

CSC 411

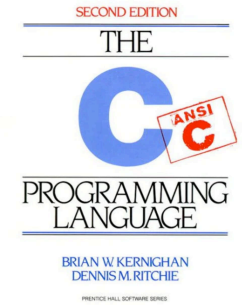
Computer Organization (Fall 2024)

Lecture 4: Integers (signed, unsigned)

Prof. Marco Alvarez, University of Rhode Island











The C Language

- Developed by Dennis Ritchie at Bell Labs in the early 1970s
- Many operating systems, including Unix and its variants (Linux), are written in C
- Allows low-level access to memory, making it efficient for system programming
- C programs are generally portable across different platforms with minimal modification
- C follows a traditional compilation process, where the source code is translated into machine code by a compiler



TIOBE Index for January 2024

- Indicator of the popularity of programming languages
 - popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings.

Jan 2024	Jan 2023	Change	Programming Language	Ratings	Change
1	1		 Python	13.97%	-2.39%
2	2		 C	11.44%	-4.81%
3	3		 C++	9.96%	-2.95%
4	4		 Java	7.87%	-4.34%
5	5		 C#	7.16%	+1.43%
6	7	▲	 JavaScript	2.77%	-0.11%
7	10	▲	 PHP	1.79%	+0.40%
8	6	▼	 Visual Basic	1.60%	-3.04%
9	8	▼	 SQL	1.46%	-1.04%
10	20	▲	 Scratch	1.44%	+0.86%

<https://www.tiobe.com/tiobe-index/>

Representing data

Representing data

- In memory, **all values** are stored as “bit-vectors”
 - data types** are used to interpret the bits (provide meaning)
 - each possible bit-vector assigned exclusively to one meaning
- In a bit sequence of w bits, we can represent 2^w different values
 - number of permutations with repetition — given w digits, there are two ways to choose each digit
 - example: how many different sequences can be represented in 4 bits?

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

$$2^4 = 16$$

What is this program doing?

```
#include <stdio.h>

int main() {
    unsigned hex = 0x7e313134;

    float *f = (float *) &hex;
    char *s = (char *) &hex;

    printf("%u %d %.4ef %c%c%c%c",
           hex, hex, *f, s[0], s[1], s[2], s[3]);

    return 0;
}
```

2117153076 2117153076 5.8882e+37f 411~

Interlude: addition and multiplication

Binary addition

	1	1	1				
	0	0	1	1	1	0	0
+	0	1	1	1	0	1	1
	1	0	1	0	1	1	1

	1				1	1	1
	1	1	0	0	0	1	1
+	0	1	0	1	0	1	1
	1	0	0	0	1	1	1

Binary multiplication

				1	0	0	1		
				0	1	1	0		
				0	0	0	0		
			1	0	0	1			
		1	0	0	1				
	0	0	0	0					
0	1	1	0	1	1	0			

				1	1	1	1		
				1	1	1	1		
				1	1	1	1		
			1	1	1	1			
		1	1	1	1				
	1	1	1	0	0	0	0	1	

Tricky? perform the addition row-by-row

Integer Representation (unsigned, signed)

Unsigned integers

- Bits represent the number directly
 - same as binary-to-decimal conversion
 - w bits can represent 2^w unsigned integers ranging from 0 to $2^w - 1$

0 0 0 0 = 0	1 0 0 0 = 8
0 0 0 1 = 1	1 0 0 1 = 9
0 0 1 0 = 2	1 0 1 0 = 10
0 0 1 1 = 3	1 0 1 1 = 11
0 1 0 0 = 4	1 1 0 0 = 12
0 1 0 1 = 5	1 1 0 1 = 13
0 1 1 0 = 6	1 1 1 0 = 14
0 1 1 1 = 7	1 1 1 1 = 15

Overflow

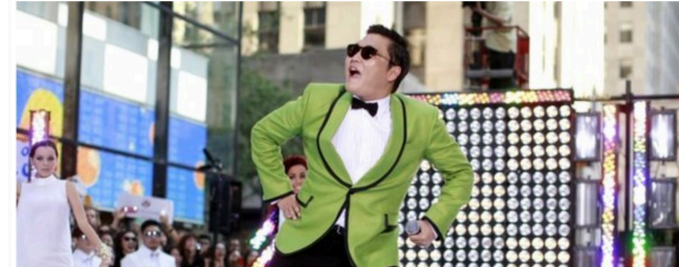
- Assume $w = 8$ is the bit-width
 - what happens if you try to add 1 to 11111111? => **Overflow**
 - there's no room for a number larger than 255 !
- The arithmetic "wraps around" back to the beginning of the range
 - adding 1 to 255 in an 8-bit system results in 0 (leading bit is discarded)
 - this wrapping around behavior can be useful in certain situations
 - applies to unsigned integers
 - basically taking the result $\bmod 2^w$ or taking the lowest w bits

Overflow

- Occurs when the result of an operation is **too large or too small** to be represented within the allocated data type
 - in C, the runtime does not produce errors, values just “**wrap**”
- Can have consequences if not handled properly
 - incorrect calculations
 - program crashes due to unexpected behavior
 - security vulnerabilities
- To prevent overflow
 - choose appropriate data types with sufficient range
 - implement checks and validations within the code

Gangnam Style music video 'broke' YouTube view limit

4 December 2014



YouTube said the video - its most watched ever - has been viewed more than **2,147,483,647** times. It has now changed the maximum view limit to **9,223,372,036,854,775,808**, or more than nine quintillion.

Zero-Day Alert: Google Chrome Under Active Attack, Exploiting New Vulnerability

Nov 29, 2023 Newsroom



Google has rolled out security updates to fix seven security issues in its Chrome browser, including a zero-day that has come under active exploitation in the wild.

Tracked as CVE-2023-6345, the high-severity vulnerability has been described as an integer overflow bug in Skia, an open source 2D graphics library.

A zero-day attack takes place when hackers exploit the flaw before developers have a chance to address it.

Signed integers

- Trivial approach (not used)
 - use MSB as the sign bit, and the remaining bits to represent the number
 - all possibilities using $w = 3$ bits:

0 0 0 = 0
0 0 1 = 1
0 1 0 = 2
0 1 1 = 3
1 0 0 = -0
1 0 1 = -1
1 1 0 = -2
1 1 1 = -3

how is zero represented?
would addition still work?
(try adding 001 and 110)

Going from x to $-x$ (and vice-versa)

▸ One's complement of x :

- flip all bits in x
- called complement because $x + -x = 11\dots11$

▸ Two's complement of x

- flip all bits in x and add 1 to the result
- $\sim x + 1$ (apply bitwise)

Using $w = 4$ bits

Binary	Unsigned	One's complement	Two's complement
0000	0	+0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	-7	-8
1001	9	-6	-7
1010	10	-5	-6
1011	11	-4	-5
1100	12	-3	-4
1101	13	-2	-3
1110	14	-1	-2
1111	15	-0	-1

Practice

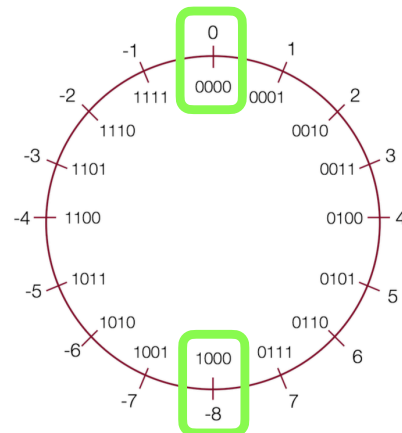
▸ Convert from x to $-x$ using two's complement

0 0 1 1 =

1 1 1 1 =

1 0 0 0 =

1 1 0 1 =



The most negative number

▸ Exceptions to $\sim x + 1$

- zero becomes zero (overflow)
- the most negative number does not have a positive counterpart — impossible to represent (overflow)

▸ Can lead to unexpected programming bugs

- the unary negation operator, $-(-128)$ becomes -128
- the absolute value may return a negative number, $\text{abs}(-128)$ becomes -128
- multiplication by -1 may fail, $-128 * -1$ becomes -128
- division by -1 may cause an exception, $-128 / -1$ crashes

▸ In C the above behaviors are undefined

From two's complement to decimal

$$\sum_{i=0}^{w-1} x_i 2^i \quad -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

Conversion to decimal for unsigned representations

Conversion to decimal for two's complement representations

Although C does not mandate using two's complement, in practice, two's complement is the **most widely used** representation for signed integers in modern computer systems.

Practice

- Convert from two's complement to decimal

-32 16 8 4 2 1

0 0 1 1 1 0 =

1 1 0 0 0 1 =

1 0 0 0 0 0 =

1 1 1 1 1 1 =

1 1 1 1 1 0 =

0 0 0 0 0 0 =

0 0 0 0 0 1 =

0 1 1 1 1 1 =

Two's complement (summary)

- Sign bit (MSB)
 - 0 for nonnegative, 1 for negative
- Non-negative numbers
 - no changes from the unsigned representation
- Negative numbers
 - two's complement of their positive counterparts
 - equivalent to their one's complement plus one

Binary	Unsigned	Signed (two's complement)
01010101	85	85
10101011	171	-85

Two's complement (advantages)

- Addition and subtraction of signed integers
 - use the same hardware as their unsigned counterparts
 - no need for special/additional circuitry
- Single representation for zero
 - no negative zero
- Trivial operation for extending the sign bit
 - e.g. increasing the bit-width of a number (**casting**)
- Widespread adoption

Numeric ranges

▸ Unsigned representation

- min => 00...00 0
- max => 11...11 $2^w - 1$

▸ Signed representation (two's complement)

- min => 10...00 -2^{w-1}
- max => 01...11 $2^{w-1} - 1$

data type	binary	hexadecimal	decimal
unsigned short int (min)			
unsigned short int (max)			
short int (min)			
short int (max)			

Basic data types in C

▸ The C standard does not define the size of “integer” types, except **char**

- much safer to use **intN_t** and **uintN_t** for signed and unsigned integers of different sizes (**stdint.h**)

▸ The type of each variable tells the compiler how many bytes are necessary in memory

- necessary for translation of high level code into machine code

```
#include <stdint.h>

int main() {
    // unsigned integer using 4 bytes ==> 0x0000000C8
    // 0000 0000 0000 0000 0000 0000 1100 1000
    uint32_t myvar = 200;
    // ....
    return 0;
}
```

Range of values

Data type	Size	Format	Value range
character	8	signed	−128 to 127
		unsigned	0 to 255
integer	16	signed	−32768 to 32767
		unsigned	0 to 65535
	32	signed	−2,147,483,648 to 2,147,483,647
		unsigned	0 to 4,294,967,295
	64	signed	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
		unsigned	0 to 18,446,744,073,709,551,615