CS 107
Lecture 3: Integer Representations

Monday, October 2, 2017

Computer Systems
Fall 2017
Stanford University
Computer Science Department

Reading: Chapter 2.2

Lecturers: Julie Zelenski and Chris Gregg

4-bit two's complement signed integer representation
Today's Topics

- Logistics
  - Assign0 — Due at midnight
  - Labs start tomorrow
  - Office hours in full coverage
- Reading: Chapter 2.2 (very mathy…)
- Integer Representations
  - Unsigned recap.
  - Signed numbers
    - two's complement
  - Signed vs Unsigned numbers
  - Casting in C
  - Signed and unsigned comparisons
- The `sizeof` operator
  - Min and Max integer values
  - Truncating integers
  - two's complement overflow
Integer Representations
The C language has two different ways to represent numbers, unsigned and signed:

**unsigned**: can only represent non-negative numbers

**signed**: can represent negative, zero, and positive numbers

We are going to talk about these representations, and also about what happens when we expand or shrink an encoded integer to fit into a different type (e.g., `int` to `long`).
So far, we have talked about converting from decimal to binary and vice-versa, which is a nice one-to-one relationship between the decimal number and its binary representation. Examples:

0b0001 = 1
0b0101 = 5
0b1011 = 11
0b1111 = 15

The range of an unsigned number is $0 \rightarrow 2^w - 1$, where $w$ is the number of bits in our integer. For example, a 32-bit int can represent numbers from 0 to $2^{32} - 1$, or 0 to 4,294,967,295.
Signed Integers: How do we represent them?

What if we want to represent negative numbers? We have choices!

One way we could encode a negative number is simply to designate some bit as a "sign" bit, and then interpret the rest of the number as a regular binary number and then apply the sign. For instance, for a four-bit number:

0 001 = 1  1 001 = -1
0 010 = 2  1 010 = -2
0 011 = 3  1 011 = -3
0 100 = 4  1 100 = -4
0 101 = 5  1 101 = -5
0 110 = 6  1 110 = -6
0 111 = 7  1 111 = -7

This might be okay...but we've only represented 14 of our 16 available numbers...
<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>1's Complement</th>
<th>2's Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 001</td>
<td>1</td>
<td>1 001</td>
<td>-1</td>
</tr>
<tr>
<td>0 010</td>
<td>2</td>
<td>1 010</td>
<td>-2</td>
</tr>
<tr>
<td>0 011</td>
<td>3</td>
<td>1 011</td>
<td>-3</td>
</tr>
<tr>
<td>0 100</td>
<td>4</td>
<td>1 100</td>
<td>-4</td>
</tr>
<tr>
<td>0 101</td>
<td>5</td>
<td>1 101</td>
<td>-5</td>
</tr>
<tr>
<td>0 110</td>
<td>6</td>
<td>1 110</td>
<td>-6</td>
</tr>
<tr>
<td>0 111</td>
<td>7</td>
<td>1 111</td>
<td>-7</td>
</tr>
</tbody>
</table>

What about 0 000 and 1 000? What should they represent?

Well...this is a bit tricky!
### Signed Integers: How do we represent them?

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 001</td>
<td>1</td>
<td>1 001</td>
<td>-1</td>
</tr>
<tr>
<td>0 010</td>
<td>2</td>
<td>1 010</td>
<td>-2</td>
</tr>
<tr>
<td>0 011</td>
<td>3</td>
<td>1 011</td>
<td>-3</td>
</tr>
<tr>
<td>0 100</td>
<td>4</td>
<td>1 100</td>
<td>-4</td>
</tr>
<tr>
<td>0 101</td>
<td>5</td>
<td>1 101</td>
<td>-5</td>
</tr>
<tr>
<td>0 110</td>
<td>6</td>
<td>1 110</td>
<td>-6</td>
</tr>
<tr>
<td>0 111</td>
<td>7</td>
<td>1 111</td>
<td>-7</td>
</tr>
</tbody>
</table>

What about 0 000 and 1 000? What should they represent? Well...this is a bit tricky!

Let's look at the bit patterns: 0 000 1 000

Should we make the 0 000 just represent decimal 0? What about 1 000? We could make it 0 as well, or maybe -8, or maybe even 8, but none of the choices are nice.
Signed Integers: How do we represent them?

0 001 = 1  1 001 = -1  What about 0 000 and 1 000? What should
0 010 = 2  1 010 = -2  they represent?
0 011 = 3  1 011 = -3
0 100 = 4  1 100 = -4  Well...this is a bit tricky!
0 101 = 5  1 101 = -5
0 110 = 6  1 110 = -6
0 111 = 7  1 111 = -7

Let's look at the bit patterns: 0 000  1 000

Should we make the 0 000 just represent decimal 0? What about 1 000? We
could make it 0 as well, or maybe -8, or maybe even 8, but none of the choices
are nice.

Fine. Let's just make 0 000 to be equal to decimal 0. How does arithmetic work?
Well...to add two numbers, you need to know the sign, then you might have to
subtract (borrow and carry, etc.), and the sign might change...this is going to get
ugly!
Signed Integers: How do we represent them?

There is a better way!
Signed Integers: How do we represent them?

Behold: the "two's complement" circle:

In the early days of computing*, two's complement was determined to be an excellent way to store binary numbers.

In two's complement notation, positive numbers are represented as themselves (phew), and negative numbers are represented as the two's complement of themselves (definition to follow).

This leads to some amazing arithmetic properties!

*John von Neumann suggested it in 1945, for the EDVAC computer.
Two's Complement

A two's-complement number system encodes positive and negative numbers in a binary number representation. The weight of each bit is a power of two, except for the most significant bit, whose weight is the negative of the corresponding power of two.

Definition: For vector \( \vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0] \) of an \( w \)-bit integer \( x_{w-1}x_{w-2} \ldots x_0 \) is given by the following formula:

\[
B2T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i.
\]

\( B2T_w \) means "Binary to Two's complement function"

In practice, a negative number in two's complement is obtained by inverting all the bits of its positive counterpart*, and then adding 1.

*Inverting all the bits of a number is its "one's complement"
Two's Complement

In practice, a negative number in two's complement is obtained by inverting all the bits of its positive counterpart*, and then adding 1, or: $x = \neg x + 1$

Example: The number 2 is represented as normal in binary: 0010

-2 is represented by inverting the bits, and adding 1:

\[
\begin{array}{c}
0010 \\
\downarrow \\
1101 \\
+ 1 \\
1110
\end{array}
\]

*Inverting all the bits of a number is its "one's complement"
Two's Complement

Trick: to convert a positive number to its negative in two's complement, start from the right of the number, and write down all the digits until you get to a 1. Then invert the rest of the digits:

Example: The number 2 is represented as normal in binary: 0010

Going from the right, write down numbers until you get to a 1:

10

Then invert the rest of the digits:

1110

*Inverting all the bits of a number is its "one's complement"
To convert a negative number to a positive number, perform the same steps!

Example: The number -5 is represented in two's complements as: 1011

5 is represented by inverting the bits, and adding 1:

\[
1011 \rightarrow 0100
\]

\[
0100
\]

\[
+ 1
\]

\[
0101
\]

Shortcut: start from the right, and write down numbers until you get to a 1:

1

Now invert all the rest of the digits:

0101
There are a number of useful properties associated with two's complement numbers:

1. There is only one zero (yay!)
2. The highest order bit (left-most) is 1 for negative, 0 for positive (so it is easy to tell if a number is negative)
3. Adding two numbers is just…adding!
   Example:
   \[ 2 + (-5) = -3 \]
   
   \[
   \begin{array}{c}
   \text{0010} \\
   \text{0101} \\
   \hline
   \text{1101}
   \end{array}
   \]
   2 + 1101 = -3 decimal (wow!)
Two's Complement: Neat Properties

More useful properties:

4. Subtracting two numbers is simply performing the two's complement on one of them and then adding.
   Example:
   \[ 4 - 5 = -1 \]

   Find the two's complement of 5: 1011
   add:
   \[
   \begin{array}{c}
   0100 \\
   +1011 \\
   \hline
   1111
   \end{array}
   \]
   \(-1\) decimal

0100 \(\rightarrow\) 4, 0101 \(\rightarrow\) 5

Find the two's complement of 5: 1011
add:
\[
\begin{array}{c}
0100 \\
+1011 \\
\hline
1111
\end{array}
\]
\(-1\) decimal
More useful properties:

5. Multiplication of two's complement works just by multiplying (throw away overflow digits).

Example: \(-2 \times -3 = 6\)

\[
\begin{array}{c}
1110 \times 1101 \\
0000 \\
1110 \times 0110 \\
+1110 \\
\hline
10110110
\end{array}
\]

\(-2 \times -3 = 6\)
Two's Complement: Powers of two remain!

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ of an $w$-bit integer $x_{w-1}x_{w-2}\ldots x_0$ is given by the following formula:

$$B2T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i.$$  

From the definition of a two's complement number, we can see that we are still dealing with bits being equal to their powers-of-two place: there isn't anything magical about the placement of the bits:

$$-5 = 1 \quad 0 \quad 1 \quad 1 \quad (1 \cdot -2^3) + (0 \cdot 2^2) + (1 \cdot 2^1) + (1 \cdot 2^0)$$
Convert the following 4-bit numbers from positive to negative, or from negative to positive using two's complement notation:

a. -4 (1100)

b. 7 (0111)

c. 3 (0011)

d. -8 (1000)
Practice

Convert the following 4-bit numbers from positive to negative, or from negative to positive using two's complement notation:

a.  -4 (1100)  0100

b.  7 (0111)  1001

c.  3 (0011)  1101

d.  -8 (1000)  1000 (?! If you look at the chart, +8 cannot be represented in two's complement with 4 bits!)
Practice

Convert the following 8-bit numbers from positive to negative, or from negative to positive using two's complement notation:

a. \(-4 \ (11111100) \rightarrow 00000100\)

b. \(27 \ (00011011) \rightarrow 11100101\)

c. \(-127 \ (10000001) \rightarrow 01111111\)

d. \(1 \ (00000001) \rightarrow 11111111\)
Converting between two numbers in C can happen explicitly (using a parenthesized cast), or implicitly (without a cast):

**explicit**

```c
1 int tx, ty;
2 unsigned ux, uy;
3 ... 
4 tx = (int) ux;
5 uy = (unsigned) ty;
```

**implicit**

```c
1 int tx, ty;
2 unsigned ux, uy;
3 ... 
4 tx = ux; // cast to signed
5 uy = ty; // cast to unsigned
```

When casting: **the underlying bits do not change**, so there isn't any conversion going on, except that the variable is treated as the type that it is. You cannot convert a signed number to its unsigned counterpart using a cast!
When casting: **the underlying bits do not change**, so there isn't any conversion going on, except that the variable is treated as the type that it is. You cannot convert a signed number to its unsigned counterpart using a cast!

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

int main() {
    int v = -12345;
    unsigned int uv = (unsigned int) v;
    printf("v = %d, uv = %u\n",v,uv);
    return 0;
}
```

```
$ ./test_cast
v = -12345, uv = 4294954951
```
Casting Between Signed and Unsigned

printf has three 32-bit integer representations:

%d : signed 32-bit int
%u : unsigned 32-bit int
%x : hex 32-bit int

As long as the value is a 32-bit type, printf will treat it according to the formatter it is applying:

```c
int x = -1;
unsigned u = 3000000000; // 3 billion
printf("x = %u = %d\n", x, x);
printf("u = %u = %d\n", u, u);
```

$ ./test_printf
x = 4294967295 = -1
u = 3000000000 = -1294967296
Signed vs Unsigned Number Wheels

4-bit two's complement signed integer representation

4-bit unsigned integer representation
Comparison between signed and unsigned values

When a C expression has combinations of signed and unsigned variables, you need to be careful!

If an operation is performed that has both a signed and an unsigned value, **C implicitly casts the signed argument to unsigned** and performs the operation assuming both numbers are non-negative. Let's take a look…

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647 &gt; -2147483647 - 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647U &gt; -2147483647 - 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison between signed and unsigned values

When a C expression has combinations of signed and unsigned variables, you need to be careful!

If an operation is performed that has both a signed and an unsigned value, C implicitly casts the signed argument to unsigned and performs the operation assuming both numbers are non-negative. Let's take a look…

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 == 0U</td>
<td>Unsigned</td>
<td>1</td>
</tr>
<tr>
<td>-1 &lt; 0</td>
<td>Signed</td>
<td>1</td>
</tr>
<tr>
<td>-1 &lt; 0U</td>
<td>Unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647 &gt; -2147483647 - 1</td>
<td>Signed</td>
<td>1</td>
</tr>
<tr>
<td>2147483647U &gt; -2147483647 - 1</td>
<td>Unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647 &gt; (int)2147483648U</td>
<td>Signed</td>
<td>1</td>
</tr>
<tr>
<td>-1 &gt; -2</td>
<td>Signed</td>
<td>1</td>
</tr>
<tr>
<td>(unsigned)-1 &gt; -2</td>
<td>Unsigned</td>
<td>1</td>
</tr>
</tbody>
</table>
Comparison between signed and unsigned values

Let's try some more...a bit more abstractly.

```c
int s1, s2, s3, s4;
unsigned int u1, u2, u3, u4;
```

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*

- `s3 > u3`
- `u2 > u4`
- `s2 > s4`
- `s1 > s2`
- `u1 > u2`
- `s1 > u3`
Comparison between signed and unsigned values

Let's try some more...a bit more abstractly.

```c
int s1, s2, s3, s4;
unsigned int u1, u2, u3, u4;
```

Which many of the following statements are true? *(assume that variables are set to values that place them in the spots shown)*

- `s3 > u3` : true
- `u2 > u4` : true
- `s2 > s4` : false
- `s1 > s2` : true
- `u1 > u2` : true
- `s1 > u3` : true
As we have seen, integer types are limited by the number of bits they hold. On the 64-bit myth machines, we can use the `sizeof` operator to find how many bytes each type uses:

```c
int main() {
    printf("sizeof(char): %d\n", (int) sizeof(char));
    printf("sizeof(short): %d\n", (int) sizeof(short));
    printf("sizeof(int): %d\n", (int) sizeof(int));
    printf("sizeof(unsigned int): %d\n", (int) sizeof(unsigned int));
    printf("sizeof(long): %d\n", (int) sizeof(long));
    printf("sizeof(long long): %d\n", (int) sizeof(long long));
    printf("sizeof(size_t): %d\n", (int) sizeof(size_t));
    printf("sizeof(void *): %d\n", (int) sizeof(void *));
    return 0;
}
```

```
sizeof(char): 1
sizeof(short): 2
sizeof(int): 4
sizeof(unsigned int): 4
sizeof(long): 8
sizeof(long long): 8
sizeof(size_t): 8
sizeof(void *): 8
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Width in bytes</th>
<th>Width in bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>size_t</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>void *</td>
<td>8</td>
<td>64</td>
</tr>
</tbody>
</table>
Because we now know how bit patterns for integers works, we can figure out the maximum and minimum values, designated by `INT_MAX, UINT_MAX, INT_MIN, (etc.),` which are defined in `limits.h`.

<table>
<thead>
<tr>
<th>Type</th>
<th>Width (bytes)</th>
<th>Width (bits)</th>
<th>Min in hex (name)</th>
<th>Max in hex (name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>8</td>
<td>80 (CHAR_MIN)</td>
<td>7F (CHAR_MAX)</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>FF (UCHAR_MAX)</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>16</td>
<td>8000 (SHRT_MIN)</td>
<td>7FFF (SHRT_MAX)</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2</td>
<td>16</td>
<td>0</td>
<td>FFFF (USHRT_MAX)</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>32</td>
<td>80000000 (INT_MIN)</td>
<td>7FFFFFFFF (INT_MAX)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4</td>
<td>32</td>
<td>0</td>
<td>FFFFFFFF (UINT_MAX)</td>
</tr>
<tr>
<td>long</td>
<td>8</td>
<td>64</td>
<td>8000000000000000 (LONG_MIN)</td>
<td>7FFFFFFFFFFFFFFFF (LONG_MAX)</td>
</tr>
<tr>
<td>unsigned long</td>
<td>8</td>
<td>64</td>
<td>0</td>
<td>FFFFFFFFFFFFFFFFF (ULONG_MAX)</td>
</tr>
</tbody>
</table>
Sometimes we want to convert between two integers having different sizes. E.g., a `short` to an `int`, or an `int` to a `long`.

We might not be able to convert from a bigger data type to a smaller data type, but we do want to always be able to convert from a smaller data type to a bigger data type.

This is easy for unsigned values: simply add leading zeros to the representation (called "zero extension").

```c
unsigned short s = 4;
// short is a 16-bit format, so s = 0000 0000 0000 0100b

unsigned int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```
Expanding the bit representation of a number

For signed values, we want the number to remain the same, just with more bits. In this case, we perform a "sign extension" by repeating the sign of the value for the new digits. E.g.,

```java
short s = 4;
// short is a 16-bit format, so s = 0000 0000 0000 0100b
int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b

— or —

short s = -4;
// short is a 16-bit format, so s = 1111 1111 1111 1100b
int i = s;
// conversion to 32-bit int, so i = 1111 1111 1111 1111 1111 1111 1111 1100b
```
// show_bytes() defined on pg. 45, Bryant and O'Halloran
int main() {
    short sx = -12345;       // -12345
    unsigned short usx = sx; // 53191
    int x = sx;              // -12345
    unsigned ux = usx;       // 53191

    printf("sx = %d:\t", sx);
    show_bytes((byte_pointer) &sx, sizeof(short));
    printf("usx = %u:\t", usx);
    show_bytes((byte_pointer) &usx, sizeof(unsigned short));
    printf("x  = %d:\t", x);
    show_bytes((byte_pointer) &x, sizeof(int));
    printf("ux = %u:\t", ux);
    show_bytes((byte_pointer) &ux, sizeof(unsigned));

    return 0;
}

$ ./sign_extension
sx = -12345:    c7 cf
usx = 53191:    c7 cf
x  = -12345:    c7 cf ff ff
ux = 53191:    c7 cf 00 00

(careful: this was printed on the little-endian myth machines!)
Recall that the right-shift (») operator behaves differently for unsigned and signed numbers:

- **Unsigned** numbers are **logically**-right shifted (by shifting in 0s, always)

- **Signed** numbers are **arithmetically**-right shifted (by shifting in the sign bit)

```c
// show_bytes() defined on pg. 45, Bryant and O'Halloran
int main() {
    int a = 1048576;
    int a_rs8 = a >> 8;
    int b = -1048576;
    int b_rs8 = b >> 8;

    printf("a = %d:\t", a);
    show_bytes((byte_pointer) &a, sizeof(int));

    printf("a >> 8 = %d:\t", a_rs8);
    show_bytes((byte_pointer) &a_rs8, sizeof(int));

    printf("b = %d:\t", b);
    show_bytes((byte_pointer) &b, sizeof(int));

    printf("b >> 8 = %d:\t", b_rs8);
    show_bytes((byte_pointer) &b_rs8, sizeof(int));

    return 0;
}
```

$ ./right_shift
a = 1048576: 00 00 10 00
a >> 8 = 4096: 00 10 00 00
b = -1048576: 00 00 f0 ff
b >> 8 = -4096: 00 f0 ff ff

(run on a little-endian machine)
Truncating Numbers: Signed

What if we want to reduce the number of bits that a number holds? E.g.

```java
int x = 53191;
short sx = (short) x;
int y = sx;
```

What happens here? Let's look at the bits in `x` (a 32-bit `int`), 53191:

```
0000 0000 0000 0000 1100 1111 1100 0111
```

When we cast `x` to a short, it only has 16-bits, and C truncates the number:

```
1100 1111 1100 0111
```

What is this number in decimal? Well, it must be negative (b/c of the initial 1), and it is \(-12345\).
Truncating Numbers: Signed

What if we want to reduce the number of bits that a number holds? E.g.

```java
int x = 53191;        // 53191
short sx = (short) x; // -12345
int y = sx;
```

This is a form of overflow! We have altered the value of the number. Be careful!

We don't have enough bits to store the int in the short for the value we have in the int, so the strange values occur.

What is y above? We are converting a short to an int, so we sign-extend, and we get -12345!

```
1100 1111 1100 0111 becomes
1111 1111 1111 1111 1100 1111 1100 0111
```

Play around here: http://www.convertforfree.com/twos-complement-calculator/
Truncating Numbers: Signed

If the number does fit into the smaller representation in the current form, it will convert just fine.

```java
int x = -3;        // -3
short sx = (short) -3; // -3
int y = sx; // -3
```

```
x: 1111 1111 1111 1111 1111 1111 1111 1101 becomes
sx: 1111 1111 1111 1111 1101
```

Truncating Numbers: Unsigned

We can also lose information with unsigned numbers:

```c
unsigned int x = 128000;
unsigned short sx = (short) x;
unsigned int y = sx;
```

Bit representation for \( x = 128000 \) (32-bit unsigned int):

```
0000 0000 0000 0001 1111 0100 0000 0000
```

Truncated unsigned short \( sx \):

```
1111 0100 0000 0000
```

which equals 62464 decimal.

Converting back to an unsigned int, \( y = 62464 \)
Overflow in Unsigned Addition

When operations overflow in C, the runtime does not produce an error:

```c
#include<stdio.h>
#include<stdlib.h>
#include<limits.h> // for UINT_MAX

int main() {
    unsigned int a = UINT_MAX;
    unsigned int b = 1;
    unsigned int c = a + b;

    printf("a = %u\n",a);
    printf("b = %u\n",b);
    printf("a + b = %u\n",c);
    return 0;
}
```

```
$ ./unsigned_overflow
a = 4294967295
b = 1
a + b = 0
```

Technically, unsigned integers in C don't overflow, they just wrap. You need to be aware of the size of your numbers. Here is one way to test if an addition will fail:

```c
#include <limits.h>
unsigned int a = <something>;
unsigned int x = <something>;
if (a > UINT_MAX - x) /* `a + x` would overflow */;
```
Signed overflow wraps around to the negative numbers:

YouTube fell into this trap — their view counter was a signed, 32-bit int. They fixed it after it was noticed, but for a while, the view count for Gangnam Style (the first video with over INT_MAX number of views) was negative.
Signed overflow wraps around to the negative numbers.

```c
#include<stdio.h>
#include<stdlib.h>
#include<limits.h> // for INT_MAX

int main() {  
    int a = INT_MAX;  
    int b = 1;  
    int c = a + b;  
    printf("a = %d\n",a);  
    printf("b = %d\n",b);  
    printf("a + b = %d\n",c);  
    return 0;  }
```

Technically, unsigned integers in C produce undefined behavior. On two's complement machines (virtually all machines these days), it does overflow predictably. You can test to see if your addition will be correct:

```c
#include <limits.h>
int a = <something>;  
int x = <something>;  
if ((x > 0) && (a > INT_MAX - x)) /* `a + x` would overflow */;  
if ((x < 0) && (a < INT_MIN - x)) /* `a + x` would underflow */;
```
References and Advanced Reading

• **References:**
  - Wikipedia on Two's complement: [https://en.wikipedia.org/wiki/Two%27s_complement](https://en.wikipedia.org/wiki/Two%27s_complement)

• **Advanced Reading:**
4-bit two's complement signed integer representation
4-bit unsigned integer representation