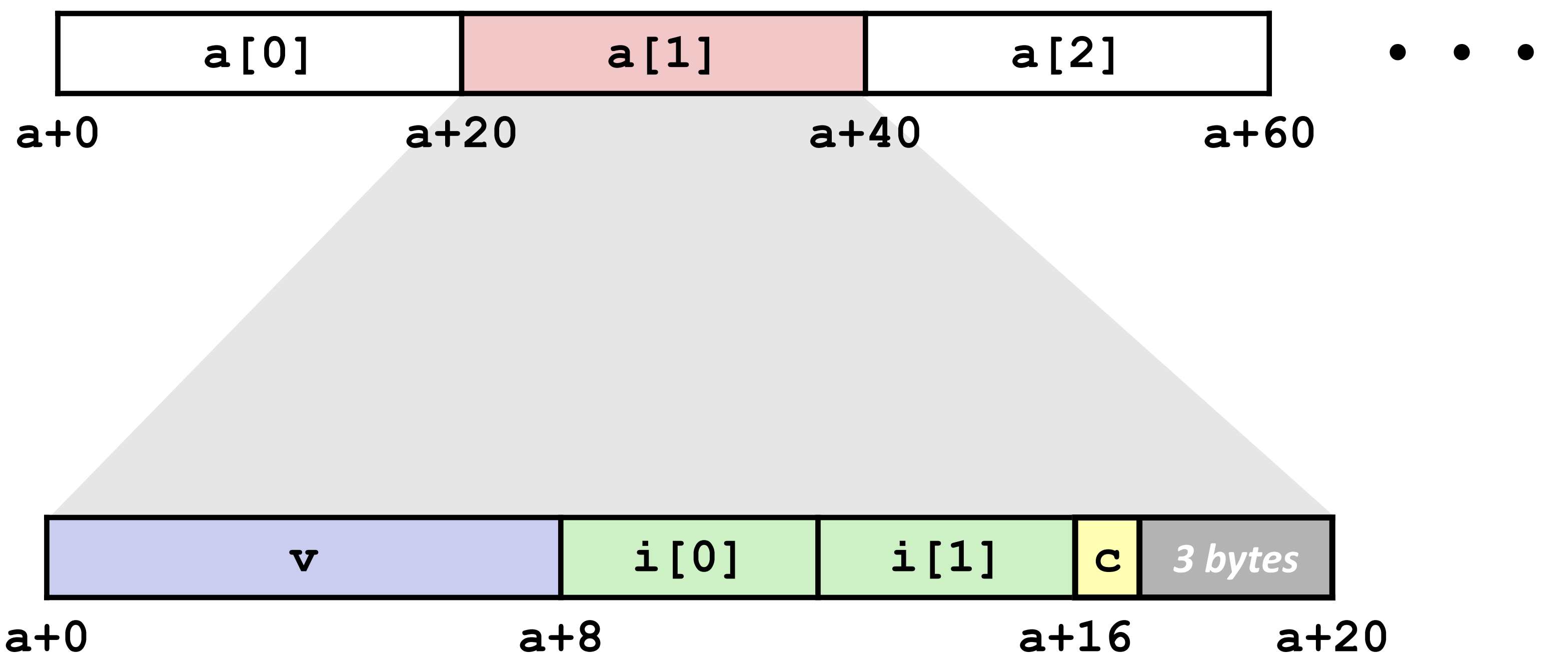
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures (IA32 Linux)

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

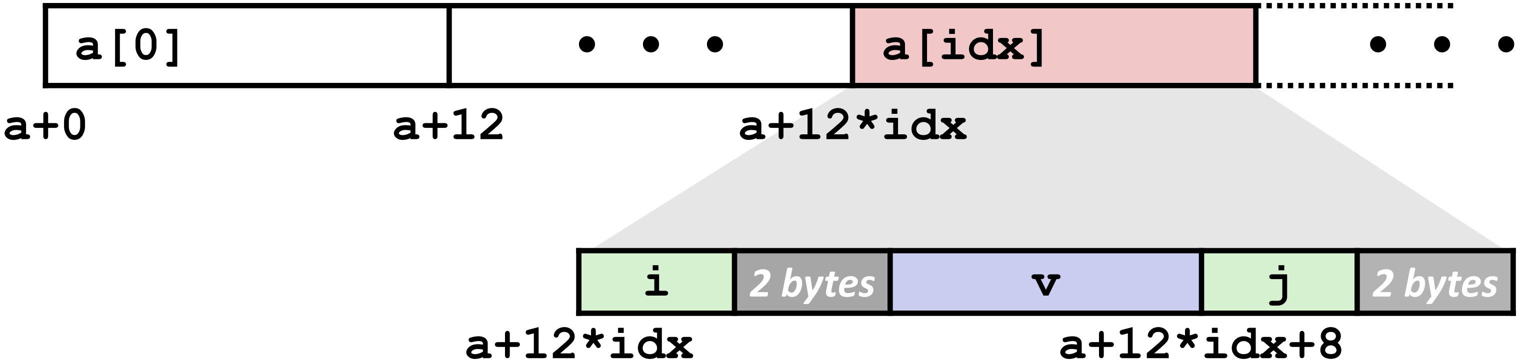
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

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

- **Compute array offset 12*idx**
  - `sizeof(S3)`, including alignment spacers
- **Element `j` is at offset 8 within structure**
- **Assembler gives offset `a+8`**
  - Resolved during linking




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

```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

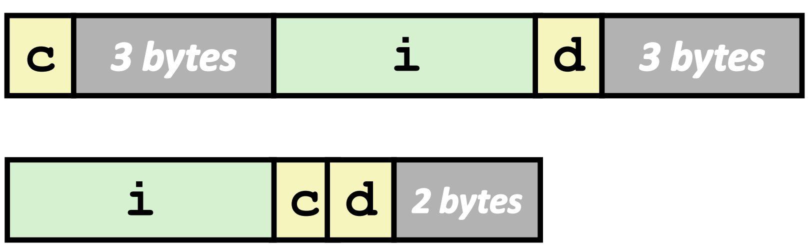
Saving Space

- Put large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- Effect (K=4)



Summary

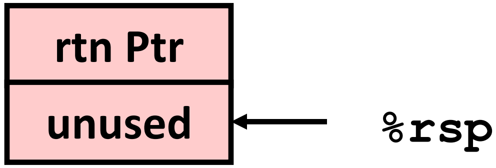
- **Procedures in x86-64**
  - Stack frame is relative to stack pointer
  - Parameters passed in registers
- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment

x86-64 NonLeaf with Unused Stack Frame

```
/* Swap a[i] and a[j] */
void swap_ele(long a[],
              long i, long j) {
  swap(&a[i], &a[j]);
}
```

- No values held while swap being invoked
- No callee saved registers needed
- 8 bytes allocated, but not used

```
swap_ele:
  subq    $8, %rsp                 # Allocate 8 bytes
  movq    %rsi, %rax               # Copy i
  leaq    (%rdi,%rdx,8), %rsi      # &a[i]
  leaq    (%rdi,%rax,8), %rdi      # &a[j]
  call    swap
  addq    $8, %rsp                 # Deallocate
  ret
```

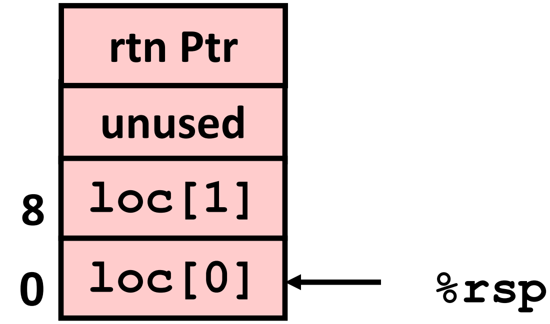


x86-64 Stack Frame Example #2

```
/* Swap a[i] and a[j] */
void swap_ele_l(long a[],
               long i, long j) {
    long *loc[2];
    long b = i & 0x1;
    loc[b] = &a[i];
    loc[1-b] = &a[j];
    swap(loc[0], loc[1]);
}
```

- Must allocate space on stack for array `loc`
- Uses `subq` to allocate, `addq` to deallocate

```
swap_ele_l:
  subq    $24, %rsp
  movq    %rsi, %rax
  andl    $1, %eax
  leaq    (%rdi,%rsi,8), %rcx
  movq    %rcx, (%rsp,%rax,8)
  movl    $1, %ecx
  subq    %rax, %rcx
  leaq    (%rdi,%rdx,8), %rdx
  movq    %rdx, (%rsp,%rcx,8)
  movq    8(%rsp), %rsi
  movq    (%rsp), %rdi
  call    swap
  addq    $24, %rsp
  ret
```

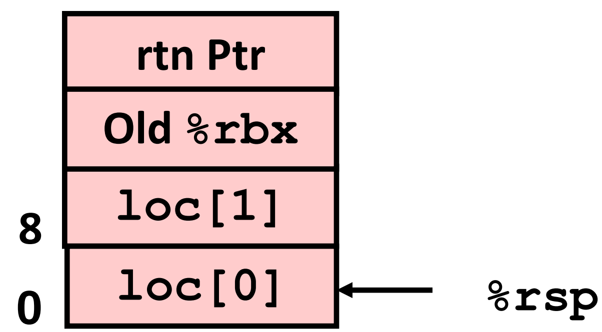


x86-64 Stack Frame Example #3

```
/* Swap a[i] and a[j] */
long swap_ele_l_diff(long a[],
                     long i, long j) {
    long *loc[2];
    long b = i & 0x1;
    long diff = a[j] - a[i];
    loc[b] = &a[i];
    loc[1-b] = &a[j];
    swap(loc[0], loc[1]);
    return diff
}
```

```
swap_ele_l_diff:
   pushq    %rbx
   subq     $16, %rsp
   . . .
   call     swap
   . . .
   addq     $16, %rsp
   popq     %rbx
   ret
```

- Have both callee saved register & local variable allocation
- Use both push/pop and sub/add



Interesting Features of Stack Frame

- **Allocate entire frame at once**
  - All stack accesses can be relative to `%rsp`
  - Do by:
    - pushing callee saved registers (if needed)
    - decrementing stack pointer (if needed)

- **Simple deallocation**
  - Do by:
    - Incrementing stack pointer (possibly)
    - Popping callee saved registers (possibly)
  - No base/frame pointer needed

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`**

  - Note: Windows Visual C/C++ compiler treats **`long double`** as regular, 8-byte **`double`**. GCC on Windows uses extended precision

Pointer Loop Example (IA32)

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

```
  movl    8(%ebp), %eax    # z
  leal    20(%eax), %edx   # zend
.L9:                       # loop:
  addl    $1, (%eax)       # *z += 1
  addl    $4, %eax         # z++
  cmpl    %eax, %edx       # zend:z
  jne     .L9              # if !=, goto loop
```

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)`**

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)`