# Review
# Chapter 7: Numbers
# Chapter 8: Characters
# Intro to C++ Programming

CSCI 2212 INTERMEDIATE PROGRAMMING IN C/C++

# I. Numbers

CHAPTER 7

# The Trouble with Numbers

Integer overflow and wrap

Floating point overflow

Floating point underflow

Floating point precision and approximate calculation

Scale of Magnitude and limits on precision

# Integer Overflow and wrap

If you keep making an integer larger, eventually it will wrap around to the smallest end.

Memorize these numbers:

- short -32,768. . . 32,767 (Know exact numbers)
- unsigned short: 0. . . .65,535 (know exact numbers)
- long: -2,147,483,647. . . 2,147,483,647 (know its on the order of +/- 2 billion)
- unsigned long 0. . . 4,294,967,295 (know its in the order of 4 billion)
- Today, an int us usually the same as a long.  Many years ago, it was the same as a short on some machines.

# Floating-Point Numbers

- A floating point number has a sign, an exponent, and a magnitude.
- The sign is the sign of the number.
- The exponent is a positive or negative base-2 (binary) number represented in bias notation. A float has a bias-127 exponent: if it starts with a **1 bit, it represents a positive exponent**, if it starts with **a 0 bit, the exponent is negative**.
- The magnitude is always kept **normalized**.
  - After every calculation, the result is shifted left or right, until the **first bit is a 1**.
  - The **exponent** is adjusted for each shift.
  - During normalization, 1 bits ,might be shifted **off the end of the register**, or 0 bits can be brought in.

# Floating point precision and approximate calculation

- A **float** has **24 bits of magnitude**, equivalent **to ~6 decimal digits of precision**.
- A float can **represent integers exactly, if the integer is shorter than 24 bits** and it is directly converted to type float.
- A **double** has **36 bits of magnitude**, equivalent **to ~15 decimal digits of precision**.
- A double can **represent any integer exactly if it is directly converted to type double**.
- However, most **fractional values cannot be exactly represented** because of the limited precision.
- Each time a **computation** is made, **round-off error is possible**. Over time, **it adds up**.
- A **double that is the result of computation is UNLIKELY to EVER exactly equal an integer**.

# Floating point **overflow**

When the **exponent becomes 0 or too large to represent**, the value becomes a **NaN, that is, Not a Number**. Some systems display a special character for NaN, others just show a nonsense value. Please learn what your system does and **learn to recognize NaN**.

- The exponent of **a float** is 8 bits, able to represent base-10 exponents of $\pm\mathbf{38}$
- The exponent of **a double** is 12 bits, able to represent base-10 exponents of $\pm\,\mathbf{308}$
- If the **exponent of a number grows beyond that**, the number becomes **a NaN**.
- Any arithmetic done on a NaN gives the result NaN.

# Floating point **underflow**

Suppose, during a calculation, an **exponent has been adjusted so that it is at the limit but the magnitude is STILL not normalized**.

- At that point, the limited precision of the float or double is decreased.
- Some systems will let you continue to use that unnormalized number.
- Others will not.

# Fuzzy Comparisons

- Because the **value of a float or double is usually not accurate in its last few bits**, we **never use == to compare anything to the result of a floating computation**.
- The **method of comparison** must include a **fuzz factor**.

```
if ( fabs( answer - target )< fuzz ) //continue to loop
```

- `fabs()` is floating point absolute value.
- `fuzz` **is a float or double constant such as .0001** that is appropriate for the amount of **precision** that you need.

# Scale of Magnitude and limits on precision

When two floating values are **added together**,

- If they have the same exponents, they are just added and renormalized.
- If the **exponent is different**, the smaller exponent must be adjusted by **DE-normalizing the number**. Then the magnitudes are added and the result is re-normalized.
- In the **process of de-normalization**, all of **the information in the smaller number can get shifted off the end and lost**, leaving a **0 value**.
- **This happens** when the **exponents differ by too much**.

# II. The char Data Types

CHAPTER 8

# Char is a dual-purpose type

A **char** is a dual-purpose data type:

- Representing **characters** like 'A' and 'z' and '5' and '}'.
- Representing **very small integers**. *Note: '5' != 5.*

Note that **single quotes are used for character literals**, and **double quotes for strings**. 'A' is a char and "A" is a string and "This is a string".

There are two char types: signed char, and unsigned char.

Type char is the same as one or the other, whichever is native on your hardware / OS.

# Signed and unsigned chars

Like other integer types, chars can be either signed or unsigned.

- Some operating systems use signed chars for characters, others use unsigned chars.
- Most of the time you don't know and you don't need to know.
- ASCII code can be stored in a signed char, since it is a 7-bit code, the sign bit is not important.
- International ASCII is an 8-bit code and requires unsigned char.
- Modern systems are moving to Unicode, a 16-bit character representation.

# Operations on chars

You can do several kinds of computations with **characters**.

| | |
|---|---|
| Copy them: | (ch = 'A') |
| Compare them: | if (ch == 'A’ \|\| ch =='a') |
| Add an integer: | nextLetter = ch + 1 |
| Subtract a char to get an int: | position = ch - 'A'; |
| Use as a subscript: | counter[ch]++ |
| Use in a switch: | switch( ch ) {<br>case 'A':. . . |

# Reading and Writing Characters in C

- Use " %c" to read a character. **Note the space between the quote and the percent. This skips leading whitespace**, then reads a single keystroke.
- Use "%c" to **read the next keystroke WITHOUT skipping whitespace**.
- Use "%c" to write a character.
- Use "%i" to read a small integer into a char variable. This could read a number with more than one keystroke. It does ASCII to integer conversion.
- Use "%s" to read a to read a whitespace-delimited string.

Note: **Using " %c" is the right way to eliminate whitespace at the end of a line when the next line starts with a char.**

# Two Examples

```
char ch1, ch2, response = '\n';
printf( "Please guess my secret character: " );
fflush(stdout);
scanf( "%c", &ch1 ); // Read the next keystroke.
ch2 = 'b';
printf( "Guess is %c, my secret is %c\n", ch1, ch2 );
printf( "Who was the first U.S. President?" );
printf( "W: Washington, A: Adams, J: Jefferson" );
fflush(stdout);
scanf( " %c", &response ); // Read one visible char.
if( response == 'W’ ) puts("Right; good job.");
else puts( "Wrong, try again.");
```

# Syntax in C++

IT’S A WHOLE NEW WORLD!

# Reading and Writing Characters in C++

- Use `cin >> ch;` to read a character. This **skips leading whitespace**, then reads a single keystroke.
- Use `ch = cin.get()` to **read the next keystroke WITHOUT skipping whitespace**.
- Use `cout << ch;` to **write a character**.
- Use `cin >> str;` to **read a whitespace-delimited string**.

# Two Examples in C++, this is a code fragment, not a program

```
char ch1, ch2, response = '@';
cout <<"Please guess my secret character: ";
ch1 = cin.get(); // Read the next keystroke.
cout <<"Guess is " << ch1
<<" my secret is " <<ch2 << ".\n";
cout <<"Who was the first U.S. President?\n"
<<"W: Washington, A: Adams, J: Jefferson" <<endl;
cin >> response; // Read one visible char.
if( response == 'W' ) cout <<"Right; good job.\n";
else cout <<"Wrong, try again.\n";
```

# The **ctype library**: Character Processing Functions

The **ctype library** contains functions for **processing character data**. https://cplusplus.com/reference/cctype/
These can be used either in **C or in C++:**

| | |
|---|---|
| **isalpha(ch)** | true for A. . . Z and a. . . z, false otherwise |
| **isdigit(ch)** | true for 0. . . 9, false otherwise |
| **isalnum(ch)** | same as isalpha() \|\| isdigit() |
| **isspace(ch)** | true for a whitspace character, false otherwise (space, tab, newline, CR, vertical tab or formfeed) |
| **islower(ch)** | true for a. . . z, false otherwise |
| **isupper(ch)** | true for A. . . Z, false otherwise |
| **tolower(ch)** | If ch is A. . . Z, return a. . . z. Else return ch unchanged. |
| **toupper(ch)** | If ch is a. . . z, return A. . . Z. Else return ch unchanged. |

# The End!