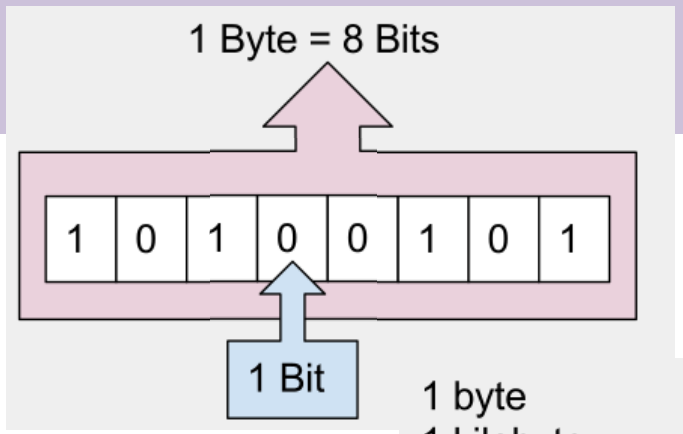
Fundamental Data Types

Outline

- Integral (Integer) Types
  - Integer range
  - Integer overflow
- Floating Point Number Types
- Type Conversion
  - Expression with mixed data types
  - Type casting

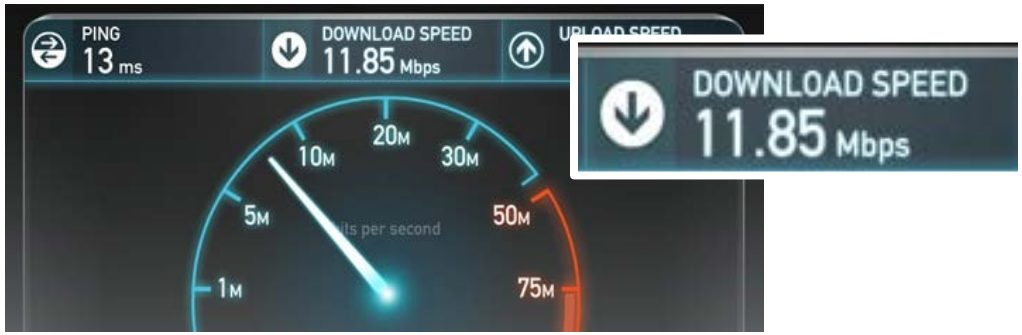
Note: will talk about strings and characters in week 6

1. Overview



- A *bit* (binary digit) is the smallest unit data.
- A bit can be either 0 or 1.
- 1 *byte* = 8 bits
- Data is represented by a sequence of bits, e.g., 1010 0101.

1 byte	= 8 bits
1 kilobyte	= 1024 bytes
1 megabyte	= 1024 kilobyte
1 gigabyte	= 1024 megabyte
1 terabyte	= 1024 gigabyte



2. Integral (Integer) Types

Key concepts

- The smallest/largest integer values (range) of type `int`
- Variation of integer types
  - Integer types of different sizes
  - Unsigned and signed integers
- Integer overflow

2.1. Integral Types (The Basics)

- Computers use N bits to represent integers.
  - Typically, N = 8, 16, 32, 64

- With N bits, computer can represent 2^N distinct values.
  - Half for negative integers, and half for non-negative integers

$$-(2^{N-1}), \ldots, -2, -1, 0, 1, \ldots, 2^{N-1}-1$$

Asymmetric!!!

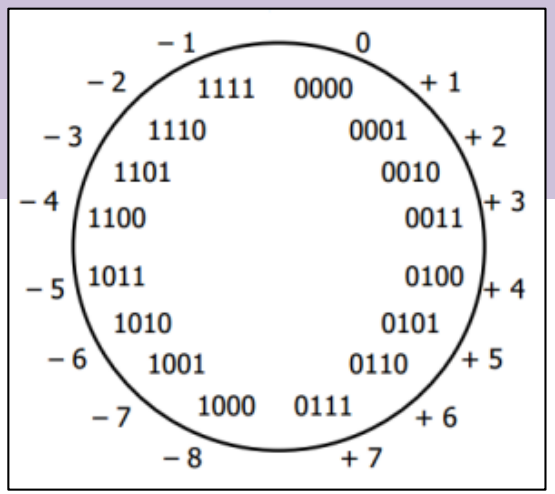
- `int` is typically 32 bits in size. As such, it can represent integers in the following range

$$-(2^{31}), -(2^{31} - 1), \ldots, -2, -1, 0, 1, 2, \ldots, 2^{31}- 1$$

or

$$-2147483648, \ldots, -2, -1, 0, 1, 2, \ldots, 2147483647$$

Optional: 2's complement



Integers are encoded or stored in computer memory like this

Sign-Magnitude

1111 1111	(-127)
1111 1110	(-126)
.....	...
.....	...
1000 0001	(-1)
1000 0000	(-0)
0111 1111	(+127)
0111 1110	(+126)
.....	...
.....	...
0000 0001	(+1)
0000 0000	(+0)

Increasing binary value

1's complement

0111 1111	(+127)
0111 1110	(+126)
.....	...
.....	...
0000 0001	(+1)
0000 0000	(+0)
1111 1111	(-0)
1111 1110	(-1)
.....	...
.....	...
1000 0001	(-126)
1000 0000	(-127)

+1

2's complement

0111 1111	(+127)
0111 1110	(+126)
.....	...
.....	...
0000 0001	(+1)
0000 0000	(+0)
1111 1111	(-1)
1111 1110	(-2)
.....	...
.....	...
1000 0001	(-127)
1000 0000	(-128)

It's a cycle!!!!

https://en.wikipedia.org/wiki/Two%27s_complement

2.2. Variations of Integral Types

Type	Size in bytes [Visual Studio]	Range
`char`	1	-128 to 127
`short` (or **`short int`**)	2	-2^{15} to $2^{15}-1$ (-32768 to 32767)
`int`	≥ 2 [4]	-2^{31} to $2^{31}-1$ (if 4 bytes)
`long` (or **`long int`**)	≥ 4 [4]	-2^{63} to $2^{63}-1$ (if 8 bytes)
`unsigned char`	1	0 to 255
`unsigned short`	2	0 to $2^{16}-1$ (0 to 65535)
`unsigned int` (or **`unsigned`**)	≥ 2 [4]	0 to $2^{32}-1$ (if 4 bytes)
`unsigned long`	≥ 4 [4]	0 to $2^{64}-1$ (if 8 bytes)

Why are there so many different types of integers?

http://www.cplusplus.com/reference/climits/
https://msdn.microsoft.com/en-us/library/7fh3a000.aspx

2.2. Variations of Integral Types

- When to use an appropriate type to represent integers in a program?
  - When the amount of data to be processed is large and the memory space is scarce
- For now it is suffice to know that these variations of integral types exist. For most applications, using `int` is adequate.

2.3. Integer Overflow

- *Integer overflow* occurs when the result of an arithmetic operation is too large to be represented by the underlying integer representation.

- e.g.: assume integers are 32 bits in size
  - Add one to the largest positive integer:
    2147483647 + 1 = -2147483648

2's complement

0111 1111	(+127)
0111 1110	(+126)
.....	...
.....	...
0000 0001	(+1)
0000 0000	(+0)
1111 1111	(-1)
1111 1110	(-2)
.....	...
.....	...
1000 0001	(-127)
1000 0000	(-128)

Hint: when you can't pass some lab. exercise in the future, this is one common reason!

- Should use **appropriate data type to avoid overflow!!!**
  `unsigned int`, `unsigned long`, …

3. Floating Point Numbers

Key Concepts

- Floating point numbers representation and arithmetic **may not be exact**.

- The smallest/largest floating point values (range) of type `double`

3.1. Floating Point Number Representation

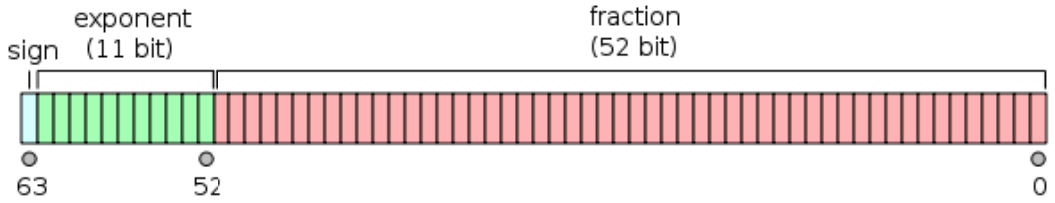
- Floating point numbers and integers have different internal representations: sign, exponent, and fraction (see next page)

Optional: floating-point format

IEEE 754 double-precision binary floating-point format: binary64 [edit]

Double-precision binary floating-point is a commonly used format on PCs, due to its wider range over single-precision floating point, in spite of its performance and bandwidth cost. As with single-precision floating-point format, it lacks precision on integer numbers when compared with an integer format of the same size. It is commonly known simply as *double*. The IEEE 754 standard specifies a **binary64** as having:

- Sign bit: 1 bit
- Exponent width: 11 bits
- Significand precision: 53 bits (52 explicitly stored)



Inputs floating point numbers:

$-241.65 = -0.24165 \times 10^{3}$

$0.0028 = 0.28 \times 10^{-2}$

$110.11 = 0.11011 \times 2^{3}$

From: http://en.wikipedia.org/wiki/Double_precision_floating-point_format

3.1. Floating Point Number Representation

- Floating point numbers and integers have different internal representations: sign, exponent, and fraction (see next page)

- Not all real numbers are representable.
  - Finite number of bits vs. infinitely many real numbers

- Floating-point number representation and arithmetic operations may not be exact.

  e.g., `printf( "%.20f %.20f" , 3.3 , 2.1 - 2.0 - 0.1 );`

  yields `3.29999999999999980000 0.00000000000000008327`

- For very large computations, rounding errors may accumulate and become significant.

3.2. C Language Floating-Point Types

Type	Size in bytes [Visual Studio]	Range
`float`	4	$\pm 3.4 \times 10^{\pm 38}$ (6 significant digits)
`double`	8	$\pm 1.7 \times 10^{\pm 308}$ (15 significant digits)
`long double`	≥ 8 [8]	

- "Significant digits" ➔ precision
- For most applications, using `double` is recommended.
- If possible, avoid using `float` which is imprecise in modern day standard, thus leading to loss of precision.

4. Type Conversion

Key Concepts

- How types are converted in an expression with mixed types of numbers:

  e.g., `2.5 + 5 / 2 = ?`

- How a `double` type value is converted to an integral type value

- Explicit Type Conversion (Type Casting)

4.1. Expressions with mixed types of data

- HK$1000 + US$100 = ?
- **`3.1 + 2`** = ?
- **`double d = 4 ;`** What value will **`d`** hold?
- **`int x = 4.1 ;`** What value will **`x`** hold?

- Some kind of conversion is needed to ensure the type of both operands are compatible before the computer can evaluate an expression, e.g., 3.1 + 2.

4.1. Implicit Type Conversion

- C language has a set of conversion rules to resolve certain mismatched operand types.

- As a convenience to programmers, compilers automatically convert the value of an operand from one type to another based on these rules whenever possible.

- Sometimes called *coercion*

4.1. Arithmetic Conversions (Simplified Rules)

- If either operand is a `double`, the other is converted to `double`. The result type is also `double`.

  e.g.:

  ```
     3.1 + 2        (3.1 is of type double)
  = 3.1 + 2.0       (Therefore, 2 is converted to 2.0)
  = 5.1             (Result is of type double)
  ```

- If both operands are of one of `char`, `short`, and `int`, then both operands are converted to `int`. The result type is also `int`.

4.2. Converting Integral Type to double

- Converting integral type to **`double`** is safe.
  - No warning is given at compile time

```
double d = 4 ;   /* 4 is converted to 4.0
                    and then 4.0 is assigned
                    to d */
```

4.2. Converting double to Integral Type

- Converting a **`double`** to an integral type may result in *loss of data*.
  - If the number is within the range of the integral type, the fractional part is **truncated**.
  - Compiler *usually* warns at compile time. (No guarantee.)

```
int x = 4.1 ;              /* 4.1 is converted to 4 */
x = 2 * 4.1 ;              /* 8.2 is converted to 8 */
x = 12345678901.2 ;        /* Behavior is hard to know
                              when the value is too
                              large (beyond range) */
```

4.3. Explicit Type Conversion (Casting)

Syntax: `(new_type) operand`

- Converts the *value of* **`operand`** to the equivalent value of type **`new_type`**.
  - **`(new_type)`** is called the type casting operator
  - Not every type conversion is possible.

e.g.,

```
double d = 4.2 ;
int y = (int) d ; // y becomes 4, no warning
int x = d ;       // x becomes 4, compiler warns
```

Type Conversion (Examples)

```
1  int    x = 5 , y = 2 ;
2  double a , b ;
3
4  a = 2.5 + x / y ;            /*    R.H.S. is evaluated as
5                                *       2.5 + (5 / 2)
6                                *   => 2.5 + 2
7                                *   => 2.5 + 2.0
8                                *   => 4.5
9                                */
10
11 b = 2.5 + (double) x / y ;  /*    R.H.S. is evaluated as
12                               *       2.5 + (5.0 / 2)
13                               *   => 2.5 + (5.0 / 2.0)
14                               *   => 2.5 + 2.5
15                               *   => 5.0
16                               */
```

Example 4.1. Expression with mixed data types

Type Conversion (Examples)

```
1  int x , y ;
2  double a = 2.6 , b = 2.4 ;
3  x = (int)( a + 0.5 );       // x is assigned 3
4  y = (int)( b + 0.5 );       // y is assigned 2
```

Example 4.2. Rounding floating point numbers to nearest integer

Will it always work?

How about negative values?

Using Type Casting Operators (Exercise)

- Average of N integers

```
// Consider the following declaration
int    total , N ;
double avg ;
…
// Suppose we have obtained the value of N and
// calculated the total of N integers.
// Which of these will correctly calculate the average?
avg = total / N ;                          // A
avg = (double)total / N ;                  // B
avg = total / (double)N ;                  // C
avg = (double)( total / N ) ;              // D
avg = (double)total / (double)N ;          // E
```

4.4. How are numbers converted?

(Apply to both implicit and explicit conversions)

- `double` to integral types
  - Only retain the integer part (no rounding)
  - e.g.:

```
int x = (int)  4.9  ;          // x gets 4
int y = (int) -3.99 ;          // y gets -3
```

- "Larger" integral types to "smaller" integral types
  - Retain only the least significant bits
  - e.g.: 32-bit integer (**`int`**) to 16-bit integer (**`short`**)

```
00000000000000100000000000000011  = 131075(10)
                0000000000000011  = 3(10)
short x = (short) 131075 ;   // x becomes 3
```

Truncated!

Summary

- All number types have a range.
  - Choose a proper data type to represent data
  - Prevent overflow
- Floating-point representation and arithmetic may not be exact.
- Expressions with mixed types of data
  - Automatic and explicit type conversion
  - Number conversion (`double` to `int`)

Appendix: Finding out the size of an integer

```
1  #include <stdio.h>
2
3  int main( void )
4  {
5      printf( "size of int   = %d\n" , sizeof(int)   );
6      printf( "size of short = %d\n" , sizeof(short) );
7      return 0 ;
8  }
```

```
size of int   = 4
size of short = 2
```

- `sizeof(`*`data_type`*`)` yields the number of bytes used to represent a value of type *`data_type`*
- You may try it for other types, e.g., *`short`*, *`float`*, etc.