Today

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Alignment
- Unions

Basic Data Types

- Integral
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used
    - Intel AT&T Bytes
      - 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 AT&T Bytes
      - Single s 4 float
      - Double d 8 double
    - IA32 x86-64

Array Allocation

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

Array Access

- Basic Principle
  - $T \text{[}A\text{]}$:
    - 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
  - $\text{val}[4]$ int 3
  - $\text{val}$ int * $x$
  - $\text{val} + \text{val}[4]$ int * $x + 4$

Array Example

- Declaration “zip_dig cmu” equivalent to “int cmu[5]”
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to be adjacent in general
Array Accessing Example

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

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

IA32

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

Array Loop Example (IA32)

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

Pointer Loop Example (IA32)

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

Exercise: Assembly Code Matching

```assembly
char *cp;  // Pointer
if (!*cp) …
incl (%eax)  // Increment
int i, ary[20];
return &ary[i];  // Return element
int *p;
clmp $0x0, (%edx)  // Compare
*p++;  // i++
short a2[10];
a2[2] += 2;  // Add
```

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

```
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 },
    {4, 5, 2, 1, 1 };
```

- **Nested Array Element Access**
  - 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**
  - Array Elements
    - pgh[index][dig] is int
    - Address: pgh + 4*(5*index + dig)
  - IA32 Code
    - Computes address pgh + 4*((index+4*index)+dig)

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

```
movl %ebp, %eax  # index
leal (%eax,%eax,4), %eax  # 5*index
```

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

```
leal (pgh,4,%eax), %eax  # (index+4*index)+dig
```

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

- **Multi-Level Array Example**
  - Variable univ denotes array of 3 elements
  - Each element is a pointer
    - A bytes
    - Each pointer points to array of int's

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

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

```
movl pgh(%eax,4), %eax  # offset 4*(5*index+dig)
```

```
movl %eax, %ebp  # index
leal (%eax,%eax,4), %eax  # 5*index+dig
```

```
addl 12(%ebp), %eax  # 5*index+dig
```

```
movl pgh(%eax,4), %eax  # pgh + 4*{(index+4*index)+dig}
```

- **Element Access in Multi-Level Array**
  - 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

Nested array

\[
\text{int get_pgh_digit}(\text{int index}, \text{int dig})
\]

\[
\text{return pgh[index][dig];}
\]

Multi-level array

\[
\text{int get_univ_digit}(\text{int index}, \text{int dig})
\]

\[
\text{return univ[index][dig];}
\]

Accesses use same syntax (different types) in C, but addresses very different:

\[
\text{Mem}[\text{pgh}+20*\text{index}+4*\text{dig}] \quad \text{Mem}[\text{Mem}[\text{univ}+4*\text{index}]+4*\text{dig}]
\]

16 X 16 Matrix Access

- **Array Elements**
  - Address A + j*\(C*K\) + j*K
  - \(C = 16, K = 4\)

\[
/* \text{Get element a}[i][j] */
\text{int fix_ele}(\text{fix_matrix} a, \text{int i}, \text{int j})
\]

\[
\text{return a}[i][j];
\]

\[
\text{movl} 12(\%ebp), \%edx \# i
\]

\[
\text{sall} \$6, \%edx \# i*64
\]

\[
\text{movl} 16(\%ebp), \%eax \# j
\]

\[
\text{sall} \$2, \%eax \# j*4
\]

\[
\text{addl} 8(\%ebp), \%eax \# a + j*4
\]

\[
\text{movl} (\%eax, \%edx), \%eax \# *(a + j*4 + i*64)
\]

16 X 16 Matrix Code

- **Fixed dimensions**
  - Know value of N at compile time

\[
/* \text{Get element a}[i][j] */
\text{int fix_ele}(\text{fix_matrix} a, \text{int i}, \text{int j})
\]

\[
\text{return a}[i][j];
\]

\[
\text{movl} 12(\%ebp), \%eax \# i
\]

\[
\text{sall} \$6, \%eax \# i*64
\]

\[
\text{movl} 16(\%ebp), \%edx \# j
\]

\[
\text{sall} \$2, \%edx \# j*4
\]

\[
\text{addl} 8(\%ebp), \%edx \# a + j*4
\]

\[
\text{movl} (\%eax, \%edx), \%eax \# *(a + j*4 + i*64)
\]

n X n Matrix Access

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

\[
/* \text{Get element a}[i][j] */
\text{int var_ele}(\text{int n}, \text{int a}[n][n], \text{int i}, \text{int j})
\]

\[
\text{return a}[i][j];
\]

\[
\text{movl} 8(\%ebp), \%eax \# n
\]

\[
\text{sall} \$2, \%eax \# n*4
\]

\[
\text{imull} 16(\%ebp), \%edx \# i*n*4
\]

\[
\text{movl} 20(\%ebp), \%eax \# j
\]

\[
\text{sall} \$2, \%eax \# j*4
\]

\[
\text{addl} 12(\%ebp), \%eax \# a + j*4
\]

\[
\text{movl} (\%eax, \%edx), \%eax \# *(a + j*4 + i*n*4)
\]

Optimizing Fixed Array Access

- **Computation**
  - Step through all elements in column

- **Optimization**
  - Retrieving successive elements from single column

Optimizing Fixed Array Access

- **Optimization**
  - Compute \(a = a[i][j]\)
    - Initially = \(a + 4\)
    - Increment by \(4*N\)

\[
/* \text{Retrieve column j from array} */
\text{void fix_column}(\text{fix_matrix} a, \text{int j}, \text{int *dest})
\]

\[
\text{return a}[i][j];
\]

\[
\text{movl} (\%ecx, \%eax), \%eax \# \text{ajp}
\]

\[
\text{addl} \$1, \%edx \# i++
\]

\[
\text{addl} \$64, \%eax \# \text{ajp} \text{= a} \times 4^4\text{N}
\]

\[
\text{cmp}l \$16, \%edx \# 16
\]

\[
\text{jne .L8} \# \text{if i != 16, goto loop}
\]

Optimizing Fixed Array Access

- **Optimization**
  - Compute \(a = a[i][j]\)
    - Initially = \(a + 4\)
    - Increment by \(4*N\)

\[
/* \text{Retrieve column j from array} */
\text{int fix_column}(\text{fix_matrix} a, \text{int j}, \text{int *dest})
\]

\[
\text{return a}[i][j];
\]
**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];
}
```

```
.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
cmp1 %edx, %esi  # n:i
jg .L18  # if >, goto loop
```

**Today**

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

**Structure Allocation**

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%ecx</td>
<td>ajp</td>
</tr>
<tr>
<td>%edi</td>
<td>dest</td>
</tr>
<tr>
<td>%edx</td>
<td>i</td>
</tr>
<tr>
<td>%ebx</td>
<td>4*n</td>
</tr>
<tr>
<td>%esi</td>
<td>n</td>
</tr>
</tbody>
</table>

```
.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
cmp1 %edx, %esi  # n:i
jg .L18  # if >, goto loop
```

**Structure Access**

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

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

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

**Generating Pointer to Structure Member**

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

```
int *get_ap (struct rec *r, int idx)
{
    return r+idx*4;
}
```

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

**Generating Pointer to Array Element**

- Offset of each structure member determined at compile time
- Arguments
  - Mem[\%ebp+8]: r
  - Mem[\%ebp+12]: idx

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

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

**Structure Access**

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

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

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

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

**Following Linked List**

```
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;
};
```

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>r</td>
</tr>
<tr>
<td>%ecx</td>
<td>val</td>
</tr>
</tbody>
</table>

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

**Structure Access**

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

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

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

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Unions

Structures & Alignment

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

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

Specific Cases of Alignment (IA32)

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

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 particularly tricky when datum spans 2 pages
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

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
Different Alignment Conventions

- x86-64 or IA32 Windows:
  - \( K = 8 \), due to double element

\[
\begin{array}{ccccc}
\text{c} & \text{i[0]} & \text{i[1]} & \text{v} & \\
p+0 & p+4 & p+8 & p+16 & p+24
\end{array}
\]

- IA32 Linux
  - \( K = 4 \); double treated like a 4-byte data type

\[
\begin{array}{ccccc}
\text{c} & \text{i[0]} & \text{i[1]} & \text{v} & \\
p+0 & p+4 & p+8 & p+12 & p+20
\end{array}
\]

Meeting Overall Alignment Requirement

- For largest alignment requirement \( K \)
- Overall structure must be multiple of \( K \)
- x86-64 again:

\[
\begin{array}{cccc}
\text{v} & \text{i[0]} & \text{i[1]} & \text{c} & \\
p+0 & p+4 & p+8 & p+16 & p+24
\end{array}
\]

Arrays of Structures

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

\[
\begin{array}{cccc}
\text{a[0]} & \text{a[1]} & \text{a[2]} & \\
a+0 & a+24 & a+48 & a+72
\end{array}
\]

Accessing Array Elements

- Compute array offset \( 12i \)
  - \( \text{sizeof}(\text{S3}) \), including alignment spacers
- Element \( j \) is at offset 8 within structure
- Assembler gives offset \( a+8 \)
  - Resolved (addition performed) during linking

Exercise Break: Structure Size/Alignment

- What is the size of each of these structs?
  - \text{struct S1} \{ char c1, c2; \}; \quad 2
  - \text{struct S2} \{ int i1, i2; \}; \quad 8
  - \text{struct S3} \{ char c; int i; \}; \quad 8
  - \text{struct S4} \{ int i; char c; \}; \quad 8
  - \text{struct S5} \{ char c; int i; char d; \}; \quad 12
  - \text{struct S6} \{ int i; char c; char d; \}; \quad 8

Saving Space

- Good basic rule: put larger data types first

\[
\begin{array}{cccc}
\text{c} & \text{i} & \text{d} & \\
2\text{bytes} & 4\text{bytes} & 4\text{bytes}
\end{array}
\]

Effect \( \text{K=4} \)
Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment
- **Unions**

Union Allocation

- Allocate according to largest element
- Can only use one field at a time

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

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

Using Union to Access Bit Patterns

```c
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;
}
```

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

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

```c
32-bit
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td>i[1]</td>
<td>i[2]</td>
<td>i[3]</td>
</tr>
</tbody>
</table>
```

```c
64-bit
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
</table>
```

Byte Ordering Example (Cont.)

```c
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;
```

```c
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]);
```

```c
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]);
```

```c
printf("Ints 0-1 == [0x%x,0x%x]n", 
    dw.i[0], dw.i[1]);
```

```c
printf("Long 0 == [0x%lx]n", 
    dw.l[0]);
```
Byte Ordering on IA32

**Little Endian**

<table>
<thead>
<tr>
<th>$0$</th>
<th>$1$</th>
<th>$2$</th>
<th>$3$</th>
<th>$4$</th>
<th>$5$</th>
<th>$6$</th>
<th>$7$</th>
</tr>
</thead>
</table>

**Output:**

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long 0 == [0xf3f2f1f0]

Byte Ordering on Sun

**Big Endian**

<table>
<thead>
<tr>
<th>$0$</th>
<th>$1$</th>
<th>$2$</th>
<th>$3$</th>
<th>$4$</th>
<th>$5$</th>
<th>$6$</th>
<th>$7$</th>
</tr>
</thead>
</table>

**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>
<thead>
<tr>
<th>$0$</th>
<th>$1$</th>
<th>$2$</th>
<th>$3$</th>
<th>$4$</th>
<th>$5$</th>
<th>$6$</th>
<th>$7$</th>
</tr>
</thead>
</table>

**Output on x86-64:**

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints 0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long 0 == [0xf7f6f5f4f3f2f1f0]

Summary

- **Arrays in C**
  - Contiguous allocation of memory
  - Aligned to satisfy every element’s alignment requirement
  - Pointer to first element
  - No bounds checking
- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- **Unions**
  - Overlay declarations to save space
  - Reveals underlying representation (circumvents type system)