

CS 370: Computer Organization

Data Structures and Alignment

Instructor:

Professor Stephen P. Carl

Review: Basic Data Types

Integral

Stored and 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

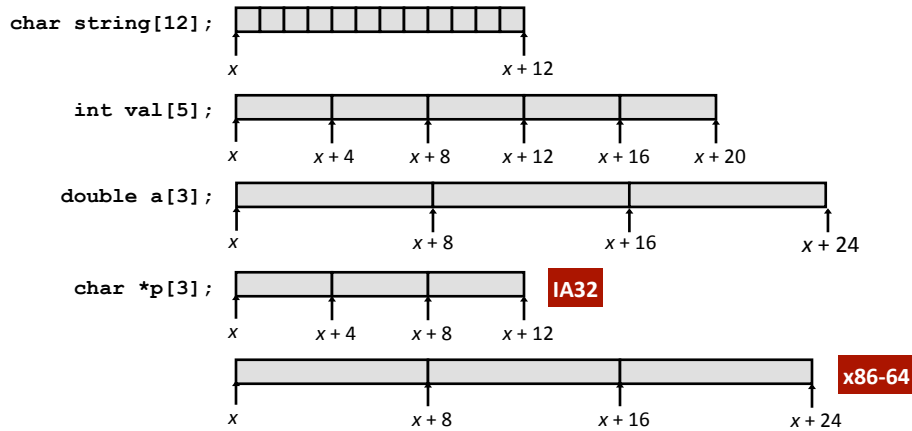
Stored & operated on in floating point registers

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

Array Allocation

Basic Principle:

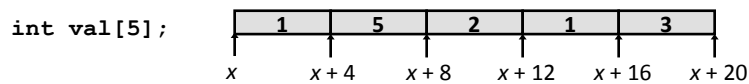
$T\ A[L];$ // Array of data type T and length L
 Contiguously allocated region of $L * \text{sizeof}(T)$ bytes



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^*

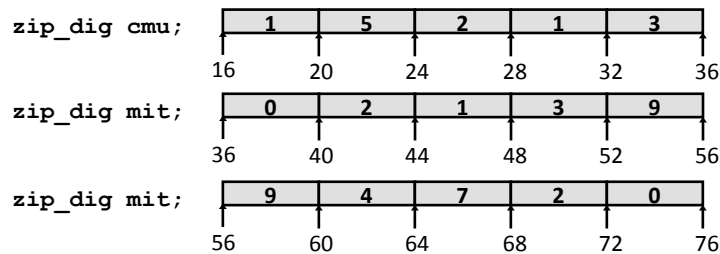


Reference	Type	Value
<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	x
<code>val+1</code>	<code>int *</code>	$x+4$
<code>&val[2]</code>	<code>int *</code>	$x+8$
<code>val[5]</code>	<code>int</code>	??
<code>*(val+1)</code>	<code>int</code>	5
<code>val + i</code>	<code>int *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

Array Example

```
typedef int zip_dig[5];

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

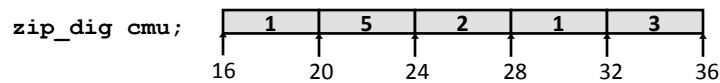


Declaration “`zip_dig cmu`” is equivalent to “`int cmu[5]`”

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

asm

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

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit is at location `%rdi + 4*%rsi`
- Use memory reference `(%rdi,%rsi,4)`

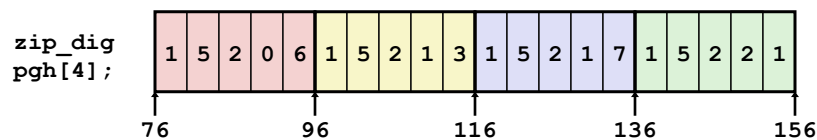
Array Loop Example

```
void zincr(zip_dig z) {
    size_t i;
    for (i = 0; i < 5; i++)
        z[i]++;
}
```

```
# %rdi holds 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
```

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]”

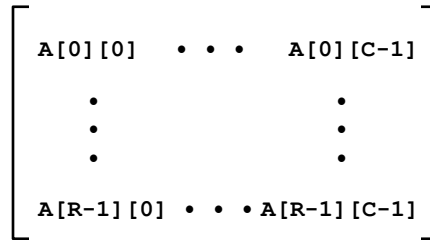
- Variable `pgh`: array of 4 elements, allocated contiguously
- Each element is an array of 5 `int`'s, allocated contiguously

“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



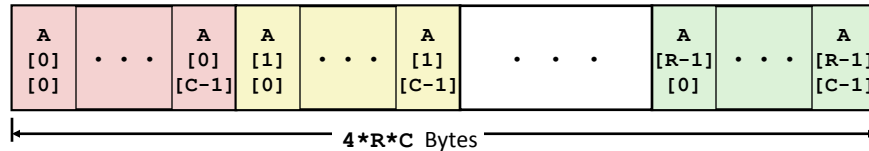
Array Size

- $R * C * K$ bytes

Arrangement

- Row-Major Ordering

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

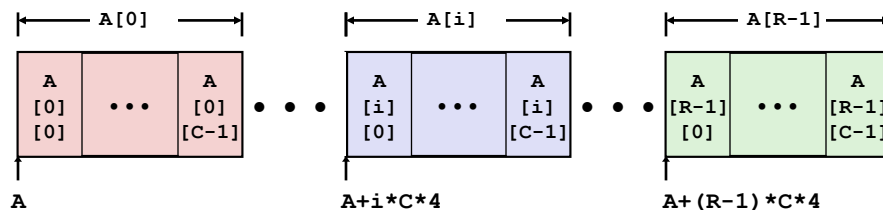


Nested Array Row Access

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 }};
```

```
# %rdi holds index
leal (%rdi,%rdi,4),%rax # 5 * index
leal pgh(,%rax,4),%rax # pgh + (20 * index)
```

Row Vector

- `pgh[index]` is an array of 5 int's
- Starting address is: `pgh+20*index`

Assembly Code

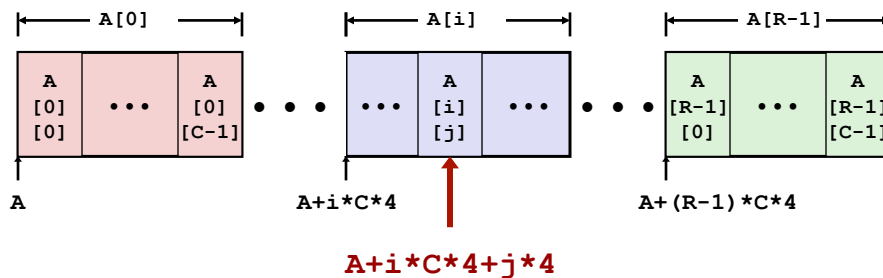
- Computes and returns address
- Compute as: `pgh + 4*(index+4*index)`

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

1	5	2	0	6	1	5	2	1	3	1	5	2	1	7	1	5	2	2	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

pgh

```
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),%rax    # M[pgh + 4*(5*index+dig)]
```

Array Elements

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

Using Nested Arrays

Strengths:

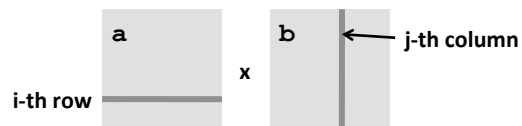
- C compiler handles doubly subscripted arrays
- Generates very efficient code
- Avoids multiply in index computation

Limitation:

- Only works for fixed array size

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

```
/* Compute element i,k of
   fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
 int i, int k)
{
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += a[i][j]*b[j][k];
    return result;
}
```



N X N Matrix Code

1) Fixed dimensions

- when value of N known at compile time

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

2) Variable dimensions, explicit indexing

- Traditional way to implement dynamic arrays

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

3) Variable dimensions, implicit indexing

- Now supported by gcc

```
/* 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];
}
```

Example: 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, size_t i, size_t j) {
    return a[i][j];
}
```

```
# a in %rdi, i in %rsi, j in %rdx
salq    $6, %rsi          # 64*i
addq    %rsi, %rdi        # a + 64*i
movl    (%rdi,%rdx,4), %eax # M[a + 64*i + 4*j]
ret
```


n X n Matrix Access

Array Elements

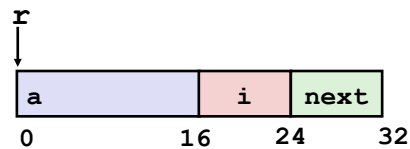
- Address $A + i * (C * K) + j * K$
- $C = n, K = 4$
- Must perform integer multiplication

```
/* Get element a[i][j] */
int var_ele(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
```

Structure Representation

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

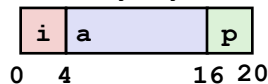


- **Structure represented as block of memory**
Big enough to hold all of the fields
- **Fields ordered according to declaration**
Even if another ordering could yield a more compact representation
- **Compiler determines overall size + positions of fields**
Machine-level program has no understanding of the structures in the source code

Example Structure

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

Memory Layout



Accessing Structure Member r->i

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

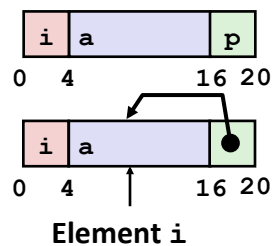
Assembly Code

```
movl %rsi, (%rdi)    # Mem[r] = val
```

Referencing Fields in Structure

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

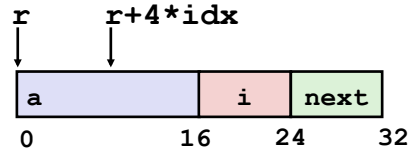
```
void
set_p(struct rec *r){
    r->i = 1;
    r->a[0] = 10;
    r->p = NULL;
}
```



```
# %rdi = r
movl $1, (%rdi)    # r->i = 1
movl $10, 4(%rdi)  # r->a[0] = 10
movq $0, 16(%rdi)  # Update r->p
```

Generating Pointer to Structure Member

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



Address of array element:

- Offset of each structure member determined at compile time
- Compute as $r + 4 * idx$

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

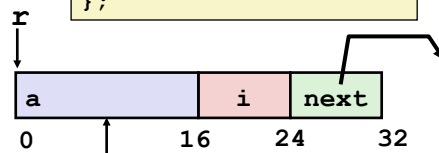
```
# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```

Example: Linked List

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];
    int i;
    struct rec *next;
};
```



Element i

Register	Value
%rdi	r
%rsi	val

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

Sewanee: The University of the South

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

Sewanee: The University of the South

Alignment Principles

Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on x86-64

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 (x86-64)

1 byte: char, ...

- no restrictions on address

2 bytes: short, ...

- lowest 1 bit of address must be 0₂

4 bytes: int, float, ...

- lowest 2 bits of address must be 00₂

8 bytes: double, long, char *, ...

- lowest 3 bits of address must be 000₂

16 bytes: long double (GCC on Linux)

- lowest 4 bits of address must be 0000₂

Satisfying Alignment with Structures

Within structure:

- Must satisfy each element's alignment requirement

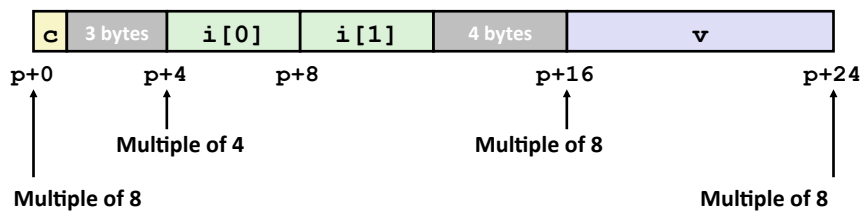
Overall structure placement

- Each structure has alignment requirement K
 - K = Largest alignment of any element
- Initial address & structure length must be multiples of K

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

Example:

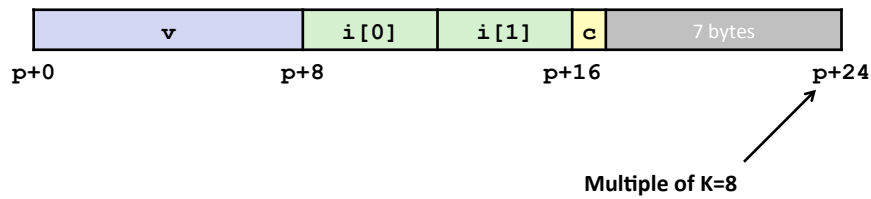
- K = 8, due to double element



Meeting Overall Alignment Requirement

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

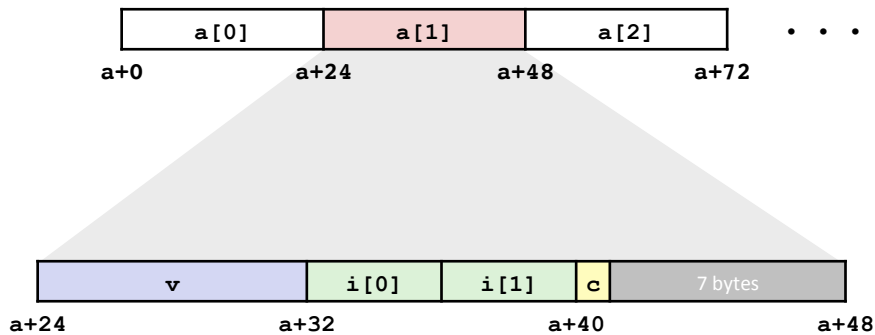
```
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```



Arrays of Structures

- 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];
```

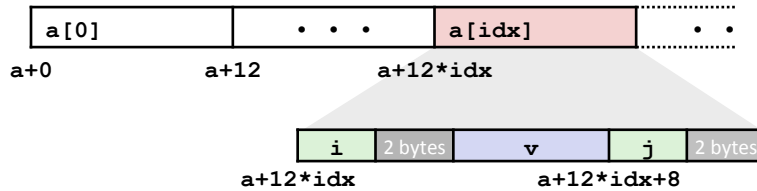
Compute array offset $12 * idx$

- `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;
}
```

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

- Example ($K=4$):

