Announcements

• Assignment 2 (Binary bomb) due Thursday

• We are trying out Piazza to allow class-wide questions and discussions
  • Go to http://piazza.com/harvard/fall2011/cs61 to sign up and join in

• Google form to help you form a study group or find a partner
  • See course web page for links
  • Welcome to discuss problems with your classmates, and to collaborate in planning and thinking through solutions.
  • For Assignments 1–3, graded individually
  • For Assignments 4–6, allowed to work in pairs; pair shares grade
  • More info on Course Policies page
Topics for today

• Structured data
  • Arrays
  • Multidimensional arrays
  • Multi-level arrays
  • Structs
  • Memory alignment
  • Arrays of structs
Basic data types

• Integer data types
  • Stored & operated on in general (integer) registers
  • Signed vs. unsigned depends on instructions used

• Floating Point
  • Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>C declaration</th>
<th>Intel data type</th>
<th>Assembly code suffix</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Byte</td>
<td>b</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>Word</td>
<td>w</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>Double word</td>
<td>l</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long int</td>
<td>Quad word</td>
<td>q</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>Single precision</td>
<td>s</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>Double precision</td>
<td>d</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>Extended precision</td>
<td>t</td>
<td>10/12</td>
<td>10/16</td>
</tr>
</tbody>
</table>
Array allocation

• \( T \ A[L]; \)
  • Declares array of data type \( T \) and length \( L \)
  • e.g., \( \text{int} \ \text{foo}[5] \)
• Represented as contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes

```
int foo[5];
foo[0] = 0;
foo[1] = 2;
foo[2] = 1;
foo[3] = 3;
foo[4] = 8;
```

```plaintext
0 2 1 3 8
```

\( x \quad x+4 \quad x+8 \quad x+12 \quad x+16 \quad x+20 \)
Array access

• Given \( T \) \( A[L] \);
  • \( A[i] \) accesses the \( i \)th element of the array
  • Identifier \( A \) can be used as a pointer to array element 0
    • Type of \( A \) is \( T * \)
    • E.g., \( \text{int foo[5]}; \)
      \( *\text{foo} = 4; \) is equivalent to \( \text{foo[0]} = 4; \)
int foo[5];

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo[2]</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>foo</td>
<td>int *</td>
<td>x</td>
</tr>
<tr>
<td>foo+1</td>
<td>int *</td>
<td>x+4</td>
</tr>
<tr>
<td>&amp;foo[2]</td>
<td>int *</td>
<td>x+8</td>
</tr>
<tr>
<td>foo[5]</td>
<td>int</td>
<td>???</td>
</tr>
<tr>
<td>*(foo+1)</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>foo + 3</td>
<td>int *</td>
<td>x+4   ×3</td>
</tr>
</tbody>
</table>
### Array notation

```c
int foo[5];
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo[2]</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td>foo</td>
<td>int *</td>
<td>x</td>
</tr>
<tr>
<td>foo+1</td>
<td>int *</td>
<td>x+4</td>
</tr>
<tr>
<td>&amp;foo[2]</td>
<td>int *</td>
<td>x+8</td>
</tr>
<tr>
<td>foo[5]</td>
<td>int</td>
<td>???</td>
</tr>
<tr>
<td>*(foo+1)</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>foo + 3</td>
<td>int *</td>
<td>x + 4×3</td>
</tr>
</tbody>
</table>

---

Stephen Chong, Harvard University
Array example

typedef int zip_dig[5];

zip_dig hvd = { 0, 2, 1, 3, 8 };
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig cor = { 1, 4, 8, 5, 3 };

• Note: declaration `zip_dig hvd` equivalent to `int hvd[5]`

• Example arrays were allocated in successive 20 byte blocks
  • Not guaranteed to be true!
### Accessing arrays

**%edx** contains starting address of array

**%eax** contains the array index

Desired value at **%edx + (%eax * 4)**

```c
typedef int zip_dig[5];

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

```assembly
push   %ebp
mov    %esp,%ebp
mov    0xc(%ebp),%eax     # %eax = dig
mov    0x8(%ebp),%edx     # %edx = z
mov    (%edx,%eax,4),%eax # %eax = z[dig]
pop    %ebp
ret
```
Referencing examples

zip_dig hvd;

zip_dig cmu;

zip_dig cor;

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed to work?</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmu[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmu[5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmu[-1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hvd[15]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Referencing examples

```plaintext
zip_dig hvd;

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>28</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
</tr>
</tbody>
</table>

zip_dig cmu;

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>44</td>
</tr>
<tr>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>3</td>
<td>52</td>
</tr>
</tbody>
</table>

zip_dig cor;

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>68</td>
</tr>
<tr>
<td>3</td>
<td>72</td>
</tr>
</tbody>
</table>

Expression | Address | Value | Guaranteed to work? |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cmu[3]</td>
<td>36 + 4*3 = 48</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>cmu[5]</td>
<td>36 + 4*5 = 56</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>cmu[-1]</td>
<td>36 + 4*-1 = 32</td>
<td>8</td>
<td>No</td>
</tr>
<tr>
<td>hv[15]</td>
<td>16+4*15 = 76</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>
```
Looping over an array

- Original source

```c
int zd2int(zip_dig z) {
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

- What gcc produces
  - (pseudocode)
  - Eliminate loop variable `i`
  - Convert array code to pointer code
  - Rewrite as do-while loop

```c
int zd2int(zip_dig z) {
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z <= zend);
    return zi;
}
```
Looping over array in machine code

int zd2int(zip_digit z) {
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z <= zend);
    return zi;
}

• Notes:
  • `xorl %eax,%eax` sets %eax to zero
  • `leal` used for arithmetic
  • 4 added to %ecx each iteration

...                      # % ecx = z
xorl %eax,%eax           # zi = 0
leal 16(%ecx),%ebx       # zend = z+4
.L59:
  leal (%eax,%eax,4),%edx # edx = zi + 4*zi = 5*zi
  movl (%ecx),%eax        # zi = *z
  addl $4,%ecx            # z++
  leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
  cmpl %ebx,%ecx          # compare
  jle .L59                # if z <= zend, loop
• x86 doesn’t have an internal notion of a string datatype.

• In C, a string is represented as an array of char, terminated by a 0 byte
  • Each character is one ASCII encoded byte
  • E.g., “this is a string”

  \[
  \begin{array}{cccccccccccccccccc}
    0x74 & 0x68 & 0x69 & 0x73 & 0x20 & 0x69 & 0x73 & 0x20 & 0x61 & 0x20 & 0x73 & 0x74 & 0x69 & 0x6e & 0x67 & 0x00 \\
  \end{array}
  \]

  0x74 is ASCII encoding of ‘t’
  0x0 terminates string

• Different languages use different string representations
  • E.g., Java stores strings in Unicode format (1-4 bytes per character)
Topics for today

• Structured data
  • Arrays
  • Multidimensional arrays
  • Multi-level arrays
  • Structs
  • Memory alignment
  • Arrays of structs
Multidimensional arrays

- $T \ A[R][C]$;
  - Two dimensional array of datatype $T$
  - $R$ rows and $C$ columns
- Represented as contiguously allocated region of $R \times C \times \text{sizeof}(T)$ bytes
  - Row major ordering: rows stored one after the other

```c
int cambridge_zips[4][5] = 
{ {0, 2, 1, 3, 8},
{0, 2, 1, 3, 9 },
{0, 2, 1, 4, 0 },
{0, 2, 1, 4, 1 };
```

![Diagram showing the allocation and ordering of elements in a 2D array]

$\begin{array}{cccccc}
0 & 2 & 1 & 3 & 8 & 0 \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
x & x+20 & x+40 & x+60 & x+80
\end{array}$
Multidimensional arrays

- typedef int zip_dig[5];
- zip_dig cambridge_zips[4]
is equivalent to
  int cambridge_zips[4][5]
- zip_dig cambridge_zips[4]: array of 4 elements, allocated continuously
- Each element is an array of 5 ints, allocated continuously

```
int cambridge_zips[4][5] =
{{0, 2, 1, 3, 8},
 {0, 2, 1, 3, 9 },
 {0, 2, 1, 4, 0 },
 {0, 2, 1, 4, 1 }};
```
Accessing a single array element

- $T A[R][C]$;
  - $A[i][j]$ is a single element of type $T$
  - Address is $A + i \times C \times \text{sizeof}(T) + j \times \text{sizeof}(T)$

- E.g., `int foo[3][4];`

- `&foo[2][1]` evaluates to
  \[
  x + 2 \times 4 \times \text{sizeof(int)} + 1 \times \text{sizeof(int)} = x + 32 + 4 = x + 36
  \]
int cambridge_zips[4][5] = { ... };

int get_zip_digit(int index, int digit) {
    return cambridge_zips[index][digit];
}

...  # %ecx = digit
    # %eax = index
    leal 0(%ecx,4),%edx                      # %edx = 4*digit
    leal (%eax,%eax,4),%eax                  # %eax = 5*index
    movl cambridge_zips(%edx,%eax,4),%eax    # return contents of
    # cambridge_zips + 4*digit
    #    + 4*5*index
    ...
Strange referencing examples

```c
int zips[4][5];
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed to work?</th>
</tr>
</thead>
<tbody>
<tr>
<td>zips[3][3]</td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>zips[2][5]</td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>zips[2][-1]</td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>zips[4][-1]</td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>zips[0][19]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zips[0][-1]</td>
<td></td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>
```c
int zips[4][5];
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed to work?</th>
</tr>
</thead>
<tbody>
<tr>
<td>zips[3][3]</td>
<td>$x + 20<em>3 + 4</em>3 = x+72$</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>zips[2][5]</td>
<td>$x + 20<em>2 + 4</em>5 = x+60$</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>zips[2][-1]</td>
<td>$x + 20<em>2 + 4</em>-1 = x+36$</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>zips[4][-1]</td>
<td>$x + 20<em>4 + 4</em>-1 = x+76$</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>zips[0][19]</td>
<td>$x + 20<em>0 + 4</em>19 = x+76$</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>zips[0][-1]</td>
<td>$x + 0<em>2 + 4</em>-1 = x-4$</td>
<td>???</td>
<td>No</td>
</tr>
</tbody>
</table>
Topics for today

• Structured data
  • Arrays
  • Multidimensional arrays
  • Multi-level arrays
  • Structs
  • Memory alignment
  • Arrays of structs
typedef int zip_dig[5];
zip_dig hvd = { 0, 2, 1, 3, 8 };  
zip_dig cmu = { 1, 5, 2, 1, 3 };  
zip_dig cor = { 1, 4, 8, 5, 3 };  
int *univ[3] = {cmu, hvd, cor};

- `univ` is array of 3 elements
- Each element is a pointer (4 bytes)
- Each pointer points to an array of ints
Multilevel array element access

- Access looks similar, but evaluation is quite different!
Multilevel array element access

```c
int cambridge_zips[4][5] = {...};

int get_zip_digit(int index, int digit) {
    return cambridge_zips[index][digit];
}
```

One memory read

```asm
# %ecx = digit
# %eax = index
leal 0(%ecx,4),%edx       # %edx = 4*digit
leal (%eax,%eax,4),%eax    # %eax = 5*index
movl cambridge_zips(%edx,%eax,4),%eax
    # return Mem[cambridge_zips + 20*index + 4*dig]
```

```c
int *univ[3] = {cmu, hvd, cor};

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

Two memory reads
- First get pointer to array
- Then access array

```asm
# %ecx = index
# %eax = dig
leal 0(%ecx,4),%edx       # %edx = 4*index
movl univ(%edx),%edx     # %edx = Mem[univ + 4*index]
movl (%edx,%eax,4),%eax # Mem[%edx + 4*dig]
```

Two memory reads
- First get pointer to array
- Then access array
## Strange referencing examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed to work?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>univ[2][3]</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>univ[1][5]</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>univ[2][-1]</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>univ[3][-1]</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>univ[1][12]</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Strange referencing examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed to work?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>univ[2][3]</code></td>
<td>56 + 4*3 = 68</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td><code>univ[1][5]</code></td>
<td>16 + 4*5 = 36</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[2][-1]</code></td>
<td>56 + 4*-1 = 52</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[3][-1]</code></td>
<td>???</td>
<td>???</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[1][12]</code></td>
<td>16 + 4*12 = 64</td>
<td>8</td>
<td>No</td>
</tr>
</tbody>
</table>
Topics for today

• Structured data
  • Arrays
  • Multidimensional arrays
  • Multi-level arrays
• Structs
• Memory alignment
• Arrays of structs
Struts

- A struct groups objects into a single object
- Each field accessed by name
- Each field can have a different type

```c
struct rect {
    int llx;
    int lly;
    int width;
    int height;
    char *name;
};

struct rect r;
r.llx = r.lly = 0;

struct rect s =
{0, 0, 10, 20, "Rodney"};

struct rect *rp = &s;
rp->height = rp->width;
(*rp).height = (*rp).width;
```
Implementing structs

• Struct represented as contiguous region of memory

• E.g.,

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

• Compiler generates code using appropriate offsets

```c
demo_struct:
pushl %ebp
    movl %esp, %ebp
    movl myrec, %eax
    movl $42, (%eax)
    movl $43, 4(%eax)
    movl $someint, 16(%eax)
popl %ebp
    ret
```
Memory alignment

• Many systems require data to be aligned in memory
  • E.g., Linux/x86 requires integers to be stored at memory addresses that is multiple of 4 bytes

• Why?
  • 32-bit machines typically read and write 4 bytes of memory at a time from word aligned addresses
  • If not aligned, reading an int from memory may require two memory accesses!
Memory alignment

- Some processors require aligned access
  - If you try to access an “int” not stored on an aligned address, would get an “alignment fault”

- Intel x86 does not require memory access to be aligned
  - However, Intel strongly recommends that compilers generate code that uses aligned access, to get the best performance.

- Different OS's and compilers have different rules
  - Linux: 2-byte types aligned to 2 byte boundaries, 4 byte and larger types aligned to 4 byte boundaries
  - Windows: primitive object of $K$ bytes must have an address that is a multiple of $K$
Alignment within structs

• Each field in struct must be aligned correctly
  • Compiler may have to insert **padding** within struct

• E.g., using Windows alignment rules
  • Field `i` must be aligned on 4 byte address
  • Field `v` must be aligned on 8 byte address

```c
struct S {
    char c;
    int i[2];
    double v;
};
```

• `sizeof(struct S)` is 24, not `1+4+4+8 = 17`
Save memory!

```c
struct T1 {
    char c;
    double v;
    char d;
    int i;
};
```

10 bytes wasted space (Windows alignment)

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>v</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x+1</td>
<td>x+8</td>
<td>x+16</td>
<td>x+20</td>
</tr>
</tbody>
</table>
```

```c
struct T2 {
    double v;
    char c;
    char d;
    int i;
};
```

2 bytes wasted space (Windows alignment)

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>v</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x+8</td>
<td>x+12</td>
<td>x+16</td>
<td></td>
</tr>
</tbody>
</table>
```
Alignment of whole structs

E.g.,

```c
struct U {
    float x[2];
    int i[2];
    char c;
};
```

- No padding needed within struct

Misaligned!
Padding at end of struct

- To ensure that arrays of structs are correctly aligned, may need to pad at end of struct

- E.g.,
  ```c
  struct U {
      float x[2];
      int i[2];
      char c;
  };
  ```
  `sizeof(U)` must be multiple of 4 (Windows and Linux)

- E.g.,
  ```c
  struct V {
      double x;
      int i[2];
      char c;
  };
  ```
  `sizeof(V)` must be multiple of 8 for Windows, 4 for Linux
Next lecture

• Buffer overruns and stack exploits
  • B3 l33t H4x0r and pwnzorz the interwebz
  • (Use your powers wisely)