Integral Data Types in C

Professor Jennifer Rexford
http://www.cs.princeton.edu/~jrex

Goals for this Lecture

- Binary number system
  - Why binary?
  - Converting between decimal and binary
  - … and octal and hexadecimal number systems
- Finite representations of binary integers
  - Unsigned and signed integers
  - Integer addition and subtraction
- Bitwise operators
  - AND, OR, NOT, and XOR
  - Shift-left and shift-right
- The C integral data types
  - char, short, int, long
  - signed and unsigned variants
Why Bits (Binary Digits)?

- Computers are built using digital circuits
  - Inputs and outputs can have only two values
  - True (high voltage) or false (low voltage)
  - Represented as 1 and 0
- Can represent many kinds of information
  - Boolean (true or false)
  - Numbers (23, 79, …)
  - Characters (‘a’, ‘z’, …)
  - Pixels, sounds
  - Internet addresses
- Can manipulate in many ways
  - Read and write
  - Logical operations
  - Arithmetic

Base 10 and Base 2

- Decimal (base 10)
  - Each digit represents a power of 10
  - \( 4173 = 4 \times 10^3 + 1 \times 10^2 + 7 \times 10^1 + 3 \times 10^0 \)
- Binary (base 2)
  - Each bit represents a power of 2
  - \( 10110 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22 \)

Decimal to binary conversion:
Divide repeatedly by 2 and keep remainders

\[ \begin{align*}
12/2 & = 6 \quad R = 0 \\
6/2 & = 3 \quad R = 0 \\
3/2 & = 1 \quad R = 1 \\
1/2 & = 0 \quad R = 1 \\
\end{align*} \]

Result = 1100
Writing Bits is Tedious for People

- **Octal (base 8)**
  - Digits 0, 1, ..., 7

- **Hexadecimal (base 16)**
  - Digits 0, 1, ..., 9, A, B, C, D, E, F

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

Thus the 16-bit binary number `1010 0010 1010 1001` converted to hex is `B2A9`.

Representing Colors: RGB

- **Three primary colors**
  - Red
  - Green
  - Blue

- **Strength**
  - 8-bit number for each color (e.g., two hex digits)
  - So, 24 bits to specify a color

- **In HTML, on the course Web page**
  - **Red**: `<font color="#FF0000">Symbol Table Assignment Due</i>`
  - **Blue**: `<font color="#0000FF">Spring Break</i>`

- **Same thing in digital cameras**
  - Each pixel is a mixture of red, green, and blue
Finite Representation of Integers

- Fixed number of bits in memory
  - Usually 8, 16, or 32 bits
  - (1, 2, or 4 bytes)
- Unsigned integer
  - No sign bit
  - Always 0 or a positive number
  - All arithmetic is modulo $2^n$
- Examples of unsigned integers
  - 00000001 $\rightarrow$ 1
  - 00001111 $\rightarrow$ 15
  - 00010000 $\rightarrow$ 16
  - 00100001 $\rightarrow$ 33
  - 11111111 $\rightarrow$ 255

Adding Two Integers

- From right to left, we add each pair of digits
- We write the sum, and add the carry to the next column
Binary Sums and Carries

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Sum</th>
<th>a</th>
<th>b</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

XOR ("exclusive OR")

0100 0101 69
+ 0110 0111 103
1010 1100 172

Modulo Arithmetic

- Consider only numbers in a range
  - E.g., five-digit car odometer: 0, 1, …, 99999
  - E.g., eight-bit numbers 0, 1, …, 255

- Roll-over when you run out of space
  - E.g., car odometer goes from 99999 to 0, 1, …
  - E.g., eight-bit number goes from 255 to 0, 1, …

- Adding $2^n$ doesn’t change the answer
  - For eight-bit number, $n=8$ and $2^n=256$
  - E.g., $(37 + 256) \mod 256$ is simply 37

- This can help us do subtraction…
  - Suppose you want to compute $a - b$
  - Note that this equals $a + (256 - 1 - b) + 1$
One’s and Two’s Complement

- One’s complement: flip every bit
  - E.g., b is 01000101 (i.e., 69 in decimal)
  - One’s complement is 10111010
  - That’s simply 255-69

- Subtracting from 11111111 is easy (no carry needed!)

\[
\begin{array}{c}
11111111 \\
- \quad 01000101 \\
\hline
10111010
\end{array}
\]

- Two’s complement
  - Add 1 to the one’s complement
  - E.g., (255 – 69) + 1 \(\Rightarrow\) 1011 1011

Putting it All Together

- Computing “a – b”
  - Same as “a + 256 – b”
  - Same as “a + (255 – b) + 1”
  - Same as “a + onesComplement(b) + 1”
  - Same as “a + twosComplement(b)”

- Example: 172 – 69
  - The original number 69: 0100 0101
  - One’s complement of 69: 1011 1010
  - Two’s complement of 69: 1011 1011
  - Add to the number 172: 1010 1100
  - The sum comes to: 0110 0111
  - Equals: 103 in decimal

\[
\begin{array}{c}
10101100 \\
+ \quad 10111011 \\
\hline
101100111
\end{array}
\]
Signed Integers

- Sign-magnitude representation
  - Use one bit to store the sign
    - Zero for positive number
    - One for negative number
  - Examples
    - E.g., 0010 1100 \(\rightarrow\) 44
    - E.g., 1010 1100 \(\rightarrow\) -44
  - Hard to do arithmetic this way, so it is rarely used

- Complement representation
  - One’s complement
    - Flip every bit
    - E.g., 1101 0011 \(\rightarrow\) -44
  - Two’s complement
    - Flip every bit, then add 1
    - E.g., 1101 0100 \(\rightarrow\) -44

Overflow: Running Out of Room

- Adding two large integers together
  - Sum might be too large to store in the number of bits available
  - 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
    - Sum: 10000+20000+30000
    - Result: -5536
Bitwise Operators: AND and OR

- Bitwise AND (&)
  \[
  \begin{array}{c|c|c}
    & 0 & 1 \\ 
    0 & 0 & 0 \\ 
    1 & 0 & 1 \\
  \end{array}
  \]

- Bitwise OR (|)
  \[
  \begin{array}{c|c|c}
    & 0 & 1 \\ 
    0 & 0 & 1 \\ 
    1 & 1 & 1 \\
  \end{array}
  \]

- Mod on the cheap!
  - E.g., 53 % 16
  - ... is same as 53 & 15;

53
  \[
  \begin{array}{c|c|c|c|c|c}
    & 0 & 0 & 1 & 1 & 0 & 1 \\
  \end{array}
  \]

& 15
  \[
  \begin{array}{c|c|c|c|c|c}
    & 0 & 0 & 0 & 1 & 1 & 1 \\
  \end{array}
  \]

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

Bitwise Operators: Not and XOR

- One’s complement (~)
  - Turns 0 to 1, and 1 to 0
  - E.g., set last three bits to 0
    - \( x = x & \sim 7; \)

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

\[
\begin{array}{c|c}
  ^ & 0 & 1 \\ 
  0 & 0 & 1 \\ 
  1 & 1 & 0 \\
\end{array}
\]
Bitwise Operators: Shift Left/Right

- 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  1 1 0 1 0 0 0 0

- Shift right (>>): Divide by powers of 2
  - Shift some # of bits to the right
    - For unsigned integer, fill in blanks with 0
    - What about signed negative integers? Varies across machines…
      - Can vary from one machine to another!

  53  0 0 1 1 0 1 0 1
  53>>2  0 0 0 1 1 0 1

Example: Counting the 1’s

- How many 1 bits in a number?
  - E.g., how many 1 bits in the binary representation of 53?

  0 0 1 1 0 1 0 1

  - Four 1 bits

- How to count them?
  - Look at one bit at a time
  - Check if that bit is a 1
  - Increment counter

- How to look at one bit at a time?
  - Look at the last bit: n & 1
  - Check if it is a 1: (n & 1) == 1, or simply (n & 1)
Counting the Number of ‘1’ Bits

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    unsigned n, count;

    printf("Number: ");
    if (scanf("%u", &n) != 1) {
        fprintf(stderr, "Error: Expect number.\n");
        exit(EXIT_FAILURE);
    }

    for (count=0; n; n >>= 1)
        count += (n & 1);

    printf("Number of 1 bits: %u\n", count);
    return 0;
}
```

Data Types

- Programming languages combine:
  - Bits into bytes
  - Bytes into larger entities

- Combinations of bytes have types; why?
  - Facilitates abstraction
  - Enables compiler to do type checking

- C has 11 primitive data types
  - 8 integral data types (described in this lecture)
    - Four different sizes (char, short, int, and long)
    - Signed vs. unsigned
  - 3 floating-point data types (described in next lecture)
C Integral Data Types

• Why char vs. short vs. int vs. long?
  • Small sizes conserve memory
  • Large sizes provide more range

• Why signed vs. unsigned?
  • Signed types allow negatives
  • Unsigned types allow larger positive numbers
  • (Dubious value: Java omits unsigned types)

• When to use unsigned?
  • When you really need that extra bit
  • When you’ll do lots of bit shifting
  • When you’ll never do \( a < 0 \) test

C Integral Data Types (continued)

• Integral types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Typically Used to Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed char</td>
<td>1</td>
<td>The numeric code of a character</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1</td>
<td>The numeric code of a character</td>
</tr>
<tr>
<td>(signed) short</td>
<td>2*</td>
<td>A small integer</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2*</td>
<td>A small non-negative integer</td>
</tr>
<tr>
<td>(signed) int</td>
<td>4*</td>
<td>An integer</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4*</td>
<td>A non-negative integer</td>
</tr>
<tr>
<td>(signed) long</td>
<td>4*</td>
<td>An integer</td>
</tr>
<tr>
<td>unsigned long</td>
<td>4*</td>
<td>A non-negative integer</td>
</tr>
</tbody>
</table>

* On hats; C90 standard does not specify size
The **int** Data Type

- **Description:** A positive or negative integer
  - Same as signed int

- **Size:** System dependent
  - \(16 \leq \text{bits in short} \leq \text{bits in int} \leq \text{bits in long}\)
  - Usually 16 bits (alias 2 bytes) or 32 bits (alias 4 bytes)
  - The "natural word size" of the computer

The **int** Data Type (cont.)

- **Example constants (assuming 4 bytes)**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Binary Representation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>00000000 00000000 00000000 01111011</td>
<td>decimal form</td>
</tr>
<tr>
<td>-123</td>
<td>11111111 11111111 11111111 10000110</td>
<td>negative form</td>
</tr>
<tr>
<td>2147483647</td>
<td>01111111 11111111 11111111 11111111</td>
<td>largest</td>
</tr>
<tr>
<td>-2147483648</td>
<td>10000000 00000000 00000000 00000000</td>
<td>smallest</td>
</tr>
<tr>
<td>123</td>
<td>00000000 00000000 00000000 01111011</td>
<td>octal form</td>
</tr>
<tr>
<td>-123</td>
<td>11111111 11111111 11111111 10000110</td>
<td>hexadecimal form</td>
</tr>
</tbody>
</table>

- **Leading zero means octal**
- **Leading zero-x means hexadecimal**
- **High-order bit indicates sign**
- **Two's complement**
The **unsigned int** Data Type

- **Description:** A positive integer
- **Size:** System dependent
  - Same as int
- **Example constants (assuming 4 bytes)**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Binary Representation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>12U</td>
<td>00000000 00000000 00000000 01111011</td>
<td>decimal form</td>
</tr>
<tr>
<td>4294967295U</td>
<td>11111111 11111111 11111111 11111111</td>
<td>largest</td>
</tr>
<tr>
<td>0</td>
<td>00000000 00000000 00000000 00000000</td>
<td>smallest</td>
</tr>
<tr>
<td>0173U</td>
<td>00000000 00000000 00000000 01111011</td>
<td>octal form</td>
</tr>
<tr>
<td>0x78U</td>
<td>00000000 00000000 00000000 01111011</td>
<td>hexadecimal form</td>
</tr>
</tbody>
</table>

Note “U” suffix

Same range as int, but shifted on number line

---

The **long** Data Type

- **Description:** A positive or negative integer
  - Same as signed long
- **Size:** System dependent
  - 16 <= bits in short <= bits in int <= bits in long
  - Usually 32 bits, alias 4 bytes
- **Example constants (assuming 4 bytes)**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Binary Representation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>12L</td>
<td>00000000 00000000 00000000 01111011</td>
<td>decimal form</td>
</tr>
<tr>
<td>-12L</td>
<td>11111111 11111111 11111111 10000101</td>
<td>negative form</td>
</tr>
<tr>
<td>2147483647L</td>
<td>01111111 11111111 11111111 11111111</td>
<td>largest</td>
</tr>
<tr>
<td>-2147483648L</td>
<td>10000000 00000000 00000000 00000000</td>
<td>smallest</td>
</tr>
<tr>
<td>0173L</td>
<td>00000000 00000000 00000000 01111011</td>
<td>octal form</td>
</tr>
<tr>
<td>0x78L</td>
<td>00000000 00000000 00000000 01111011</td>
<td>hexadecimal form</td>
</tr>
</tbody>
</table>

Note “L” suffix
The **unsigned long** Data Type

- **Description:** A positive integer
- **Size:** System dependent
  - Same as `long`
- **Example constants (assuming 4 bytes)**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Binary Representation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 (UL)</td>
<td>00000000 00000000 00000000 01111011</td>
<td>decimal form</td>
</tr>
<tr>
<td>4294967295UL</td>
<td>11111111 11111111 11111111 11111111</td>
<td>largest</td>
</tr>
<tr>
<td>0UL</td>
<td>00000000 00000000 00000000 00000000</td>
<td>smallest</td>
</tr>
<tr>
<td>0x7BUL</td>
<td>00000000 00000000 00000000 01111011</td>
<td>hexadecimal form</td>
</tr>
</tbody>
</table>

Note "UL" suffix

The **short** Data Type

- **Description:** A positive or negative integer
  - Same as `signed short`
- **Size:** System dependent
  - $16 \leq$ bits in `short` $\leq$ bits in `int` $\leq$ bits in `long`
  - Usually 16 bits, alias 2 bytes
- **Example constants (assuming 2 bytes)**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Binary Representation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>(short)12</td>
<td>00000000 01111011</td>
<td>decimal form</td>
</tr>
<tr>
<td>(short)-12</td>
<td>11111111 10000101</td>
<td>negative form</td>
</tr>
<tr>
<td>(short)32767</td>
<td>01111111 11111111</td>
<td>largest</td>
</tr>
<tr>
<td>(short)-32768</td>
<td>10000000 00000000</td>
<td>smallest</td>
</tr>
<tr>
<td>(short)0173</td>
<td>00000000 01111011</td>
<td>octal form</td>
</tr>
<tr>
<td>(short)0x1B</td>
<td>00000000 01111011</td>
<td>hexadecimal form</td>
</tr>
</tbody>
</table>

No way to express constant of type short, so must use cast
The **unsigned short** Data Type

- **Description:** A positive integer
- **Size:** System dependent
  - Same as short
- **Example constants (assuming 4 bytes)**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Binary Representation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>(unsigned short) 123U</td>
<td>00000000 01111011</td>
<td>decimal form</td>
</tr>
<tr>
<td>(unsigned short) 65535U</td>
<td>11111111 11111111</td>
<td>largest</td>
</tr>
<tr>
<td>(unsigned short) 0U</td>
<td>00000000 00000000</td>
<td>smallest</td>
</tr>
<tr>
<td>(unsigned short) 0173U</td>
<td>00000000 01111011</td>
<td>octal form</td>
</tr>
<tr>
<td>(unsigned short) 0x78U</td>
<td>00000000 01111011</td>
<td>hexadecimal form</td>
</tr>
</tbody>
</table>

No way to express constant of type unsigned short, so must use cast

---

The **signed char** Data Type

- **Description:** A (small) positive or negative integer
- **Size:** 1 byte
- **Example constants**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Binary Representation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>(signed char) 123</td>
<td>01111011</td>
<td>decimal form</td>
</tr>
<tr>
<td>(signed char) -123</td>
<td>10000101</td>
<td>negative form</td>
</tr>
<tr>
<td>(signed char) 12</td>
<td>01111111</td>
<td>largest</td>
</tr>
<tr>
<td>(signed char) -128</td>
<td>10000000</td>
<td>smallest</td>
</tr>
<tr>
<td>(signed char) 0173</td>
<td>01111011</td>
<td>octal form</td>
</tr>
<tr>
<td>(signed char) 0x78</td>
<td>01111011</td>
<td>hexadecimal form</td>
</tr>
</tbody>
</table>

No way to express constant of type signed char, so must use cast
The unsigned char Data Type

- Description: A (small) positive integer
- Size: 1 byte
- Example constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Binary Representation</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned char 123</td>
<td>01111011</td>
<td>decimal form</td>
</tr>
<tr>
<td>(unsigned char) 255</td>
<td>11111111</td>
<td>largest</td>
</tr>
<tr>
<td>(unsigned char) 0</td>
<td>00000000</td>
<td>smallest</td>
</tr>
<tr>
<td>(unsigned char) 0177</td>
<td>01111011</td>
<td>octal form</td>
</tr>
<tr>
<td>(unsigned char) 0x7B</td>
<td>01111011</td>
<td>hexadecimal form</td>
</tr>
</tbody>
</table>

No way to express constant of type unsigned char, so must use cast

The char Data Type

- On some systems, char means signed char
- On other systems, char means unsigned char
- Obstacle to portability

```c
int a[256];
char c;
c = (char)255;
...
... a[c] ...
/* char is unsigned => a[255] => OK */
/* char is signed => a[-1] => out of bounds */
```
The char Data Type (cont.)

• On your system, is char signed or unsigned?

```c
#include <stdio.h>
int main(void) {
    char c = (char)0x80;
    if (c > 0)
        printf("unsigned");
    else
        printf("signed");
    return 0;
}
```

• Output on hats

```
signed
```
Summary

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

- Binary arithmetic through logic operations
  - Sum (XOR) and Carry (AND)
  - Two’s complement for subtraction

- Binary operations in C
  - AND, OR, NOT, and XOR
  - Shift left and shift right
  - Useful for efficient and concise code, though sometimes cryptic

- C integral data types
  - char, short, int, long (signed and unsigned)

The Rest of the Week

- Reading
  - Required: *C Programming*: 4, 5, 6, 7, 14, 15, and 20.1
  - Recommended: *Computer Systems*: 2
  - Recommended: *Programming with GNU Software*: 3, 6

- Monday office hours
  - My office hours by appointment, instead of usual 4:30pm

- Wednesday’s lecture
  - C Fundamentals

- Programming assignment
  - A “Decomment” Program
  - Due Sunday at 9pm