Machine-Level Representation

CSCI 2021: Machine Architecture and Organization

Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

With Slides from Bryant and O'Hallaron

University of Minnesota

Arrays

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

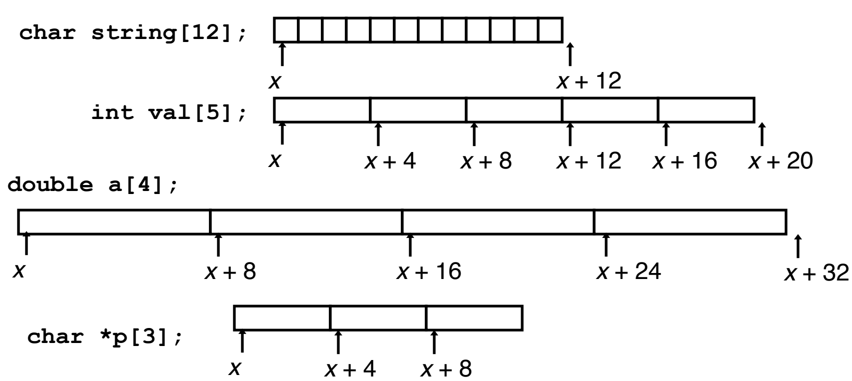
Intel	ASM	Bytes	C
Single	`s`	4	`float`
Double	`l`	8	`double`
Extended	`t`	10/12/16	`long double`

Array Allocation

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

`int val[5];`

1	5	2	1	3	
x	*x* + 4	*x* + 8	*x* + 12	*x* + 16	*x* + 20

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

```
typedef int zip_dig[5];

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

`zip_dig cmu;`

1	5	2	1	3	
16	20	24	28	32	36

`zip_dig mit;`

0	2	1	3	9	
36	40	44	48	52	56

`zip_dig umn;`

5	5	4	5	5	
56	60	64	68	72	76

Declaration "`zip_dig umn`" equivalent to "`int umn[5]`"

Example arrays were allocated in successive 20 byte blocks

- Not guaranteed to happen in general

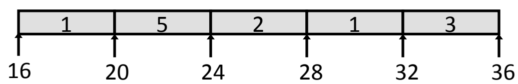
Array Accessing Example

Computation

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

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

`zip_dig cmu;`



Memory Reference Code

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


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
```

Referencing Examples

`zip_dig cmu;`

1	5	2	1	3	
16	20	24	28	32	36

`zip_dig mit;`

0	2	1	3	9	
36	40	44	48	52	56

`zip_dig umn;`

5	5	4	5	5	
56	60	64	68	72	76

Code Does Not Do Any Bounds Checking!

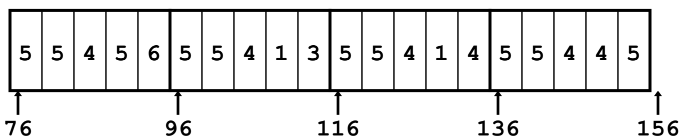
Reference	Address	Value	Guaranteed?
`mit[3]`	`36 + 4* 3 = 48`	3	Yes
`mit[5]`	`36 + 4* 5 = 56`	5	No
`mit[-1]`	`36 + 4*-1 = 32`	3	No
`cmu[15]`	`16 + 4*15 = 76`	??	No

- Out of range behavior implementation-dependent
- No guaranteed relative allocation of different arrays

Nested Array Example

```
#define PCOUNT 4
zip_dig mpls[PCOUNT] =
  {{5, 5, 4, 5, 5},
   {5, 5, 4, 1, 3 },
   {5, 5, 4, 1, 4 },
   {5, 5, 4, 4, 5 }};
```

`zip_dig mpls[4];`



Declaration "`zip_dig mpls[4]`" equivalent to "`int mpls[4][5]`"

- Variable `mpls` denotes array of 4 elements
  - Allocated contiguously
  - Each element is an array of 5 `int`'s
    - Allocated contiguously
- "Row-Major" ordering of all elements guaranteed

Nested Array Allocation

Declaration: *T* `A[`*R*`][`*C*`];`

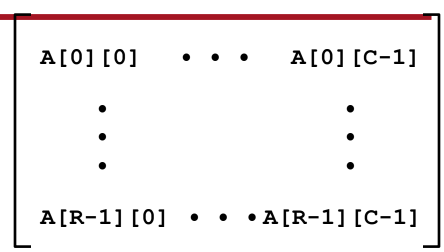
- Array of data type *T*
- *R* rows, *C* columns
- Type *T* element requires *K* bytes

Array Size

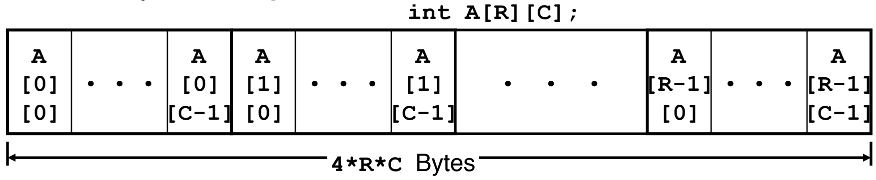
- *R* * *C* * *K* bytes

Arrangement

- Row-Major Ordering



`int A[R][C];`

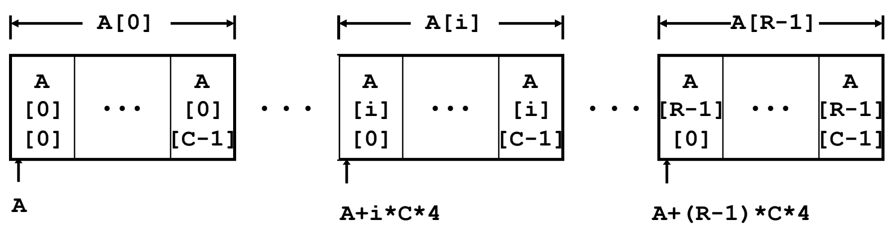


Nested Array Row Access

Row Vectors

- `A[i]` is array of *C* elements
- Each element of type *T*
- Starting address `A +` *i * C * K*

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



Nested Array Row Access Code

```
int *get_mpls_zip(int index)
{
  return mpls[index];
}
```

Row Vector

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

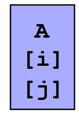
Code

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

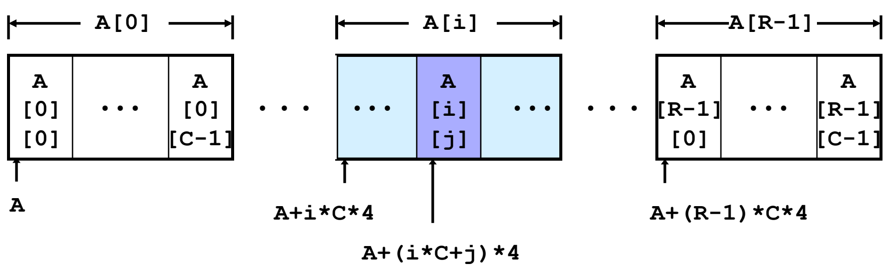
```
# %eax = index
leal (%eax,%eax,4),%eax # 5 * index
leal mpls(,%eax,4),%eax # mpls + (20 * index)
```

Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type *T*
  - Address `A +` *(i * C + j) * K*




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



Nested Array Element Access Code

Array Elements

- `mpls[index][dig]` is `int`
- Address: `mpls + 20*index + 4*dig`

Code

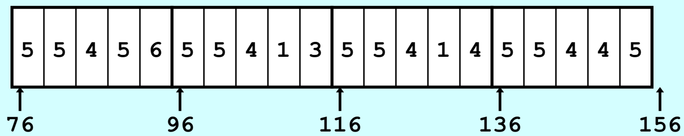
- Computes address
- `mpls + 4*dig + 4*(index+4*index)`
- `movl` performs memory reference

```
int get_mpls_digit
  (int index, int dig)
{
  return mpls[index][dig];
}
```

```
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx          # 4*dig
leal (%eax,%eax,4),%eax       # 5*index
movl mpls(%edx,%eax,4),%eax  # *(mpls + 4*dig + 20*index)
```

Strange Referencing Examples

`zip_dig mpls[4];`



Reference	Address	Value	Guaranteed?
`mpls[3][3]`	`76+20*3+4*3 = 148`		Yes
`mpls[2][5]`	`76+20*2+4*5 = 136`		Yes
`mpls[2][-1]`	`76+20*2+4*-1 = 112`		Yes
`mpls[4][-1]`	`76+20*4+4*-1 = 152`		Yes
`mpls[0][19]`	`76+20*0+4*19 = 152`		Yes
`mpls[0][-1]`	`76+20*0+4*-1 = 72`		No

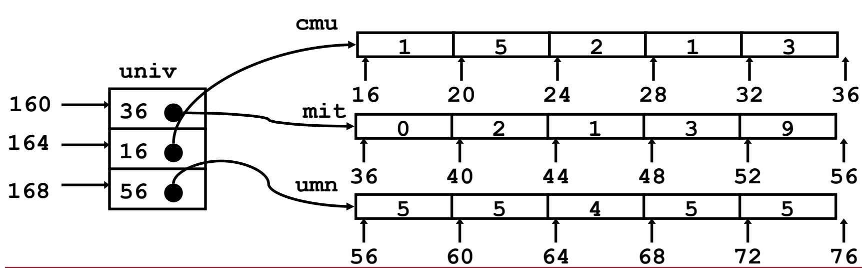
Code does not do any bounds checking
Ordering of elements within array guaranteed

Multi-Level Array Example

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

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

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



Element Access in Multi-Level Array

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

```
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]
```


Array Element Accesses

Similar C references

Different address computation

Nested Array

```
int get_mpls_digit
  (int index, int dig)
{
  return mpls[index][dig];
}
```

Element at

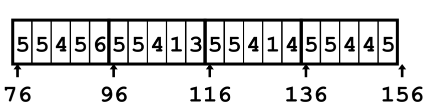
`Mem[mpls+20*index+4*dig]`

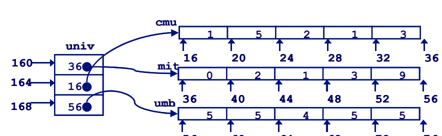
Multi-Level Array

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

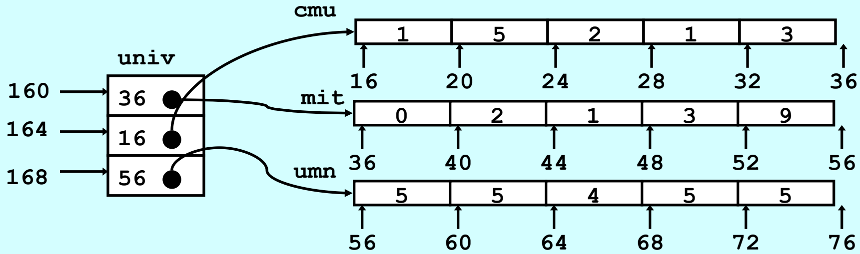
Element at

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





Strange Referencing Examples



Reference	Address	Value	Guaranteed?
`univ[2][3]`	`56+4*3 = 68`		Yes
`univ[1][5]`	`16+4*5 = 36`		No
`univ[2][-1]`	`56+4*-1 = 52`		No
`univ[3][-1]`	`??`		No
`univ[1][12]`	`16+4*12 = 64`		No

Code does not do any bounds checking
Ordering of elements in different arrays not guaranteed

N X N Matrix

- 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];
}
```

Dynamic Nested Arrays

Can create matrix of arbitrary size

Must do index computation explicitly

- Accessing single element costly
- Must do multiplication

```
int * new_var_matrix(int n)
{
  return (int *)
    calloc(sizeof(int), n*n);
}
```

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

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

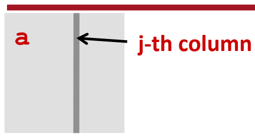
n X n Matrix Access

- **Array Elements**
  - Address **A +** *i* * (*C* * *K*) + *j* * *K*
  - C = n, K = 4

```
/* 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)
```

Optimizing Fixed Array Access




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

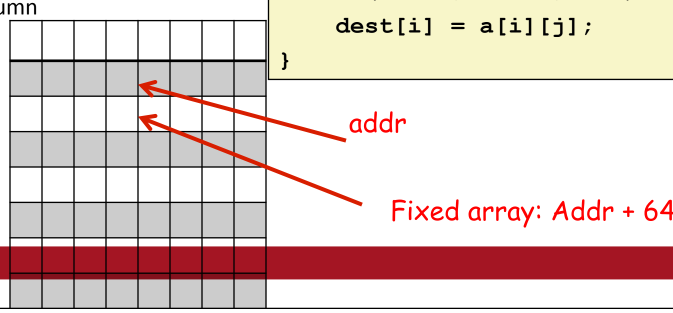
```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

Computation

- Step through all elements in column j

Optimization

- Retrieving successive elements from single column



Optimizing Fixed Array Access

Optimization

- Compute ajp = &a[i][j]
  - Initially = a + 4*j
  - Increment by 4*N

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest {
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

Register	Value
%ecx	ajp
%ebx	dest
%edx	i

```
.L8:                           # loop:
   movl  (%ecx), %eax          #   Read *ajp
   movl  %eax, (%ebx,%edx,4)   #   Save in dest[i]
   addl  $1, %edx              #   i++
   addl  $64, %ecx             #   ajp += 4*N
   cmpl  $16, %edx             #   i:N
   jne   .L8                   #   if !=, goto loop
```

Optimizing Variable Array Access

Compute ajp = &a[i][j]

- Initially = a + 4*j
- Increment by 4*n

```
/* Retrieve column j from array */
void var_column
  (int n, int a[n][n],
   int j, int *dest)
{
  int i;
  for (i = 0; i < n; i++)
    dest[i] = a[i][j];
}
```

Register	Value
%ecx	ajp
%edi	dest
%edx	i
%ebx	4*n
%esi	n

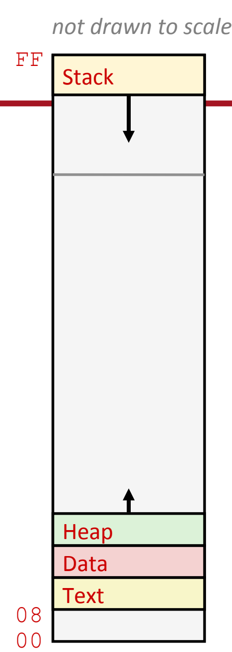
```
.L18:                          # loop:
   movl  (%ecx), %eax          #   Read *ajp
   movl  %eax, (%edi,%edx,4)   #   Save in dest[i]
   addl  $1, %edx              #   i++
   addl  $ebx, %ecx            #   ajp += 4*n
   cmpl  $edx, %esi            #   n:i
   jg    .L18                  #   if >, goto loop
```

Memory Layout

Memory Allocation Example

```
char big_array[1<<24];  /*  16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() {  return 0; }

int main()
{
 p1 = malloc(1 <<28);  /* 256 MB */
 p2 = malloc(1 << 8);  /* 256 B  */
 p3 = malloc(1 <<28);  /* 256 MB */
 p4 = malloc(1 << 8);  /* 256 B  */
 /* Some print statements ... */
}
```

Where does everything go?



IA32 Example Addresses

not drawn to scale

address range ~2^{32}

`$esp`	`0xffffbcd0`
`p3`	`0x65586008`
`p1`	`0x55585008`
`p4`	`0x1904a110`
`p2`	`0x1904a008`
`&p2`	`0x18049760`
`&beyond`	`0x08049744`
`big_array`	`0x18049780`
`huge_array`	`0x08049760`
`main()`	`0x080483c6`
`useless()`	`0x08049744`
final `malloc()`	`0x006be166`

`malloc()` is dynamically linked

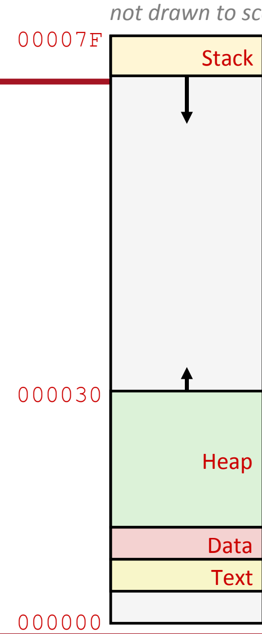
address determined at runtime

x86-64 Example Addresses

not drawn to scale

address range ~2^{47}

`$rsp`	`0x00007ffffff8d1f8`
`p3`	`0x00002aaabaadd010`
`p1`	`0x00002aaaaaadc010`
`p4`	`0x0000000011501120`
`p2`	`0x0000000011501010`
`&p2`	`0x0000000010500a60`
`&beyond`	`0x0000000000500a44`
`big_array`	`0x0000000010500a80`
`huge_array`	`0x0000000000500a50`
`main()`	`0x0000000000400510`
`useless()`	`0x0000000000400500`
final `malloc()`	`0x000000386ae6a170`

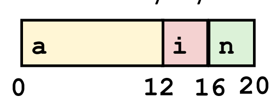


Heterogeneous Data Structures

Structure Allocation

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

Memory Layout

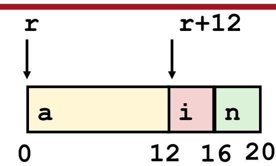


Concept

- 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**

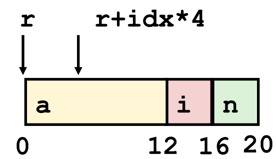
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

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];
}
```

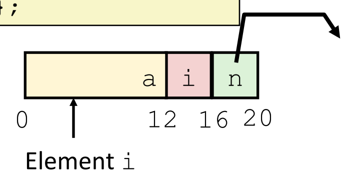
```
movl   12(%ebp), %eax    # Get idx
sall   $2, %eax          # idx*4
addl   8(%ebp), %eax     # r+idx*4
```

Following Linked List

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

- C Code

```
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
```

Alignment

Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on IA32
  - treated differently by Linux and Windows!

Motivation for Aligning Data

- Memory accessed by (aligned) double or quad-words
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory very tricky 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, 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`, `char *`, …
  - Windows & Linux:
    - lowest 3 bits of address must be 000_2
- 16 bytes: `long double`
  - Linux:
    - lowest 3 bits of address must be 000_2
    - i.e., treated the same as a 8-byte primitive data type

Satisfying Alignment with Structures

Offsets Within Structure

- Must satisfy element's alignment requirement

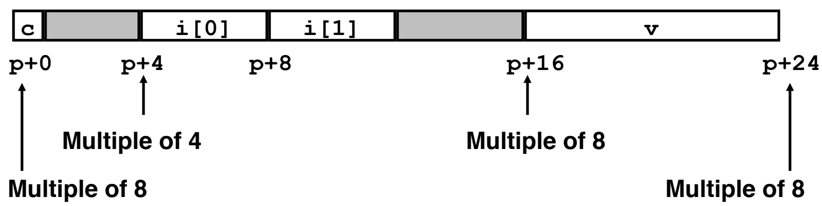
Overall Structure Placement

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

Example (under Windows):

- K = 8, due to `double` element

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

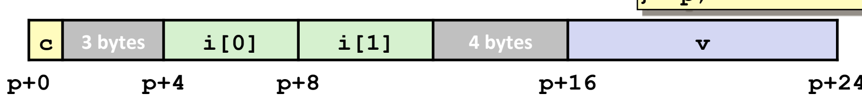


Different Alignment Conventions

x86-64 or IA32 Windows:

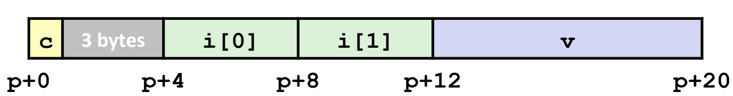
- K = 8, due to **double** element

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



IA32 Linux

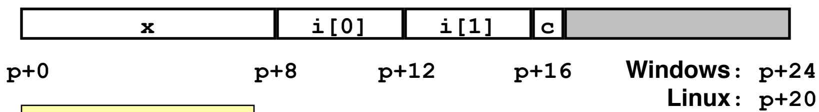
- K = 4; `double` treated like a 4-byte data type



Overall Alignment Requirement

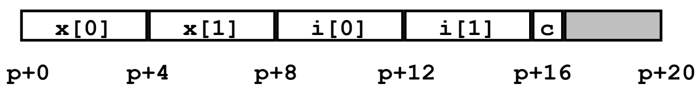
```
struct S2 {
  double x;
  int i[2];
  char c;
} *p;
```

p must be multiple of:
4 for Linux




```
struct S3 {
  float x[2];
  int i[2];
  char c;
} *p;
```

p must be multiple of 4

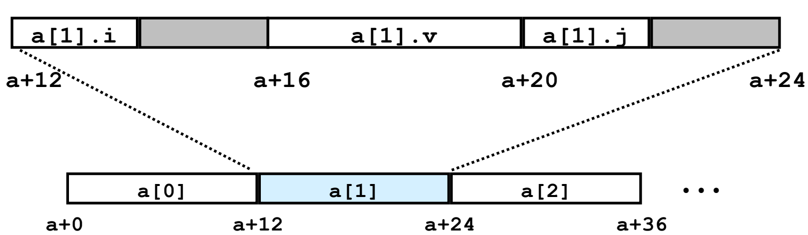


Arrays of Structures

Principle

- Allocated by repeating allocation for array type
- In general, may nest arrays & structures to arbitrary depth

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```



Accessing Element within Array

Compute offset to start of structure

- Compute 12*i as 4*(i+2i)

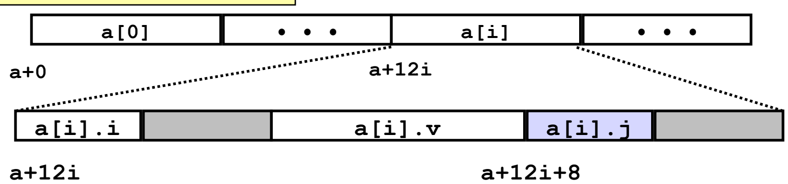
Access element according to its offset within structure

- Offset by 8
- Assembler gives displacement as a + 8
  - Linker must set actual value

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```

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

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



(Note: **movswl** loads a 16-bit value into a 32-bit register with sign-extension)

Satisfying Alignment within Structure

Starting address of structure array must be multiple of worst-case alignment for any element

- `a` must be multiple of 4

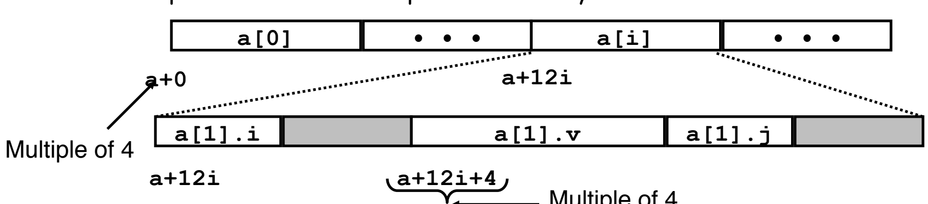
Offset of element within structure must be multiple of element's alignment requirement

- `v`'s offset of 4 is a multiple of 4

Overall size of structure must be multiple of worst-case alignment for any element

- Structure padded with unused space to be 12 bytes

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```



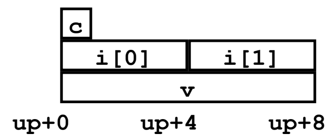
Union Allocation

Overlay union elements

Allocate according to largest element

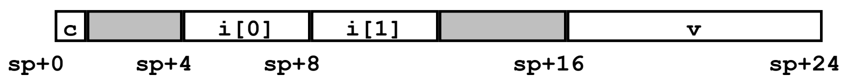
Can only use one field at a time

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



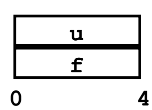

```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```

(Windows alignment)



Using Union to Access Bit Patterns

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




```
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;
}
```

Get direct access to bit representation of float

`bit2float` generates float with given bit pattern

- NOT the same as `(float) u`

`float2bit` generates bit pattern from float

- NOT the same as `(unsigned) f`

Byte Ordering Revisited

- Idea
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- Big Endian
  - Most significant byte has lowest address
  - Sparc
- Little Endian
  - Least significant byte has lowest address
  - Intel x86

Byte Ordering Example

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

32-bit

<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="4">l[0]</td><td colspan="4"></td></tr>
</table>

64-bit

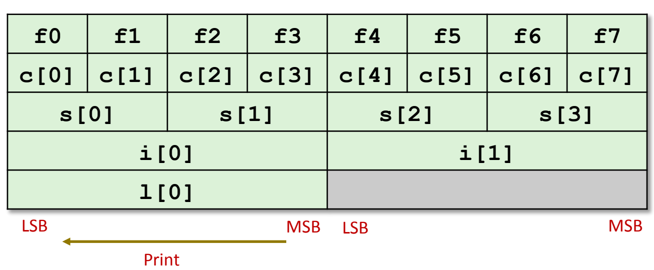
<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

Byte Ordering Example (Cont).

```
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]);
```

Byte Ordering on IA32

Little Endian



Output:

```
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   == [0xf3f2f1f0]
```

Byte Ordering on Sun

Big Endian

<table>
<tr><td>f0</td><td>f1</td><td>f2</td><td>f3</td><td>f4</td><td>f5</td><td>f6</td><td>f7</td></tr>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="4">l[0]</td><td colspan="4"></td></tr>
</table>

MSB LSB MSB LSB

Print

Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Byte Ordering on x86-64

Little Endian

<table>
<tr><td>f0</td><td>f1</td><td>f2</td><td>f3</td><td>f4</td><td>f5</td><td>f6</td><td>f7</td></tr>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="8">l[0]</td></tr>
</table>

LSB MSB

Print

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]
```

Summary

Arrays in C

- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

Compiler Optimizations

- Compiler often turns array code into pointer code
- Uses addressing modes to scale array indices
- Lots of tricks to improve array indexing in loops

Structures

- Allocate bytes in order declared
- Pad in middle and at end to satisfy alignment

Unions

- Overlay declarations
- Way to circumvent type system