The Hardware/Software Interface
CSE351 Spring 2015
Lecture 3

Instructor:
Katelin Bailey

Teaching Assistants:
Announcements
Announcements

- Everyone waiting to get in should have heard by now
  - If we talked about auditing, please send me mail so I have your UWnetIDs
Announcements

• Everyone waiting to get in should have heard by now
  • If we talked about auditing, please send me mail so I have your UWnetIDs
• Lab 0 due on Monday, still having fun?
  • Covering arrays today in lecture
Announcements

• Everyone waiting to get in should have heard by now
  • If we talked about auditing, please send me mail so I have your UWnetIDs
• Lab 0 due on Monday, still having fun?
  • Covering arrays today in lecture
• Lab 1 goes out today, by the end of class
Everyone waiting to get in should have heard by now
  - If we talked about auditing, please send me mail so I have your UWnetIDs

Lab 0 due on Monday, still having fun?
  - Covering arrays today in lecture

Lab 1 goes out today, by the end of class

Stuck on something? Confused at the end of class?
  - Post on the discussion board
  - Come by office hours (mine or the TAs: posted online)
  - Send mail to cse351-staff@cse.uw.edu
Everyone waiting to get in should have heard by now
  • If we talked about auditing, please send me mail so I have your UWnetIDs

Lab 0 due on Monday, still having fun?
  • Covering arrays today in lecture

Lab 1 goes out today, by the end of class

Stuck on something? Confused at the end of class?
  • Post on the discussion board
  • Come by office hours (mine or the TAs: posted online)
  • Send mail to cse351-staff@cse.uw.edu

Readings for each lecture are posted on the website
Announcements

• Everyone waiting to get in should have heard by now
  • If we talked about auditing, please send me mail so I have your UWnetIDs

• Lab 0 due on Monday, still having fun?
  • Covering arrays today in lecture

• Lab 1 goes out today, by the end of class

• Stuck on something? Confused at the end of class?
  • Post on the discussion board
  • Come by office hours (mine or the TAs: posted online)
  • Send mail to cse351-staff@cse.uw.edu

• Readings for each lecture are posted on the website

• Slides for each lecture are posted ~1 day ahead of time
  • A 1-page summary of concepts and definitions will go up after class
    Not everything you need to know but most of the important topics.
Roadmap

C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:

```
get_mpg:
pushq   %rbp
movq    %rsp, %rbp
...
popq    %rbp
ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:

- Windows 8
- Mac

OS:

- Memory, data, & addressing
- Integers & floats
- Machine code & C
- x86 assembly
- Procedures & stacks
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C
Today

• Brief review of topics from last lecture
• Arrays and address arithmetic
• Strings as arrays
• Boolean algebra and bitwise manipulations
Today

- Brief review of topics from last lecture
- Arrays and address arithmetic
- Strings as arrays
- Boolean algebra and bitwise manipulations
Binary, Hex, Decimal Format (review)
Binary, Hex, Decimal Format (review)

- Byte = 8 bits (binary digits)
  = 2 hex digits
Binary, Hex, Decimal Format (review)

- Byte = 8 bits (binary digits)
  = 2 hex digits
- 0x 00 00 01 5F is hex format
  - 4 bytes —> 32 bits
Binary, Hex, Decimal Format (review)

- Byte = 8 bits (binary digits) = 2 hex digits
- 0x 00 00 01 5F is hex format
  - 4 bytes → 32 bits
- In Binary?
  - 0000 0000 0000 0000 0000 0001 0101 1111
  - 10101111
**Binary, Hex, Decimal Format (review)**

- **Byte = 8 bits (binary digits)**
  = 2 hex digits
- **0x 00 00 01 5F is hex format**
  - 4 bytes → 32 bits
- **In Binary?**
  - **0000 0000 0000 0000 0000 0001 0101 1111**
  - **10101111**

<table>
<thead>
<tr>
<th>bit numbering</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit numbering</td>
<td>$2^8$</td>
<td>$2^7$</td>
<td>$2^6$</td>
<td>$2^5$</td>
<td>$2^4$</td>
<td>$2^3$</td>
<td>$2^2$</td>
<td>$2^1$</td>
<td>$2^0$</td>
</tr>
</tbody>
</table>
Binary, Hex, Decimal Format (review)

- Byte = 8 bits (binary digits)  
  = 2 hex digits
- 0x 00 00 01 5F is hex format
  - 4 bytes —> 32 bits
- In Binary?
  - 0000 0000 0000 0000 0000 0001 0101 1111
  - 10101111
- In Decimal?
  - $2^8 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$
  - $256 + 64 + 16 + 8 + 4 + 2 + 1 = 351$
  - Bits are numbered from 0!
Words & Addresses (review)

- A **word** is fixed number of contiguous bytes in memory, chosen by HW
- the largest unit of data a machine instruction can use
Words & Addresses (review)

• A **word** is fixed number of contiguous bytes in memory, chosen by HW
  • the largest unit of data a machine instruction can use

• **word size = address size = register size (Always!)**
  • take advantage of all bytes we’re given, can’t use more than we have!
Words & Addresses (review)

- A **word** is fixed number of contiguous bytes in memory, chosen by HW
  - the largest unit of data a machine instruction can use
- **word size = address size = register size (Always!)**
  - take advantage of all bytes we’re given, can’t use more than we have!
- **Word size bounds the size of the address space and memory.**
  - word size = \( w \) bits \( \rightarrow \) \( 2^w \) addresses
  - 32-bit (4-byte) words \( \rightarrow \) 32-bit (4-byte) addresses \( \rightarrow \) \( 2^{32} \) bytes of space
  - 64-bit (8-byte) words \( \rightarrow \) 64-bit (8-byte) addresses \( \rightarrow \) \( 2^{64} \) bytes of space
Words & Addresses (review)

- A **word** is fixed number of contiguous bytes in memory, chosen by HW
  - the largest unit of data a machine instruction can use
- **word size = address size = register size (Always!)**
  - take advantage of all bytes we’re given, can’t use more than we have!
- **Word size bounds the size of the address space and memory.**
  - word size = \( w \) bits \( \rightarrow \) \( 2^w \) addresses
  - 32-bit (4-byte) words \( \rightarrow \) 32-bit (4-byte) addresses \( \rightarrow \) \( 2^{32} \) bytes of space
  - 64-bit (8-byte) words \( \rightarrow \) 64-bit (8-byte) addresses \( \rightarrow \) \( 2^{64} \) bytes of space
- Doesn’t mean we have \( 2^{32} \) bytes (4GB) of memory, just means we could find all of those bytes *if* we had them.
Memory Alignment (review)

- Data of size $n$ only stored at addresses $a$ where $a \mod n = 0$
  - $n$ is usually a power of 2.
- A 32-bit (4-byte) word-aligned view of memory:
  - Each row is a word composed of 4 bytes.
  - Cells in a row are the word’s bytes.

More about alignment later in the course.
Addresses and Pointers (review)

- An **address** is a location in memory.
- A **pointer** is a data object that holds an address.
- The value 351 is stored at address **0x04**.
  - \( 351_{10} = 15F_{16} = 0x00 \ 00 \ 01 \ 5F \)
- A pointer stored at address **0x1C**
  points to address **0x04**.
- A pointer to a pointer
  is stored at address **0x24**.
- The value 12 is stored
  at address **0x14**.
  - Is it a pointer?
## Big & Little Endian (review)

### Table: Big & Little Endian Formats

<table>
<thead>
<tr>
<th>Format</th>
<th>Size</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Little</td>
<td>4B</td>
<td>0x00 0x00 0x01 0x5F</td>
</tr>
<tr>
<td>Little</td>
<td>8B</td>
<td>0x00 0x00 0x00 0x04</td>
</tr>
<tr>
<td>Little</td>
<td>12B</td>
<td>0x00 0x00 0x00 0x1C</td>
</tr>
<tr>
<td>Little</td>
<td>16B</td>
<td>0x00 0x00 0x00 0x24</td>
</tr>
<tr>
<td>Big</td>
<td>4B</td>
<td>0x5F 0x01 0x00 0x00</td>
</tr>
<tr>
<td>Big</td>
<td>8B</td>
<td>0x1C 0x00 0x00 0x00</td>
</tr>
<tr>
<td>Big</td>
<td>12B</td>
<td>0x24 0x00 0x00 0x00</td>
</tr>
<tr>
<td>Big</td>
<td>16B</td>
<td>0x04 0x00 0x00 0x00</td>
</tr>
</tbody>
</table>

### Diagram:

- Little Endian: Upper bytes are written first.
- Big Endian: Lower bytes are written first.
Big & Little Endian (review)
Big & Little Endian (review)

Little-Endian

Big-Endian

0x00
0x04
0x08
0x0C
0x10
0x14
0x18
0x1C
0x20
0x24
Big & Little Endian *(review)*

<table>
<thead>
<tr>
<th>Little-Endian</th>
<th>Big-Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>5F 01 00 00</td>
<td>0x00 0x04 0x08 0x10 0x14 0x18 0x1C 0x20 0x24</td>
</tr>
<tr>
<td>1C 00 00 00</td>
<td>0x00 0x04 0x08 0x0C 0x10 0x14 0x18 0x1C 0x20 0x24</td>
</tr>
<tr>
<td>04 00 00 00</td>
<td>00 00 01 5F 00 00 00 1C 00 04 0x1C 0x14 0x18 0x20 0x24</td>
</tr>
<tr>
<td>1C 00 00 00</td>
<td>00 00 00 1C 00 00 00 04 0x1C 0x14 0x18 0x20 0x24</td>
</tr>
</tbody>
</table>
Assignment in C (review)

- Left-hand-side = right-hand-side;
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (Could be an address!)
  - Store RHS value at LHS location.
- int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - // Get value at y, add 3, put it in x.
- int *z

& = ‘address of’
* = ‘value at address’
or ‘dereference’
Assignment in C (review)

- Left-hand-side = right-hand-side;
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (Could be an address!)
  - Store RHS value at LHS location.

- int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - // Get value at y, add 3, put it in x.
- int *z = &y + 3;
  - // Get address of y, add 12 put it in z

\[ & = \text{‘address of’} \]
\[ * = \text{‘value at address’} \] or ‘dereference’
Assignment in C (review)

- **Left-hand-side = right-hand-side;**
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (Could be an address!)
  - Store RHS value at LHS location.

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - // Get value at y, add 3, put it in x.
- `int *z = &y + 3;`
  - // Get address of y, add 12 put it in z

Pointer arithmetic is scaled by size of target type.
Assignment in C (review)

- Left-hand-side = right-hand-side;
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (Could be an address!)
  - Store RHS value at LHS location.
- int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - // Get value at y, add 3, put it in x.
- int *z = &y + 3;
  - // Get address of y, add 12 put it in z

```
0x18 = 24 (decimal)
+ 12
36 = 0x24
```

```
0x00
0x04 x
0x08
0x0C
0x10
0x14
0x18 y
0x1C
0x20 z
0x24
```

& = ‘address of’
*
= ‘value at address’
or ‘dereference’

Pointer arithmetic is scaled by size of target type.
Assignment in C (review)

- Left-hand-side = right-hand-side;
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (Could be an address!)
  - Store RHS value at LHS location.
- int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - // Get value at y, add 3, put it in x.
- int *z = &y + 3;
  - // Get address of y, add 12 put it in z

Pointer arithmetic can be dangerous.

& = ‘address of’
* = ‘value at address’ or ‘dereference’

Pointer arithmetic is scaled by size of target type.

\[
0x18 = 24 \text{ (decimal)} \\
0x24
\]

\[
\frac{12}{36} = 0x24
\]
Assignment in C (review)

- Left-hand-side = right-hand-side;
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (Could be an address!)
  - Store RHS value at LHS location.

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - // Get value at y, add 3, put it in x.
- `int *z = &y + 3;`
  - // Get address of y, add 12 put it in z
- `*z = y;`
  - // Get value of y, put it at the address stored in z
Today

• Brief review of topics from last lecture
• **Arrays and address arithmetic**
• Strings as arrays
• Boolean algebra and bitwise manipulations
Arrays in C

Declaration: `int a[6];`

Arrays are adjacent locations in memory storing the same type of data object.

*a* is a name for the array’s address, not a pointer to the array.

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x00</td>
</tr>
<tr>
<td>1</td>
<td>0x04</td>
</tr>
<tr>
<td>2</td>
<td>0x08</td>
</tr>
<tr>
<td>3</td>
<td>0x0C</td>
</tr>
<tr>
<td>4</td>
<td>0x10</td>
</tr>
<tr>
<td>5</td>
<td>0x14</td>
</tr>
<tr>
<td>6</td>
<td>0x18</td>
</tr>
<tr>
<td>7</td>
<td>0x1C</td>
</tr>
<tr>
<td>8</td>
<td>0x20</td>
</tr>
<tr>
<td>9</td>
<td>0x24</td>
</tr>
</tbody>
</table>
Arrays in C

Declaration: \( \text{int a}[6]; \)

Arrays are adjacent locations in memory storing the same type of data object. 

\( a \) is a name for the array’s address, not a pointer to the array.
Arrays in C

Declaration: `int a[6];`

Arrays are adjacent locations in memory storing the same type of data object.

`a` is a name for the array’s address, not a pointer to the array.
Arrays in C

Declaration: `int a[6];`

- **element type**: `int`
- **name**: `a`
- **number of elements**: `6`

Arrays are adjacent locations in memory storing the same type of data object.

`a` is a name for the array’s address, not a pointer to the array.

| 0x00 | 0x04 | 0x08 | 0x0C | 0x10 | 0x14 | 0x18 | 0x1C | 0x20 | 0x24 |
Arrays in C

Declaration:  `int a[6];`

Indexing:

Arrays are adjacent locations in memory storing the same type of data object.

`a` is a name for the array’s address, not a pointer to the array.

The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes.
Arrays in C

Declaration:  
\[
\text{int } a[6];
\]

Indexing:  
\[
a[0] = 0x015f; \\
a[5] = a[0];
\]

Arrays are adjacent locations in memory storing the same type of data object. 

a is a name for the array’s address, not a pointer to the array.

The address of \(a[i]\) is the address of \(a[0]\) plus \(i\) times the element size in bytes.
Arrays in C

Declaration: int a[6];

Indexing: a[0] = 0x015f;
a[5] = a[0];

Arrays are adjacent locations in memory storing the same type of data object.
a is a name for the array’s address, not a pointer to the array.
The address of a[i] is the address of a[0] plus i times the element size in bytes.
Arrays in C

Declaration: \[ \text{int } a[6]; \]

Indexing: \[ a[0] = 0x01 5f; \]
\[ a[5] = a[0]; \]

Arrays are adjacent locations in memory storing the same type of data object.

\( a \) is a name for the array’s address, not a pointer to the array.

The address of \( a[i] \) is the address of \( a[0] \) plus \( i \) times the element size in bytes.
Arrays in C

Declaration: \( \text{int } a[6]; \)

Indexing: \( a[0] = 0x015f; \)
\( a[5] = a[0]; \)

Arrays are adjacent locations in memory storing the same type of data object.

\( a \) is a name for the array’s address, not a pointer to the array.

The address of \( a[i] \) is the address of \( a[0] \) plus \( i \) times the element size in bytes.

\( \text{sizeof(int)} = 4 = \text{word size (convenient)} \)
Arrays in C

Declaration: \( \text{int } a[6]; \)

Indexing: \( a[0] = 0x015f; \)
\( a[5] = a[0]; \)

No bounds check:

Arrays are adjacent locations in memory storing the same type of data object.

\( a \) is a name for the array’s address, not a pointer to the array.

The address of \( a[i] \) is the address of \( a[0] \) plus \( i \) times the element size in bytes.
Arrays in C

Declaration: \( \text{int a}[6]; \)

Indexing: \( \text{a}[0] = 0x015f; \)
\( \text{a}[5] = \text{a}[0]; \)

No bounds check: \( \text{a}[6] = 0x\text{BAD}; \)
\( \text{a}[-1] = 0x\text{BAD}; \)

Arrays are adjacent locations in memory storing the same type of data object.

\( \text{a} \) is a name for the array’s address, not a pointer to the array.

The address of \( \text{a}[i] \) is the address of \( \text{a}[0] \) plus \( i \) times the element size in bytes.
Arrays in C

Declaration:  
```
int a[6];
```

Indexing:  
```
a[0] = 0x015f;
a[5] = a[0];
```

No bounds check:  
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Arrays are adjacent locations in memory storing the same type of data object.

The address of a[i] is the address of a[0] plus i times the element size in bytes.

a is a name for the array’s address, not a pointer to the array.
Arrays in C

Declaration: int a[6];

Indexing: a[0] = 0x015f;
          a[5] = a[0];

No bounds check: a[6] = 0xBAD;
                 a[-1] = 0xBAD;

Arrays are adjacent locations in memory storing the same type of data object.

a is a name for the array’s address, not a pointer to the array.

The address of a[i] is the address of a[0] plus i times the element size in bytes.

```
<table>
<thead>
<tr>
<th>AD</th>
<th>0B</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>5F</td>
<td>01</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>AD</td>
<td>0B</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
```

0x00 0x04 a[0] 0x08 a[1] 0x0C 0x10 0x14 ...
0x18 a[5] 0x1C 0x20 0x24
**Arrays in C**

**Declaration:**
```c
int a[6];
```

**Indexing:**
- `a[0] = 0x015f;`
- `a[5] = a[0];`

**No bounds check:**
- `a[6] = 0xBAD;`
- `a[-1] = 0xBAD;`

**Pointers:**
Arrays are adjacent locations in memory storing the same type of data object.

- `a` is a name for the array’s address, not a pointer to the array.

The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes.

```
<table>
<thead>
<tr>
<th>AD</th>
<th>0B</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>5F</td>
<td>01</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>AD</td>
<td>0B</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
```

- `0x00`  `a[0]`
- `0x04`  `a[1]`
- `0x08`  `a[2]`
- `0x0C`  `a[3]`
- `0x10`  `a[4]`
- `0x14`  `a[5]`
- `0x18`  `a[6]`
- `0x1C`
- `0x20`
- `0x24`
Arrays in C

Declaration: \( \text{int } a[6]; \)

Indexing:
- \( a[0] = 0x015f; \)
- \( a[5] = a[0]; \)

No bounds check:
- \( a[6] = 0xBAD; \)
- \( a[-1] = 0xBAD; \)

Pointers:
- \( \text{int}\* p; \)
- \( p = a; \)
- \( p = &a[0]; \)

Arrays are adjacent locations in memory storing the same type of data object.

\( a \) is a name for the array’s address, not a pointer to the array.

The address of \( a[i] \) is the address of \( a[0] \) plus \( i \) times the element size in bytes.
Arrays in C

Declaration:

```
int a[6];
```

Indexing:

```
a[0] = 0x015f;
a[5] = a[0];
```

No bounds check:

```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:

```
int* p;
```

```
p = a;
p = &a[0];
```

Arrays are adjacent locations in memory storing the same type of data object.

\[a\] is a name for the array’s address, not a pointer to the array.

The address of \[a[i]\] is the address of \[a[0]\] plus \(i\) times the element size in bytes.

```
AD 0B 00 00
5F 01 00 00
```

0x00
0x04 \(a[0]\)
0x08 \(a[1]\)
0x0C 
0x10 ...
0x14 
0x18 \(a[5]\)
0x1C 
0x20 
0x24
Arrays in C

Declaration:  
int a[6];

Indexing:   
a[0] = 0x015f;
a[5] = a[0];

No bounds check:  
a[6] = 0xBAD;
a[-1] = 0xBAD;

Pointers:  
int* p;

equivalent {  
p = a;
p = &a[0];

arrays are adjacent locations in memory storing the same type of data object.
a is a name for the array’s address, not a pointer to the array.
The address of a[i] is the address of a[0] plus i times the element size in bytes.

0x00 0x04 a[0]
0x08 a[1]
0x0C
0x10 ...
0x14
0x18 a[5]
0x1C
0x20 p
0x24
Arrays in C

Declaration: \(\text{int } a[6];\)

Indexing: 
\(a[0] = 0x015f;\)
\(a[5] = a[0];\)

No bounds check:
\(a[6] = 0xBAD;\)
\(a[-1] = 0xBAD;\)

Pointers:
\(\text{int* } p;\)
\(p = a;\)
\(p = \&a[0];\)
\(*p = 0xA;\)
Arrays in C

Declaration:  \( \text{int } a[6]; \)

Indexing:  
- \( a[0] = 0x015f; \)
- \( a[5] = a[0]; \)

No bounds check:
- \( a[6] = 0xBAD; \)
- \( a[-1] = 0xBAD; \)

Pointers:
\[
\begin{align*}
\text{int* } p; \\
\text{p = a; } \\
\text{p = &a[0]; } \\
\text{*p = 0xA; }
\end{align*}
\]

Arrays are adjacent locations in memory storing the same type of data object.

\( a \) is a name for the array’s address, not a pointer to the array.

The address of \( a[i] \) is the address of \( a[0] \) plus \( i \) times the element size in bytes.
Arrays in C

Declaration:  
\[\text{int } a[6];\]

Indexing:  
\[a[0] = 0x015f;\]
\[a[5] = a[0];\]

No bounds check:  
\[a[6] = 0xBAD;\]
\[a[-1] = 0xBAD;\]

Pointers:  
\[
\begin{align*}
\text{int } &p; \\
p & = a; \\
p & = &a[0]; \\
*p & = 0xA; \\
p[1] & = 0xB;
\end{align*}
\]

Arrays are adjacent locations in memory storing the same type of data object.  
\(a\) is a name for the array’s address, not a pointer to the array.

The address of \(a[i]\) is the address of \(a[0]\) plus \(i\) times the element size in bytes.
Arrays in C

Declaration: \[ \text{int } a[6]; \]

Indexing: \[ a[0] = 0x015f; \]
\[ a[5] = a[0]; \]

No bounds check:
\[ a[6] = 0xBAD; \]
\[ a[-1] = 0xBAD; \]

Pointers:
\[ \text{int* } p; \]
\[ p = a; \]
\[ p = \&a[0]; \]
\[ *p = 0xA; \]
\[ p[1] = 0xB; \]

array indexing = address arithmetic
Both are scaled by the size of the type.

Arrays are adjacent locations in memory storing the same type of data object.

\( a \) is a name for the array’s address, not a pointer to the array.

The address of \( a[i] \) is the address of \( a[0] \) plus \( i \) times the element size in bytes.
Arrays in C

Declaration:  int a[6];

Indexing:  a[0] = 0x015f;
a[5] = a[0];

No bounds check:  a[6] = 0xBAD;
a[-1] = 0xBAD;

Pointers:  int* p;
  p = a;
  p = &a[0];
  *p = 0xA;

equivalent  

array indexing = address arithmetic
Both are scaled by the size of the type.

Arrays are adjacent locations in memory storing the same type of data object.
a is a name for the array’s address, not a pointer to the array.
The address of a[i] is the address of a[0] plus i times the element size in bytes.
Arrays in C

Declaration:  \[ \text{int } a[6]; \]

Indexing:  \[ a[0] = 0x015f; \]
\[ a[5] = a[0]; \]

No bounds check:  \[ a[6] = 0xBAD; \]
\[ a[-1] = 0xBAD; \]

Pointers:  \[ \text{int* } p; \]
\[
\begin{align*}
  &p = a; \\
  &p = &a[0]; \\
  &*p = 0xA;
\end{align*}
\]

\[ p[1] = 0xFB; \]
\[ *(p + 1) = 0xB; \]
\[ p = p + 2; \]

\textit{array indexing = address arithmetic}
Both are scaled by the size of the type.

Arrays are adjacent locations in memory storing the same type of data object.

\[ \text{a is a name for the array's address, not a pointer to the array.} \]

The address of \( a[i] \) is the address of \( a[0] \) plus \( i \) times the element size in bytes.
Arrays in C

Declaration: \[
\text{int a[6];}
\]

Indexing: \[
\begin{align*}
&\text{a[0]} = 0x015f; \\
&a[5] = a[0];
\end{align*}
\]

No bounds check: \[
\begin{align*}
&a[6] = 0xBAD; \\
&a[-1] = 0xBAD;
\end{align*}
\]

Pointers: \[
\begin{align*}
&\text{int* p;} \\
&p = \text{a;} \\
&p = \&a[0]; \\
*p = 0xA;
\end{align*}
\]

\[
\begin{align*}
&p[1] = 0xB; \\
*(p + 1) = 0xB; \\
&p = p + 2;
\end{align*}
\]

array indexing = address arithmetic
Both are scaled by the size of the type.

Arrays are adjacent locations in memory storing the same type of data object.

\[
\begin{array}{cccccc}
AD & 0B & 00 & 00 & 00 \\
0A & 00 & 00 & 00 \\
0B & 00 & 00 & 00 \\
5F & 01 & 00 & 00 \\
AD & 0B & 00 & 00 \\
0C & 00 & 00 & 00 \\
\end{array}
\]

\[
\begin{array}{cccccc}
0x00 & 0x04 & a[0] & 0x08 & a[1] & 0x0C \\
0x10 & \ldots & 0x14 & 0x18 & a[5] & 0x1C \\
0x20 & p & 0x24
\end{array}
\]

\[
\begin{align*}
&a \text{ is a name for the array’s address, not a pointer to the array.} \\
&\text{The address of a[i] is the address of a[0] plus i times the element size in bytes.}
\end{align*}
\]
Arrays in C

Declaration:

```c
int a[6];
```

Indexing:

```c
a[0] = 0x015f;
a[5] = a[0];
```

No bounds check:

```c
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:

```c
int* p;
p = a;
p = &a[0];
*p = 0xA;
```

```
 AD 0B 00 00
0A 00 00 00
0B 00 00 00
```

```
 5F 01 00 00
AD 0B 00 00
0C 00 00 00
```

```
 0x00
0x04 a[0]
0x08 a[1]
0x0C ...
0x10
0x14
0x18 a[5]
0x1C
0x20
0x24 p
```

Arrays are adjacent locations in memory storing the same type of data object.

*a* is a name for the array’s address, not a pointer to the array.

The address of a[i] is the address of a[0] plus i times the element size in bytes.

**array indexing = address arithmetic**

Both are scaled by the size of the type.
Arrays in C

Declaration: \( \text{int } a[6]; \)

Indexing: \( a[0] = 0x015f; \)
\( a[5] = a[0]; \)

No bounds check: \( a[6] = 0xBAD; \)
\( a[-1] = 0xBAD; \)

Pointers:
\[
\begin{align*}
\text{int}^* & \ p; \\
& \ p = a; \\
& \ p = &a[0]; \\
& \ *p = 0xA;
\end{align*}
\]

\[
\begin{align*}
& \ p[1] = 0xB; \\
& \ *(p + 1) = 0xB; \\
& \ p = p + 2; \\
& \ *p = a[1] + 1;
\end{align*}
\]

*array indexing = address arithmetic*
Both are scaled by the size of the type.

Arrays are adjacent locations in memory storing the same type of data object.

\( a \) is a name for the array’s address, not a pointer to the array.

The address of \( a[i] \) is the address of \( a[0] \) plus \( i \) times the element size in bytes.

\[
\begin{array}{cccc}
\text{AD} & 0B & 00 & 00 \\
0A & 00 & 00 & 00 \\
0B & 00 & 00 & 00 \\
0C & 00 & 00 & 00 \\
0D & 00 & 00 & 00 \\
\end{array}
\]

\( 0x00 \)
\( 0x04 \ a[0] \)
\( 0x08 \ a[1] \)
\( 0x0C \)
\( 0x10 \ ... \)
\( 0x14 \)
\( 0x18 \ a[5] \)
\( 0x1C \)
\( 0x20 \ p \)
\( 0x24 \)
Today

- Brief review of topics from last lecture
- Arrays and address arithmetic
- **Strings as arrays**
- Boolean algebra and bitwise manipulations
Representing strings

- A C-style string is represented by an array of bytes \((\text{char})\).
  - Elements are one-byte ASCII codes for each character.
  - ASCII = American Standard Code for Information Interchange
Representing strings

- A C-style string is represented by an array of bytes (char).
  - Elements are one-byte ASCII codes for each character.
  - ASCII = American Standard Code for Information Interchange

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>space</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
</tr>
<tr>
<td>39</td>
<td>'</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>51</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>5</td>
</tr>
<tr>
<td>54</td>
<td>6</td>
</tr>
<tr>
<td>55</td>
<td>7</td>
</tr>
<tr>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>57</td>
<td>9</td>
</tr>
<tr>
<td>58</td>
<td>:</td>
</tr>
<tr>
<td>59</td>
<td>;</td>
</tr>
<tr>
<td>60</td>
<td>&lt;</td>
</tr>
<tr>
<td>61</td>
<td>=</td>
</tr>
<tr>
<td>62</td>
<td>&gt;</td>
</tr>
<tr>
<td>63</td>
<td>?</td>
</tr>
<tr>
<td>64</td>
<td>@</td>
</tr>
<tr>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>69</td>
<td>E</td>
</tr>
<tr>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>74</td>
<td>J</td>
</tr>
<tr>
<td>75</td>
<td>K</td>
</tr>
<tr>
<td>76</td>
<td>L</td>
</tr>
<tr>
<td>77</td>
<td>M</td>
</tr>
<tr>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>79</td>
<td>O</td>
</tr>
<tr>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>81</td>
<td>Q</td>
</tr>
<tr>
<td>82</td>
<td>R</td>
</tr>
<tr>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>85</td>
<td>U</td>
</tr>
<tr>
<td>86</td>
<td>V</td>
</tr>
<tr>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>88</td>
<td>X</td>
</tr>
<tr>
<td>89</td>
<td>Y</td>
</tr>
<tr>
<td>90</td>
<td>Z</td>
</tr>
<tr>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>95</td>
<td>_</td>
</tr>
<tr>
<td>96</td>
<td>`</td>
</tr>
<tr>
<td>97</td>
<td>a</td>
</tr>
<tr>
<td>98</td>
<td>b</td>
</tr>
<tr>
<td>99</td>
<td>c</td>
</tr>
<tr>
<td>100</td>
<td>d</td>
</tr>
<tr>
<td>101</td>
<td>e</td>
</tr>
<tr>
<td>102</td>
<td>f</td>
</tr>
<tr>
<td>103</td>
<td>g</td>
</tr>
<tr>
<td>104</td>
<td>h</td>
</tr>
<tr>
<td>105</td>
<td>i</td>
</tr>
<tr>
<td>106</td>
<td>j</td>
</tr>
<tr>
<td>107</td>
<td>k</td>
</tr>
<tr>
<td>108</td>
<td>l</td>
</tr>
<tr>
<td>109</td>
<td>m</td>
</tr>
<tr>
<td>110</td>
<td>n</td>
</tr>
<tr>
<td>111</td>
<td>o</td>
</tr>
<tr>
<td>112</td>
<td>p</td>
</tr>
<tr>
<td>113</td>
<td>q</td>
</tr>
<tr>
<td>114</td>
<td>r</td>
</tr>
<tr>
<td>115</td>
<td>s</td>
</tr>
<tr>
<td>116</td>
<td>t</td>
</tr>
<tr>
<td>117</td>
<td>u</td>
</tr>
<tr>
<td>118</td>
<td>v</td>
</tr>
<tr>
<td>119</td>
<td>w</td>
</tr>
<tr>
<td>120</td>
<td>x</td>
</tr>
<tr>
<td>121</td>
<td>y</td>
</tr>
<tr>
<td>122</td>
<td>z</td>
</tr>
<tr>
<td>123</td>
<td>{</td>
</tr>
<tr>
<td>124</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>}</td>
</tr>
<tr>
<td>126</td>
<td>~</td>
</tr>
<tr>
<td>127</td>
<td>del</td>
</tr>
</tbody>
</table>
Null-terminated Strings
Null-terminated Strings

- For example, “Harry Potter” can be stored as a 13-byte array.
Null-terminated Strings

- For example, “Harry Potter” can be stored as a 13-byte array.

<table>
<thead>
<tr>
<th>72</th>
<th>97</th>
<th>114</th>
<th>114</th>
<th>121</th>
<th>32</th>
<th>80</th>
<th>111</th>
<th>116</th>
<th>116</th>
<th>101</th>
<th>114</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>a</td>
<td>r</td>
<td>y</td>
<td>P</td>
<td>o</td>
<td>t</td>
<td>t</td>
<td>e</td>
<td>r</td>
<td>\0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Harry Potter \0
Null-terminated Strings

- For example, “Harry Potter” can be stored as a 13-byte array.

```
| 72 | 97 | 114 | 114 | 121 | 32 | 80 | 111 | 116 | 116 | 101 | 114 | 0 |
```

Harry Potter \0

- Why do we put a 0, or null zero, at the end of the string?
  - Note the special symbol: `string[12] = '\0';`
Null-terminated Strings

- For example, “Harry Potter” can be stored as a 13-byte array.

<table>
<thead>
<tr>
<th>72</th>
<th>97</th>
<th>114</th>
<th>114</th>
<th>121</th>
<th>32</th>
<th>80</th>
<th>111</th>
<th>116</th>
<th>116</th>
<th>101</th>
<th>114</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Why do we put a 0, or null zero, at the end of the string?
  - Note the special symbol: string[12] = '\0';

- How do we compute the string length?
Endianness and Strings
• Byte ordering (endianness) is not an issue for 1-byte values.
  • Arrays are not values; elements are values; chars are single bytes.
Endianness and Strings

- **Byte ordering (endianness)** is not an issue for 1-byte values.
  - Arrays are not values; elements are values; chars are single bytes.
- **Unicode characters** – up to 4 bytes/character
  - ASCII codes still work (just add leading zeros).
    Unicode can support the many characters in all languages in the world.
  - Java and C have libraries for Unicode (Java commonly uses 2 bytes/char)
Endianness and Strings

C (char = 1 byte)
char s[6] = "12345";

<table>
<thead>
<tr>
<th>IA32, x86-64</th>
<th>SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>

- Byte ordering (endianness) is not an issue for 1-byte values.
  - Arrays are not values; elements are values; chars are single bytes.

- **Unicode characters – up to 4 bytes/character**
  - ASCII codes still work (just add leading zeros).
    Unicode can support the many characters in all languages in the world.
  - Java and C have libraries for Unicode (Java commonly uses 2 bytes/char)
Endianness and Strings

**C (char = 1 byte)**

```c
char s[6] = "12345";
```

IA32, x86-64  SPARC

| 31 | 31 |
| 32 | 32 |
| 33 | 33 |
| 34 | 34 |
| 35 | 35 |
| 00 | 00 |

**Java (char = 2 bytes)**

```java
String s = "123";
```

(not all of the String representation is shown)

IA32, x86-64  SPARC

| 31 | 00 |
| 32 | 00 |
| 33 | 00 |

- Byte ordering (endianness) is not an issue for 1-byte values.
  - Arrays are not values; elements are values; chars are single bytes.

- **Unicode characters – up to 4 bytes/character**
  - ASCII codes still work (just add leading zeros).
    Unicode can support the many characters in all languages in the world.
  - Java and C have libraries for Unicode (Java commonly uses 2 bytes/char)
Today

• Brief review of topics from last lecture
• Arrays and address arithmetic
• Strings as arrays
• Boolean algebra and bitwise manipulations
Examining Data Representations

- Code to print byte representation of data
  - Any data type can be treated as a byte array by casting it to char.
  - C has unchecked casts. << DANGER >>

printf directives:
- %p  Print pointer
- \t  Tab
- %x  Print value as hex
- \n  New line
Examining Data Representations

- Code to print byte representation of data
  - Any data type can be treated as a byte array by casting it to char.
  - C has unchecked casts. << DANGER >>

```c
typedef char byte;  // size of char == 1 byte

void show_bytes(byte* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
```

printf directives:
- `%p` Print pointer
- `\t` Tab
- `%x` Print value as hex
- `\n` New line
Examining Data Representations

• Code to print byte representation of data
  • Any data type can be treated as a byte array by casting it to char.
  • C has unchecked casts. << DANGER >>

```c
typedef char byte; // size of char == 1 byte

void show_bytes(byte* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
```

printf directives:
- `%p` Print pointer
- `	` Tab
- `%x` Print value as hex
- `\n` New line

```c
void show_int (int x) {
    show_bytes( (byte *) &x, sizeof(int));
}
```
show_bytes  Execution Example
show_bytes Execution Example

```c
int a = 12345;       // represented as 0x00003039
printf("int a = 12345;\n");
show_int(a);         // show_bytes((pointer) &a, sizeof(int));
```
int a = 12345; // represented as 0x00003039
printf("int a = 12345;\n");
show_int(a); // show_bytes((pointer) &a, sizeof(int));

Result (Linux):

int a = 12345;
0x11ffffffcb8 0x39
0x11ffffffcb9 0x30
0x11ffffffcba 0x00
0x11ffffffcbb 0x00
Boolean Algebra
Boolean Algebra

- Developed by George Boole in 19th Century
Boolean Algebra

- Developed by George Boole in 19th Century
- Algebraic representation of logic
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
    - AND: A&B = 1 when both A is 1 and B is 1
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - AND: $A \& B = 1$ when both $A$ is 1 and $B$ is 1
  - OR: $A \mid B = 1$ when either $A$ is 1 or $B$ is 1
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
    - AND: \( A \& B = 1 \) when both \( A \) is 1 and \( B \) is 1
    - OR: \( A \mid B = 1 \) when either \( A \) is 1 or \( B \) is 1
    - XOR: \( A \uparrow B = 1 \) when either \( A \) is 1 or \( B \) is 1, but not both
Boolean Algebra

• Developed by George Boole in 19th Century
  • Algebraic representation of logic
    • Encode “True” as 1 and “False” as 0
  • AND: $A \& B = 1$ when both $A$ is 1 and $B$ is 1
  • OR: $A \mid B = 1$ when either $A$ is 1 or $B$ is 1
  • XOR: $A \wedge B = 1$ when either $A$ is 1 or $B$ is 1, but not both
  • NOT: $\sim A = 1$ when $A$ is 0 and vice-versa
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - AND: $A \& B = 1$ when both $A$ is 1 and $B$ is 1
  - OR: $A \mid B = 1$ when either $A$ is 1 or $B$ is 1
  - XOR: $A \^ B = 1$ when either $A$ is 1 or $B$ is 1, but not both
  - NOT: $\sim A = 1$ when $A$ is 0 and vice-versa
  - DeMorgan’s Law: $\sim(A \mid B) = \sim A \& \sim B$
Boolean Algebra

• Developed by George Boole in 19th Century
  • Algebraic representation of logic
    • Encode “True” as 1 and “False” as 0
  • AND: A&B = 1 when both A is 1 and B is 1
  • OR: A|B = 1 when either A is 1 or B is 1
  • XOR: A^B = 1 when either A is 1 or B is 1, but not both
  • NOT: ~A = 1 when A is 0 and vice-versa
  • DeMorgan’s Law: ~(A | B) = ~A & ~B

\[
\begin{array}{c|cc}
\& & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & 1 \\
\end{array}
\]
Boolean Algebra

• Developed by George Boole in 19th Century
  • Algebraic representation of logic
    • Encode “True” as 1 and “False” as 0
  • AND: A&B = 1 when both A is 1 and B is 1
  • OR: A|B = 1 when either A is 1 or B is 1
  • XOR: A^B = 1 when either A is 1 or B is 1, but not both
  • NOT: ~A = 1 when A is 0 and vice-versa
  • DeMorgan’s Law: ~(A | B) = ~A & ~B

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode “True” as 1 and “False” as 0
  - AND: $A \& B = 1$ when both $A$ is 1 and $B$ is 1
  - OR: $A | B = 1$ when either $A$ is 1 or $B$ is 1
  - XOR: $A ^ B = 1$ when either $A$ is 1 or $B$ is 1, but not both
  - NOT: $\neg A = 1$ when $A$ is 0 and vice-versa
  - DeMorgan’s Law: $\neg(A | B) = \neg A \& \neg B$

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>^</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Boolean Algebra

• Developed by George Boole in 19th Century
  • Algebraic representation of logic
    • Encode “True” as 1 and “False” as 0
  • AND: A&B = 1 when both A is 1 and B is 1
  • OR: A|B = 1 when either A is 1 or B is 1
  • XOR: A^B = 1 when either A is 1 or B is 1, but not both
  • NOT: ~A = 1 when A is 0 and vice-versa
  • DeMorgan’s Law: ~(A | B) = ~A & ~B
General Boolean Algebras
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{array}{c}
01101001 \\
\& 01010101 \\
\hline
01000001
\end{array}
\]

\[
\begin{array}{c}
\hline
\hline
\hline
\end{array}
\]

36
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{array}{c}
01101001 \\
\& 01010101 \\
\hline
01000001
\end{array}
\begin{array}{c}
01101001 \\
\mid 01010101 \\
\hline
01111101
\end{array}
\]

- 
- 
- 
-
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{array}{c}
01101001 \\
\& 01010101 \\
\hline
01000001
\end{array} \quad \begin{array}{c}
01101001 \\
| 01010101 \\
\hline
01111101
\end{array} \quad \begin{array}{c}
01101001 \\
^ 01010101 \\
\hline
00111100
\end{array}
\]
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

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

\[ 01101001 \& 01010101 = 01000001 \]
\[ 01101001 \mid 01010101 = 01111101 \]
\[ 01101001 \^ 01010101 = 00111100 \]
\[ 01010101 \~ 01010101 = 10101010 \]
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{align*}
01101001 \& 01010101 &= 01000001 \\
01010101 \mid 01010101 &= 01111101 \\
01101001 \^ 01010101 &= 00111100 \\
\sim 01010101 &= 10101010
\end{align*}
\]

- All of the properties of Boolean algebra apply
General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise

\[
\begin{array}{c}
01101001 \\
& \phantom{01101001} 01010101 \\
\hline
01000001 \\
\end{array}
\begin{array}{c}
01101001 \\
| \phantom{01101001} 01010101 \\
\hline
01111101 \\
\end{array}
\begin{array}{c}
01101001 \\
^ {01010101} \\
\hline
00111100 \\
\end{array}
\begin{array}{c}
01010101 \\
\sim \phantom{01010101} 01010101 \\
\hline
10101010 \\
\end{array}
\]

- All of the properties of Boolean algebra apply

\[
\begin{array}{c}
01010101 \\
\wedge \phantom{01010101} 01010101 \\
\hline
00000000 \\
\end{array}
\]
General Boolean Algebras

• Operate on bit vectors
  • Operations applied bitwise

  \[
  \begin{array}{ccc}
  01101001 & 01101001 & 01101001 \\
  \& 01010101 & | 01010101 & ^ 01010101 \\
  01000001 & 01111101 & 00111100 & \sim 01010101 \\
  \end{array}
  \]

• All of the properties of Boolean algebra apply

  \[
  \begin{array}{c}
  01010101 \\
  ^ \ 01010101 \\
  00000000
  \end{array}
  \]

• How does this relate to set operations?
Representing & Manipulating Sets
Representing & Manipulating Sets

- Representation
Representing & Manipulating Sets

• Representation
  • A $w$-bit vector represents subsets of $\{0, \ldots, w-1\}$
Representing & Manipulating Sets

• Representation
  • A \textit{w}-bit vector represents subsets of \{0, \ldots, w-1\}
  • \(a_j = 1\) iff \(j \in A\)
Representing & Manipulating Sets

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

01101001 \hspace{2cm} \{ 0, 3, 5, 6 \}
Representing & Manipulating Sets

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

01101001  
\{ 0, 3, 5, 6 \}

76543210
Representing & Manipulating Sets

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

01101001

$\{0, 3, 5, 6\}$

76543210
Representing & Manipulating Sets

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

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

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

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

• Representation
  • A $w$-bit vector represents subsets of $\{0, \ldots, w-1\}$
  • $a_j = 1$ iff $j \in A$

    01101001  \quad \{ 0, 3, 5, 6 \}

    76543210

    01010101  \quad \{ 0, 2, 4, 6 \}

    76543210

• Operations
Representing & Manipulating Sets

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

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

• Operations
  • \& Intersection

\[
01000001 \quad \{0, 6\}
\]
Representing & Manipulating Sets

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

  01101001   \{ 0, 3, 5, 6 \}
  76543210

  01010101   \{ 0, 2, 4, 6 \}
  76543210

• **Operations**
  - &  Intersection
  - $01000001$   \{ 0, 6 \}
  - |  Union
  - $01111101$   \{ 0, 2, 3, 4, 5, 6 \}
Representing & Manipulating Sets

• Representation
  • A $w$-bit vector represents subsets of $\{0, \ldots, w-1\}$
  • $a_j = 1$ iff $j \in A$

\[
\begin{array}{c}
01101001 & \{ 0, 3, 5, 6 \} \\
76543210
\end{array}
\]

\[
\begin{array}{c}
01010101 & \{ 0, 2, 4, 6 \} \\
76543210
\end{array}
\]

• Operations
  • $\&$ Intersection
  • $|$ Union
  • $^\wedge$ Symmetric difference

\[
\begin{array}{c}
01000001 & \{ 0, 6 \} \\
01111101 & \{ 0, 2, 3, 4, 5, 6 \} \\
00111100 & \{ 2, 3, 4, 5 \}
\end{array}
\]
Representing & Manipulating Sets

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

  \[
  01101001 \quad \{0, 3, 5, 6\} \\
  76543210
  \]

  \[
  01010101 \quad \{0, 2, 4, 6\} \\
  76543210
  \]

• **Operations**
  • \& Intersection  \[
  01000001 \quad \{0, 6\}
  \]
  • | Union  \[
  01111101 \quad \{0, 2, 3, 4, 5, 6\}
  \]
  • ^ Symmetric difference  \[
  00111100 \quad \{2, 3, 4, 5\}
  \]
  • ~ Complement  \[
  10101010 \quad \{1, 3, 5, 7\}
  \]
Bit-Level Operations in C
Bit-Level Operations in C

- &
- |  
- ^
- ~
Bit-Level Operations in C

- &  |  ^  ~
- Apply to any “integral” data type
Bit-Level Operations in C

- &  |  ^  ~
- Apply to any “integral” data type
  - long, int, short, char, unsigned
Bit-Level Operations in C

- & | ^ ~
- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
Bit-Level Operations in C

- `&` | `^` | `~`
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Examples (char data type)
Bit-Level Operations in C

- &   |   ^   ~
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
- Examples (char data type)
  - ~0x41 --> 0xBE
Bit-Level Operations in C

- `&`, `|`, `^`, `~`
- Apply to any “integral” data type
  - `long`, `int`, `short`, `char`, `unsigned`
- View arguments as bit vectors
- **Examples (char data type)**
  - `~0x41 --> 0xBE`
  - `~01000001_2 --> 10111110_2`
Bit-Level Operations in C

- & | ^ ~

- Apply to any “integral” data type
  - long, int, short, char, unsigned

- View arguments as bit vectors

- **Examples (char data type)**
  - ~0x41 --> 0xBE
    - ~01000001₂ --> 10111110₂
  - ~0x00 --> 0xFF
Bit-Level Operations in C

- & | ^ ~
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
- Examples (char data type)
  - ~0x41 --> 0xBE
    ~01000001 \(_{2}\) --> 10111110 \(_{2}\)
  - ~0x00 --> 0xFF
    ~00000000 \(_{2}\) --> 11111111 \(_{2}\)
Bit-Level Operations in C

• &  |  ^  ~
  • Apply to any “integral” data type
    • long, int, short, char, unsigned
  • View arguments as bit vectors
• Examples (char data type)
  • ~0x41 --> 0xBE
    ~01000001₂ --> 10111110₂
  • ~0x00 --> 0xFF
    ~00000000₂ --> 11111111₂
  • 0x69 & 0x55 --> 0x41
Bit-Level Operations in C

- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- **Examples (char data type)**
  - ~0x41 --> 0xBE
    - ~01000001₂ --> 10111110₂
  - ~0x00 --> 0xFF
    - ~00000000₂ --> 11111111₂
  - 0x69 & 0x55 --> 0x41
    - 01101001₂ & 01010101₂ --> 01000001₂
Bit-Level Operations in C

- & | ^ ~
- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors
- Examples (char data type)
  - ~0x41 --> 0xBE
    ~01000001₂ --> 10111110₂
  - ~0x00 --> 0xFF
    ~00000000₂ --> 11111111₂
  - 0x69 & 0x55 --> 0x41
    01101001₂ & 01010101₂ → 01000001₂
  - 0x69 | 0x55 --> 0x7D
Bit-Level Operations in C

- &   |   ^   ~

- Apply to any “integral” data type
  - long, int, short, char, unsigned
- View arguments as bit vectors

- Examples (char data type)
  - \( \sim 0x41 \rightarrow 0xBE \)
    \( \sim 01000001_2 \rightarrow 10111110_2 \)
  - \( \sim 0x00 \rightarrow 0xFF \)
    \( \sim 00000000_2 \rightarrow 11111111_2 \)
  - \( 0x69 \& 0x55 \rightarrow 0x41 \)
    \( 01101001_2 \& 01010101_2 \rightarrow 01000001_2 \)
  - \( 0x69 \mid 0x55 \rightarrow 0x7D \)
    \( 01101001_2 \mid 01010101_2 \rightarrow 01111101_2 \)
Bit-Level Operations in C

• & | ^ ~
  • Apply to any “integral” data type
    • long, int, short, char, unsigned
  • View arguments as bit vectors
  • Examples (char data type)
    • ~0x41 --> 0xBE
      ~01000001₂ --> 10111110₂
    • ~0x00 --> 0xFF
      ~00000000₂ --> 11111111₂
    • 0x69 & 0x55 --> 0x41
      01101001₂ & 01010101₂ --> 01000001₂
    • 0x69 | 0x55 --> 0x7D
      01101001₂ | 01010101₂ --> 01111101₂
  • Many bit-twiddling puzzles in Lab 1
Contrast: Logic Operations in C
Contrast: Logic Operations in C

- Contrast to logical operators
Contrast: Logic Operations in C

- Contrast to logical operators
  - &&     ||     !
Contrast: Logic Operations in C

- Contrast to logical operators
  - &&     ||     !
  - 0 is “False”
Contrast: Logic Operations in C

- Contrast to logical operators
  - &&  ||  !
  - 0 is “False”
  - Anything nonzero is “True”
Contrast: Logic Operations in C

- Contrast to logical operators
  - && | | !
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
Contrast: Logic Operations in C

- Contrast to logical operators
  - `&&` | `||` | `!`
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
  - Early termination a.k.a. short-circuit evaluation
Contrast: Logic Operations in C

- Contrast to logical operators
  - &&    |   |   !
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
  - Early termination a.k.a. short-circuit evaluation

- Examples (char data type)
Contrast: Logic Operations in C

• Contrast to logical operators
  • && || !
  • 0 is “False”
  • Anything nonzero is “True”
  • Always return 0 or 1
  • Early termination a.k.a. short-circuit evaluation

• Examples (char data type)
  • !0x41 --> 0x00
Contrast: Logic Operations in C

- Contrast to logical operators
  - `&&` | `||` | `!`
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
    - Early termination a.k.a. short-circuit evaluation

- Examples (char data type)
  - `!0x41` --> `0x00`
  - `!0x00` --> `0x01`
Contrast: Logic Operations in C

- Contrast to logical operators
  - &&     |     |
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
    - Early termination  a.k.a.  short-circuit evaluation

- Examples (char data type)
  - !0x41  -->  0x00
  - !0x00  -->  0x01
  - !!0x41  -->  0x01
Contrast: Logic Operations in C

• Contrast to logical operators
  • &&     |     !
    • 0 is “False”
    • Anything nonzero is “True”
    • Always return 0 or 1
  • Early termination a.k.a. short-circuit evaluation

• Examples (char data type)
  • !0x41  -->  0x00
  • !0x00  -->  0x01
  • !!0x41  -->  0x01
  • 0x69  &&  0x55  -->  0x01
Contrast: Logic Operations in C

• Contrast to logical operators
  • &&            | |            !
  • 0 is “False”
  • Anything nonzero is “True”
  • Always return 0 or 1
  • Early termination  a.k.a.  short-circuit evaluation

• Examples (char data type)
  • !0x41  --→  0x00
  • !0x00  --→  0x01
  • !!0x41  --→  0x01
  • 0x69  &&  0x55  --→  0x01
  • 0x69  ||  0x55  --→  0x01
Contrast: Logic Operations in C

- Contrast to logical operators
  - &&   |   ||   !
  - 0 is “False”
  - Anything nonzero is “True”
  - Always return 0 or 1
  - Early termination a.k.a. short-circuit evaluation

- Examples (char data type)
  - !0x41  -->  0x00
  - !0x00  -->  0x01
  - !!0x41 -->  0x01
  - 0x69 && 0x55  -->  0x01
  - 0x69 || 0x55  -->  0x01
  - p && *p++  (avoids null pointer access, null pointer = x00000000)
Announcements

- Everyone waiting to get in should have heard by now
  - If we talked about auditing, please send me mail so I have your UWnetIDs
- Lab 0 due on Monday, still having fun?
  - Covering arrays today in lecture
- Lab 1 goes out today, by the end of class
- Stuck on something? Confused at the end of class?
  - Post on the discussion board
  - Come by office hours (mine or the TAs: posted online)
  - Send mail to cse351-staff@cse.uw.edu
- Readings for each lecture are posted on the website
- Slides for each lecture are posted ~1 day ahead of time
  - A 1-page summary of concepts and definitions will go up after class
  Not everything you need to know but most of the important topics.