Bitwise and Logical Operators
(Common to C and Java)

Brief Detour:
Input and Output in C

printf and scanf
Input and Output (section 4.3)

- `printf()`

  Conversion Specifiers

  - `%c`
  - `%d`
  - `%u`
  - `%f`
  - `%s`
  - `%o`
  - `%x`
  - `%X`
  - `%%`

- `scanf()`

  - The order of filling in values comes from the list of variables.
  - Format strings should match declared types.

.printf Example

```c
int age = 12;
int shoesize = 5;
char width = 'N';
printf("I wear a size %d%c shoe. I am %d years old.\n",
    shoesize, width, age);
```
scanf Example

```c
float temp;
printf("Enter the temperature:\n");
scanf("%f", &temp);
```

- Format string should match declared type.
- There **MUST** be an ampersand before the name of the variable being scanned.

You Try It

```c
#include <stdio.h>
int main()
{
  int number;
  char letter;
  float money;
  /* Read an integer into number */ _____________________________
  /* Read a character into letter */ _____________________________
  /* Read a float into money */ _____________________________
  /* Print the value of each variable, with a message */
  ____________________________________
  ____________________________________
  ____________________________________
}
```
printf Format Modifiers

• A number is a field width. Example:

    printf("6d", 4);
    prints ___ spaces before the 4

• The dot '.' followed by a number is "precision". Example:

    printf("%6.3f", 2.8)
    prints 2.800 (with ___ spaces before the 2)
    Note that that "6" includes the 3 decimal places and the '.'

• 0 (the digit zero) means pad with zeroes to field width. Ex:

    printf("%06d", 71);
    prints ___ zeroes before the 71

Bitwise Operators
(Section 4.4.8)
Used in Networking

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version (4 bit)</td>
<td></td>
</tr>
<tr>
<td>Traffic Class (4 bit)</td>
<td></td>
</tr>
<tr>
<td>Flow Label (24 bit)</td>
<td></td>
</tr>
<tr>
<td>Payload Length (16 bit)</td>
<td></td>
</tr>
<tr>
<td>Next Header (8 bit)</td>
<td></td>
</tr>
<tr>
<td>Hop Limit (8 bit)</td>
<td></td>
</tr>
</tbody>
</table>

- Source Address (128 bit)
- Destination Address (128 bit)

Used in Encryption/Decryption
**Used in Compression**

![Compression Diagram](image)

**Bitwise Operators**

**Table 4.13 Boolean Operators for Variables $x_1$ and $x_2$**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND</td>
<td>$x_1 &amp; x_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Exclusive-OR</td>
<td>$x_1 \oplus x_2 = (x_1 \cdot x_2') + (x_1'^* x_2)$</td>
</tr>
<tr>
<td>~</td>
<td>NOT (1's complement)</td>
<td>$\neg x_1$</td>
</tr>
</tbody>
</table>
Assume that \( x \) is an integer variable.

- Determine the least significant bit of \( x \):
  \[
  \text{int bit} = x \ & \ _____;
  \]

- Set the least significant bit of \( x \) to 1:
  \[
  x = x \ | \ _____;
  \]
Bitwise Operator: **NOT**

- **NOT** (~) is the same as one’s complement
  - Turns 0 to 1, and 1 to 0

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

- **Examples:**
  - Assume that x is an integer on 8 bits. Set x = 1.
    What is the value of ~x?

  - Set the least significant bit of y to 0:
    \[ y = y \& _____; \]

Bitwise Operator: **XOR**

- **XOR** (^)
  - 0 if both bits are the same
  - 1 if the two bits are different

<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>

- For an integer x, what is the value of

\[ x \^ x \]
Bitwise Operator: Shift Left

- Shift left (<<): Multiply by powers of 2
  - Shift some # of bits to the left, filling the blanks with 0
  
  53 \[0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\]
  \[53<<2\ \ \ \ \ \ \ \ \ \]

Bitwise Operator: Shift Right

- Shift right (>>): Divide by powers of 2
  - Shift some # of bits to the right.
  - Fill in blanks with sign bit.

  53 \[0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\]  -75 \[1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\]
  \[53>>2\ \ \ \ \ \ \ \ \ \]  \[-75>>2\ \ \ \ \ \ \ \ \]

  sign extension  
  sign extension

Note: Java also has the unsigned shift operator >>>
Not available in C.
Bitwise Operators

• Print all the bits of a character from right to left (starting with the least significant bit):

```c
char c = 0xB5;          \[0|1|1|0|1|0|1\]
void ReversePrintBits(char c)
{
    /* add code here */
}
```

Bitwise Operators

• Print all the bits of a character from left to right (starting with the most significant bit):

```c
char c = 0xB5;          \[0|0|1|1|0|1|0|1\]
void PrintBits(char c)
{
    /* add code here */
}
```
Bitmasks

• Used to change or query one or more bits in a variable.

• The bitmask indicates which bits are to be affected.

• Common operations:
  – Set one or more bits (set to 1)
  – Clear one or more bits (set to zero)
  – Read one or more bits

• Examples:
  – Set bit 2 of x (bit 0 is least significant): \( x = x \ldots \)
  – Clear bit 3 of x: \( x = x \ldots \)
  – Read bit 4 of x: \( \text{bit} = x \ldots \)

Bitmasks

• Set the least significant byte of x to FF:
  \( x = x \ldots \)

• Clear the least significant byte of x:
  \( x = x \ldots \)

• Read the least significant byte of x:
  \( \text{byte} = x \ldots \)
Bitmasks

• Set to 1 bits 2, 4 and 7 of x (0 is least significant):

\[ x = x \text{ \_\_\_\_\_\_} \;
\]

• Clear bits 3, 4 and 5 of x:

\[ x = x \text{ \_\_\_\_\_\_} \;
\]

Logical Operators
(Section 4.4.5)
Familiar Operators (common to C, Java)

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
</table>
| Arithmetic    | ++expr  --expr  expr++  expr--  
expr1*expr2  expr1/expr2  expr1%expr2  
expr1+expr2  expr1-expr2 |
| Assignment    | expr1=expr2  
expr1*=expr2  expr1/=expr2  expr1%=expr2  
expr1+=expr2  expr1-=expr2 |
| Relational    | expr1<expr2  expr1<=expr2  expr1>expr2  
expr1>=expr2  expr1==expr2  expr1!=expr2 |
| Logical       | !expr  expr1&expr2  expr1|expr2 |
| Function Call | func(paramlist) |
| Cast          | (type)expr |
| Conditional   | expr1?expr2:expr3 |

Review: Bit-Level Operators

& (AND), | (OR), ~ (NOT), ^ (XOR), << (LSHIFT), >> (RSHIFT)

• Examples (char data type)

- ~0x41 --> ______
- ~0x00 --> ______
- 0x69 & 0x55 --> ______
- 0x69 | 0x55 --> ______
- 0xBE << 3 --> ______
- 0xBE >> 3 --> ______
Contrast: Logical Operators

• Always return 0 or 1
• View 0 as “False”
• Anything nonzero as “True”
• Examples (char data type)
  - !0x41  -->  _____
  - !0x00  -->  _____
  - !!0x41 -->  _____
  - 0x69 && 0x55  -->  _____
  - 0x69 || 0x55  -->  _____

What is the Output?

```c
int a = 0x43, b = 0x21;
printf("a | b = \%x\n", a | b);   __________
printf("a || b = \%x\n", a || b);  __________
printf("a & b = \%x\n", a & b);   __________
printf("a && b = \%x\n", a && b); __________
```
Mental Exercise

/*
 * isNotEqual - return 0 if x == y, and 1 otherwise
 * Examples: isNotEqual(5,5) = 0, isNotEqual(4,5) = 1
 */

int isNotEqual (int x, int y)
{
    return ______________;
}

How much Memory Space for Data?
Storage Units

1 bit = smallest unit of memory

1 byte = 8 bits

4 bytes = 1 word
(system dependent)

Words

- On most machines, bytes are assembled into larger structures called “words”, where a word is usually defined to be the number of bits the processor can operate on at one time.

- Some machines use four-byte words (32 bits), while some others use 8-byte words (64 bits) and some machines use less conventional sizes.
The **sizeof** Operator

<table>
<thead>
<tr>
<th>Category</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>sizeof</td>
<td>sizeof(type)</td>
</tr>
<tr>
<td></td>
<td>sizeof(expr)</td>
</tr>
</tbody>
</table>

- Unique among operators: evaluated at compile-time
- Evaluates to type `size_t`; on tanner, same as `unsigned int`
- Examples

```c
int i = 10;
double d = 100.0;
...
... sizeof(int) ... /* On tanner, evaluates to 4 */
... sizeof(i) ... /* On tanner, evaluates to 4 */
... sizeof(double)... /* On tanner, evaluates to 8 */
... sizeof(d) ... /* On tanner, evaluates to 8 */
... sizeof(d + 200.0) ... /* On tanner, evaluates to 8 */
```

Determining Data Sizes

- To determine data sizes on your computer

```c
#include <stdio.h>

int main()
{
    printf("char:        \%d\n", (int)sizeof(char));
    printf("short:       \%d\n", (int)sizeof(short));
    printf("int:         \%d\n", (int)sizeof(int));
    printf("long:        \%d\n", (int)sizeof(long));
    printf("float:       \%d\n", ____________);
    printf("double:      \%d\n", ____________);
    printf("long double: \%d\n", ____________);
    return 0;
}
```

- Output on tanner

```
char: 1
short: 2
int: 4
long: 4
float: 4
double: 8
long double: 16
```
Overflow: Running Out of Room

• Adding two large integers together
  – Sum might be too large to store in available bits
  – What happens?

• Unsigned integers
  – All arithmetic is “modulo” arithmetic
  – Sum would just wrap around

• Signed integers
  – Can get nonsense values
  – Example with 16-bit integers
    o Sum: 10000+20000+30000
    o Result: -5536

Try It Out

• Write a program that computes the sum
  10000+20000+30000

Use only short int variables in your code:

```
short int a = 10000;
short int b = 20000;
short int c = 20000;
short int sum = a + b + c;

printf("sum = %d\n", sum);
```
Exercise

• Assume only four bits are available for representing integers, and signed integers are represented in 2’s complement.

• Compute the value of the expression $7 + 7$

Data Type Conversions
Operators: C vs. Java (cont.)

- Java: demotions are not automatic
- C: demotions are automatic

```c
int i;
char c;
...
i = c;         /* Implicit promotion */
               /* OK in Java and C */
c = i;         /* Implicit demotion */
               /* Java: Compiletime error */
               /* C: OK; truncation */
c = (char)i;   /* Explicit demotion */
               /* Java: OK; truncation */
               /* C: OK; truncation */
```

Int to Char? Try It Out ...

```c
#include <stdio.h>
int main()
{
    char c;
    int i;

    i = 1000;
    c = i;

    printf(" integer = %d
 character code = %d\n", i, c);
    return 0;
}
```

Change %d to %6d and see what happens ...
Change %d to %06d and see what happens ...
Exercise: Binary Output

1. Write a function `PrintCharBin` takes as argument a character and prints out its binary representation.

   Example: `PrintCharBin(10)` should print out `00001010`

2. Write a function `PrintIntBin` that takes as argument an integer and prints out its binary representation. (Should work on any machine.)

   Example: `PrintIntBin(1025)` should print out `00000000 00000000 00000100 00000001`

3. Use your functions to print out the binary representations of `c` and `i` from the previous Int-to-Char example.

Exercise: Hex Output

1. Write a function `PrintCharHex` takes as argument a character and prints out its hexadecimal representation.

   Example: `PrintCharHex(10)` should print out `0x0A`

   (Use the format “%02x” to print out a value on 2 hexadecimal digits.)

2. Write a function `PrintIntHex` that takes as argument an integer and prints out its hexadecimal binary representation.

   Example: `PrintIntHex(1025)` should print out `0x00000401`

3. Use your functions to print out the hexadecimal representations of `c` and `i` from the previous Int-to-Char example.
Signed vs. Unsigned in C

• C code:

```c
char a = 0xFF;
unsigned char b = 0xFF;
```

• Fixed number of bits in memory (8 bits)

• What is the difference?

• Try

```c
printf("a = %d\n", a);
printf("b = %d\n", b);
```

Float to Int? Try It Out ...

```c
#include <stdio.h>
int main()
{
    int i;
    float f;

    f = 3.1415926;
    i = f;

    printf(" float = \n integer = %d", f, i);
    return 0;
}
```

Change %f to %.2f and see what happens ...
Change %f to %06.2f and see what happens ...
Summary

• Computer represents everything in binary
  – Integers, floating-point numbers, characters, addresses, …
  – Pixels, sounds, colors, etc.

• Bitwise operations in C (and Java)
  – AND &, OR |, NOT ~, XOR ^, shift left << and shift right >>

• Logical operations in C (and Java)
  – AND &&, OR ||, NOT !

• How much space for data?
  – sizeof

Required Reading

• Cavanagh Textbook, Chapters 1 and 4