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!
Arrays and Parameters

```c
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 */
}
```
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"

  Hello \0

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.
2-D Arrays

- Suppose T is a datatype (such as int)
- T n[6]
  - declares n to be an array of (six) T
  - the type of n is T[6]
- Thus T[6] is effectively a datatype
- Thus we can have an array of T[6]
- T m[7][6]
  - m is an array of (seven) T[6]
  - m[i] is of type T[6]
  - m[i][j] is of type T

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 \texttt{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”.

### Which Base?

- **1999**
  - base 10
    » $9\cdot10^2+9\cdot10^1+9\cdot10^0+1\cdot10^3$
  - base 2
    » $1111001111$
    » $1\cdot2^9+1\cdot2^8+1\cdot2^7+1\cdot2^6+0\cdot2^5+0\cdot2^4+1\cdot2^3+1\cdot2^2+1\cdot2^1+1\cdot2^0+1\cdot2^0$
  - base 8
    » $3717$
    » $7\cdot8^0+1\cdot8^1+7\cdot8^2+3\cdot8^3$
    » why are we interested?
  - base 16
    » $7CF$
    » $15\cdot16^0+12\cdot16^1+7\cdot16^2$
    » why are we interested?
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`.
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 00000000\textsubscript{2} to 11111111\textsubscript{2}
  - decimal: 0\textsubscript{10} to 255\textsubscript{10}
  - hexadecimal 00\textsubscript{16} to FF\textsubscript{16}
    - base 16 number representation
    - use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - write FA1D37B\textsubscript{16} 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.
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>&amp;</td>
<td>0 1</td>
</tr>
<tr>
<td></td>
<td>0 0 0</td>
</tr>
<tr>
<td></td>
<td>1 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Not</th>
<th>Exclusive-Or (Xor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~A = 1 when A=0</td>
<td>A^B = 1 when either A=1 or B=1, but not both</td>
</tr>
<tr>
<td>~</td>
<td>0 1</td>
</tr>
<tr>
<td></td>
<td>0 1</td>
</tr>
<tr>
<td></td>
<td>1 0</td>
</tr>
</tbody>
</table>

CS33 Intro to Computer Systems  III–31

Supplied by CMU.
General Boolean Algebras

• Operate on bit vectors
  – operations applied bitwise

<table>
<thead>
<tr>
<th>01101001 &amp; 01010101</th>
<th>01010101 ^ 01010101 ~ 01010101</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101001</td>
<td>01010101</td>
</tr>
<tr>
<td>01101001</td>
<td>01010101</td>
</tr>
<tr>
<td>01101001</td>
<td>01010101</td>
</tr>
<tr>
<td>01101001</td>
<td>01010101</td>
</tr>
<tr>
<td>01000001</td>
<td>01111101</td>
</tr>
<tr>
<td>01111101</td>
<td>00111100</td>
</tr>
<tr>
<td>01111101</td>
<td>10101010</td>
</tr>
</tbody>
</table>

• All of the properties of boolean algebra apply

Supplied by CMU.
### Example: Representing & Manipulating Sets

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

<table>
<thead>
<tr>
<th>Bit Vector</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101001</td>
<td>{0, 3, 5, 6}</td>
</tr>
<tr>
<td>10101001</td>
<td>{0, 2, 4, 6}</td>
</tr>
</tbody>
</table>

- **Operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Bit Vector</th>
<th>Subset</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; intersection</td>
<td>01000001</td>
<td>{0, 6}</td>
</tr>
<tr>
<td></td>
<td>union</td>
<td>01111101</td>
</tr>
<tr>
<td>^ symmetric difference</td>
<td>00111100</td>
<td>{2, 3, 4, 5}</td>
</tr>
<tr>
<td>~ complement</td>
<td>10101010</td>
<td>{1, 3, 5, 7}</td>
</tr>
</tbody>
</table>
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)
  \[
  \begin{align*}
  \sim 0x41 & \rightarrow 0xBE \\
  \sim 01000001_2 & \rightarrow 1011110_2 \\
  \sim 0x00 & \rightarrow 0xFF \\
  \sim 00000000_2 & \rightarrow 11111111_2 \\
  0x69 \ & \& \ 0x55 \rightarrow 0x41 \\
  01101001_2 \ \& \ 01010101_2 \ & \rightarrow \ 01000010_2 \\
  0x69 \ | \ 0x55 & \rightarrow 0x7D \\
  01101001_2 \ | \ 01010101_2 & \rightarrow 01111110_2
  \end{align*}
\]
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)
  1. !0x41 → 0x00
  2. !0x00 → 0x01
  3. ||0x41 → 0x01

\[\begin{align*}
0x69 &\land 0x55 &\rightarrow 0x01 \\
0x69 &\lor 0x55 &\rightarrow 0x01 \\
p &\land *p &\text{(avoids null pointer access)}
\end{align*}\]
Contrast: Logic Operations in C

- Contrast to Logical Operators
  - &&, ||, !
  - && vs. & (and || vs. |)
  - One of the more common oopsies in C programming

Watch out for && vs. & (and || vs. |)

\[
\begin{align*}
0 \times 41 & \rightarrow 0 \times 00 \\
!0 \times 00 & \rightarrow 0 \times 01 \\
!10 \times 41 & \rightarrow 0 \times 01 \\
0 \times 69 \ & \& \ 0 \times 55 & \rightarrow 0 \times 01 \\
0 \times 69 \ | | \ 0 \times 55 & \rightarrow 0 \times 01 \\
p \ & \& \ *p & \ (avoids \ null \ pointer \ access)
\end{align*}
\]
Recall that a char is an 8-bit integer.

Quiz 5

- Which of the following would determine whether the next-to-the-rightmost bit of Y (declared as a char) is 1? (i.e., the expression evaluates to true if and only if that bit of Y is 1.)
  a) Y & 0x02
  b) !(~Y) & 0x02
  c) both of the above
  d) none of the above
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).
Global Variables

```c
#define NUM_ROWS 3
#define NUM_COLS 4
int m[NUM_ROWS][NUM_COLS];

int main() {
    int row, col;
    for(row=0; row<NUM_ROWS; row++)
        for(col=0; col<NUM_COLS; col++)
            m[row][col] = row*NUM_COLS+col;
    return 0;
}
```

- The scope is global; $m$ can be used by all functions
Note that the reference to “m” gives the address of the array in memory.

The point of the slide is that global variables are in a different area of memory than are local variables.
If you don’t explicitly initialize a global variable, its initial value is guaranteed to be zero.
Here we have two declarations for \textit{a} – one as a global variable and one as a local variable. References to \textit{a} in \textit{main} are to the local variable, but elsewhere references are to the global variable.
Here $a$ is declared as a parameter to `proc`, thus references to $a$ in `proc` are to the parameter and not to the global variable.
Syntax error: one can’t have a local variable in a scope in which a parameter is declared with the same name.
Scope (more ...)

```c
int a; // global variable

int proc() {
    {
        // the brackets define a new scope
        int a;
        a = 6;
    }
    printf("a = %d\n", a); // what's printed?
    return 0;
}
```

```
$ ./a.out
0
```
Quiz 6

```c
int a;

int proc(int b) {
    int b=4;
    a = b;
    return a+2;
}

int main() {
    int a = proc(6);
    printf("a = %d\n", a);
    return 0;
}
```

- What’s printed?
  
a) 0
b) 4
c) 6
d) 8
e) nothing; there’s a syntax error