Some of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer’s Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in Fall 2010. These slides are indicated “Supplied by CMU” in the notes section of the slides.
Meet Your TAs!!

- Come to CIT 3rd-floor atrium on Thursday at 5
- Eat tasty cookies!
- Talk to the course staff!
- Talk to fellow students in the course!
void func(int arg[]) {
    /* arg points to the caller’s array */
    int local[7];    /* seven ints */
    arg++;           /* legal */
    arg = local;     /* legal */
    local++;         /* illegal */
    local = arg;     /* illegal */
}

Arrays and Parameters
Operator precedence is hard to remember!
Dereferencing C Pointers

```c
int main() {
    int *p; int a = 4;
    p = &a;
    (*p)++;
    printf("%d %u\n", *p, p);
}
```

```
$ ./a.out
5 3221224356
```
Dereferencing C Pointers

```c
int main() {
    int *p; int a = 4;
    p = &a;
    ++*p;
    printf("%d %u\n", *p, p);
}
```

```
$ ./a.out
5 3221224356
```
Quiz 1

```c
int proc(int arg[]) {
    arg++;
    return arg[1];
}

int main() {
    int A[3] = {0, 1, 2};
    printf("%d\n", proc(A));
}
```

What's printed?

a) 0  
b) 1  
c) 2  
d) indeterminate
Strings

- Strings are arrays of characters terminated by `\0` ("null")
  - the `\0` is included at the end of string constants
    » "Hello"

Note that `\0` is represented as a byte containing all zeroes.
Since we didn’t explicitly output a newline character, the prompt for the next command goes on the same line as the string that was printed.
We’ve added the newline character to the format specifier of printf – the prompt now appears on the next line.
We can also print a single character at a time. Note the test for the null character to determine whether we’ve reached the end of the string.
Note that even though we might think of “int [6]” as being a datatype, to declare “n” to be of that type, we must write “int n[6]” — the size of the array goes just after the identifier.
3-D Arrays

- How do we declare an array of eight $T[7][6]$?

  $T \ p[8][7][6]$
  - $p$ is an array of (eight) $T[7][6]$
  - $p[i]$ is of type $T[7][6]$
  - $p[i][j]$ is of type $T[6]$
  - $p[i][j][k]$ is of type $T$
Here we initialize a 2D array, then call a function (described in the next slide) to print it.
We print the array by rows.
C arrays are stored in *row-major order*, as shown in the slide. The idea is that the left index references the row, the right index references the column. Thus C arrays are stored row-by-row. Thus to index into a 2D array, we need to know how large each row is (i.e., how many columns there are). But it’s not necessary, for indexing purposes, to know how many rows there are.
In general we don’t need to specify the size of the leftmost dimension of an array argument. In the current 2D example, what’s important is that the compiler know the size of each row so that it can generate code to compute where a particular element is.
Note that \textit{m} is an array of arrays (in particular, an array of 1-D arrays).
While it's convenient to think of something as being a 2D array, its elements are stored linearly in memory. Thus, as shown in the slide where we are calling `AccessAs1D` to get the value of `A2D[1][2]`, given a pointer to a 2D array, we can access its elements as if it were a 1D array.
Quiz 2

Consider the array

```c
int A[3][3];
```

- which element is adjacent to `A[0][0]` in memory?
  
a) `A[0][1]`
b) `A[1][0]`
c) none of the above
Quiz 3

Consider the array

```c
int A[3][3];
int *B = &A[0][0];

B[8] = 8;
```

- which element of A was modified?
  a) A[0][3]
  b) A[2][2]
  c) A[3][0]
  d) none of the above
Number Representation

- Hindu-Arabic numerals
  - developed by Hindus starting in 5th century
    » positional notation
    » symbol for 0
  - adopted and modified somewhat later by Arabs
    » known by them as “Rakam Al-Hind” (Hindu numeral system)
  - 1999 rather than MCMXCIX
    » (try doing long division with Roman numerals!)
Base 2 is known as “binary” notation.
Base 8 is known as “octal” notation.
Base 10 is known as “decimal” notation.
Base 16 is known as “hexadecimal” notation. Note that “hexa” is derived from the Greek language and “decimal” is derived from the Latin language. Many people feel you shouldn’t mix languages when you invent words, but IBM, who coined the term “hexadecimal” in the 1960s, didn’t think their corporate image could withstand “sexadecimal”.

\[
\begin{align*}
1999 &= 9 \cdot 10^2 + 9 \cdot 10^1 + 9 \cdot 10^0 + 1 \cdot 10^3 \\
2 &= 11111001111_2 \\
&= 1 \cdot 2^9 + 2 \cdot 2^8 + 1 \cdot 2^7 + 2^6 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
8 &= 3717_8 \\
&= 7 \cdot 8^2 + 8 \cdot 8^1 + 3 \cdot 8^0 \\
&= \text{why are we interested?} \\
16 &= 7CF_{16} \\
&= 1 \cdot 16^2 + 12 \cdot 16^1 + 7 \cdot 16^0 \\
&= \text{why are we interested?}
\end{align*}
\]
Note that a byte consists of two hexadecimal digits, which are sometimes known as “nibbles”. A 32-bit computer word would then have eight nibbles; a 64-bit computer word would have sixteen nibbles.
This routine prints the base base representation of num. The “%” operator yields the remainder. E.g., “10%3” evaluates to 1: the remainder after dividing 10 by 3. (Note that the “...” is not heretofore unexplained C syntax, but is shorthand for “fill this in to the extent needed.”)
“bc” (it stands for basic calculator, or perhaps better calculator) is a standard Unix command that handles arbitrary-precision arithmetic. Among its features is the ability to specify which base to use for input and output of numbers. The default base for both input and output is ten. Setting `obase` to 16 sets the base for output to 16. Similarly, one can change the base for input numbers by setting `ibase`. 

```
$ bc
obase=16
1999
7CF
$
```
Quiz 4

- What’s the decimal (base 10) equivalent of $23_{16}$?
  a) 19
  b) 33
  c) 35
  d) 37
**Encoding Byte Values**

- **Byte = 8 bits**
  - binary 00000000₂ to 1111111₁₂
  - decimal: 0₁₀ to 255₁₀
  - hexadecimal 0₀₁₆ to FF₁₆
    - base 16 number representation
    - use characters '0' to '9' and 'A' to 'F'
    - write FA1D37B₁₆ in C as
      - 0xFA1D37B
      - 0xfa1d37b

<table>
<thead>
<tr>
<th>Hex</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>13</td>
<td>1101</td>
</tr>
<tr>
<td>E</td>
<td>14</td>
<td>1110</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

Supplied by CMU.
If a computer word is to be interpreted as an unsigned integer, we can do so as shown in the slide for 32-bit integers. Thus integers are represented in binary (base-2) notation in the computer. We'll discuss representing negative integers in an upcoming lecture.
Storing and Viewing Ints

```c
int main() {
    int n = 57;
    printf("binary: %b, decimal: %d, 
    "hex: %x\n", n, n, n);
    return 0;
}
```

$ ./a.out
binary: 111001, decimal: 57, hex: 39
$

Here n is an int whose value is 57 (expressed in base 10). As we've seen, it's represented in the computer in binary. When we print its value using printf, we choose to view it in the base specified by the format code. %b means binary, %d means decimal, and %x means hexadecimal.

Note, in the arguments for printf, that the format string is in two parts. C allows you to do this: "string 1 " "string 2" is treated the same as "string1 string2".
**Boolean Algebra**

- Developed by George Boole in 19th Century
  - algebraic representation of logic
    » encode “true” as 1 and “false” as 0

<table>
<thead>
<tr>
<th>And</th>
<th>Or</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A &amp; B = 1 ) when both ( A=1 ) and ( B=1 )</td>
<td>( A</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 1 )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

**Not**

- \( \sim A = 1 \) when \( A=0 \)
- \( A^\land B = 1 \) when either \( A=1 \) or \( B=1 \), but not both

<table>
<thead>
<tr>
<th>~</th>
<th>( \wedge )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 0 )</td>
<td>( 1 )</td>
</tr>
</tbody>
</table>
General Boolean Algebras

- **Operate on bit vectors**
  - operations applied bitwise
    
    \[
    \begin{array}{cccc}
    01101001 & 01101001 & 01101001 \\
    \& 01010101 & \mid 01010101 & ^ 01010101 & \sim 01010101 \\
    01000001 & 01111101 & 00111100 & 10101010
    \end{array}
    \]

- **All of the properties of boolean algebra apply**
Example: Representing & Manipulating Sets

- **Representation**
  - width-w bit vector represents subsets of \{0, ..., w-1\}
  - \( a_j = 1 \) iff \( j \in A \)

\[
\begin{align*}
01101001 & \quad \{ 0, 3, 5, 6 \} \\
76543210 & \\
01010101 & \quad \{ 0, 2, 4, 6 \} \\
76543210 & 
\end{align*}
\]

- **Operations**
  - \& intersection \quad 01000001 \quad \{ 0, 6 \}
  - | union \quad 01111101 \quad \{ 0, 2, 3, 4, 5, 6 \}
  - ^ symmetric difference \quad 00111100 \quad \{ 2, 3, 4, 5 \}
  - ~ complement \quad 10101010 \quad \{ 1, 3, 5, 7 \}

Supplied by CMU.
Bit-Level Operations in C

- Operations &, |, ~, ^ available in C
  - apply to any "integral" data type
    - long, int, short, char
  - view arguments as bit vectors
  - arguments applied bit-wise

- Examples (char datatype)
  \(~0x41 \rightarrow 0xBE\)
  \(~01000001_2 \rightarrow 10111110_2\)
  \(~0x00 \rightarrow 0xFF\)
  \(~00000000_2 \rightarrow 11111111_2\)
  \(0x69 \& 0x55 \rightarrow 0x41\)
  \(01101001_2 \& 01010101_2 \rightarrow 01000012\)
  \(0x69 \mid 0x55 \rightarrow 0x7D\)
  \(01101001_2 \mid 01010101_2 \rightarrow 01111101_2\)
Contrast: Logic Operations in C

• Contrast to Logical Operators
  – &&, ||, !
    » view 0 as “false”
    » anything nonzero as “true”
    » always return 0 or 1
    » early termination/short-circuited execution

• Examples (char datatype)
  !0x41 → 0x00
  !0x00 → 0x01
  !!0x41 → 0x01
  0x69 && 0x55 → 0x01
  0x69 || 0x55 → 0x01
  p && *p (avoids null pointer access)
Contrast: Logic Operations in C

- Contrast to Logical Operators
  - `&&`, `||`, `!`
  - Viewed as "false"

- Watch out for `&&` vs. `&` (and `||` vs. `|`)...
  - One of the more common oopsies in C programming

- Examples:
  - `0x01 -> 0x00`
  - `!0x00 -> 0x01`
  - `!!0x41 -> 0x01`

  - `0x69 && 0x55 -> 0x01`
  - `0x69 || 0x55 -> 0x01`
  - `p & *p` (avoids null pointer access)
Recall that a char is an 8-bit integer.
**Shift Operations**

- **Left Shift:** \( x \ll y \)
  - shift bit-vector \( x \) left \( y \) positions
  - throw away extra bits on left
  - fill with \( 0 \)'s on right

- **Right Shift:** \( x \gg y \)
  - shift bit-vector \( x \) right \( y \) positions
  - throw away extra bits on right
  - logical shift
    - fill with \( 0 \)'s on left
  - arithmetic shift
    - replicate most significant bit on left

- **Undefined Behavior**
  - shift amount < 0 or ≥ word size

---

<table>
<thead>
<tr>
<th>Argument</th>
<th>01100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( \gg 2 )</td>
<td>00011000</td>
</tr>
<tr>
<td>Arith. ( \gg 2 )</td>
<td>00011000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument</th>
<th>10100010</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ll 3 )</td>
<td>00010000</td>
</tr>
<tr>
<td>Log. ( \gg 2 )</td>
<td>00101000</td>
</tr>
<tr>
<td>Arith. ( \gg 2 )</td>
<td>11101000</td>
</tr>
</tbody>
</table>

Supplied by CMU.

Why we need both logical and arithmetic shifts should be clear by the end of an upcoming lecture. If one is applying a right shift to an `int`, it will be an arithmetic right shift. Why this is so will be explained in the upcoming lecture (it has to do with the representation of negative numbers).