An Introduction to Programming
through C++

with

Manoj Prabhakaran

Lecture 6

Revision

Internal Representation of Data Types.
A Checklist of concepts encountered so far.

Based on material developed by Prof. Abhiram G. Ranade

So far

- Control flow: sequential, `if-else` conditions, loops
- Variables, types (`int`, `char`, `bool`, `...`), operators, expressions
- Bit-level representation: `bool`, `char`, `int`

Today

- Bit-level representation of data continued
  - , `float`, `double`, …
- Bit shift operation
- Examples

Reminder: Quiz 1 on Aug 21
Covers Lectures 1 through 5

Binary Representation of Fractions

- Recall binary representation of integers
  - An n-bit binary representation $b_{n-1} \ldots b_1 \; b_0$ stands for the number $b_{n-1}*2^{n-1} + \ldots + b_1*2^1 + b_0*2^0$
    - E.g., Binary `101` represents 5
- For fractions, the place values are of the form 2^{-i}
  - $b_{n-1} \ldots b_1 \, b_0 \, . \, b_{-1} \, b_{-2} \ldots b_{-m}$ stands for the number $b_{n-1}*2^{n-1} + \ldots + b_1*2^1 + b_0*2^0 + b_{-1}/2^1 + b_{-2}/2^2 + \ldots b_{-m}/2^m$
    - $= b_{n-1} \ldots b_1 \; b_0 + (b_{-1} \; b_{-2} \ldots b_{-m})/2^m$
    - E.g., Binary `101.101` represents 5 + 5/8

`float` and Friends

- `float` stands for floating point number
  - E.g. in decimal: $7.9225 \times 10^2 = 792.25$ has 5 digits of ***precision***, and its ***scale*** (given by the exponent 2) is such that is is between 100 and 999
  - E.g. in binary: $1.10001100001 \times 2^9 = 1100011000.01$ has 12 bits of precision and its scale is such that it is between 512 and 1023
  - E.g. in decimal: $1.875 \times 10^{-1} = .1875$ in binary: $1.1 \times 2^{-3} = .0011$
- By changing the exponent, the "point" floats left or right
- While representing a real number as a floating point number, will use some bits for precision, and some for scale (both signed)
  - Only finitely many real numbers have an exact representation

`float` and Friends

- `float` uses 32 bits
  - 1 bit for sign. Precision of 24 bits (23 bits stored, a leading 1 is implicit). Scale stored using 8-bits: 2^{-126} to 2^{127} (two values of the exponent are used for indicating special values).
  - Special values:
    - 0 (actually, ±0). Since implicit leading 1 won't allow representing 0.
    - Subnormal numbers (no implicit leading 1, with exponent 2^{-126})
    - ± infinity (e.g., result of dividing a non-0 number by 0)
    - "Not a Number" (NaN)
- `double` (for double precision floating point number) uses 64 bits
  - 1 bit for sign, 53 bits for precision (one implicit), 11 bits for scale.
- `long double` : may be 64, 80 or 128 bits (platform specific)

`float` and Friends: Literal Formats

- Format for floating point literals (numbers appearing in the programs) and also as used by cin/cout
  - We write <u>*num*</u> E <u>*exp*</u> (with no spaces) to mean *num* × 10^{exp}, where <u>*num*</u> can optionally have a decimal point.
  - Note: Exponent is for 10. Also, the number is in decimal.
    (There is a format allowing numbers to be specified in hexadecimal.)
  - Examples: `314E-2`, `-.01`, `1.` (E part is optional if `.` present) , `6.02214076e23` (can use E or e), `+1E+1` (+ signs are optional)
- By default, the literal is taken as a `double`. Suffix F to force `float`.

Example: Precision Issues

Demo

- Floating point arithmetic has a lot of subtleties

```
// Order of operations matters
float f = 2e7; // 20 million > 2^24
cout << 1 + f - f << endl;  // gives 0 instead of 1
cout << f - f + 1 << endl;  // gives 1 as expected

// for fractions, internal representation being binary matters
cout << 1 + 0.01F - 1 << endl;  // not equal to 0.01!
cout << 1 + 0.0078125F - 1 << endl;  // is equal to 0.0078125 !
```

Working with Real Numbers

- For the sake of better precision, use `double` instead of `float`
  - Using `double` can be a little less efficient in large applications: more memory needed, and (hence) slower

C++23 has "minifloats" too, if efficiency is more important

- When comparing, allow a "tolerance" (and be prepared for false positives)
  - E.g., instead of `a == b`, use `abs(a-b) <= epsilon`
  - E.g., instead of `a >= b`, use `(a-b) >= -epsilon`
  - The choice of the tolerance value will be application dependent!
    - Further, epsilon could be a function of a , b:
      e.g., `epsilon = max(abs(a),abs(b))*delta` (delta application dependent)

Bit Shift Operators

- Recall that the operators &, |, ^ and ~ can be used for bit-level manipulations
- Bit shift operators << and >> operate on an integral type (char, int, etc.) variable, and takes a number (how much to shift by) as an additional input
  - `(a << n )` shifts the bits in `a` by `n` positions to the left; `n` most significant bits fall off, and `n` least significant bits are set to `0`.
    - Essentially `(a << n)` is the same as `a*2*..*2` (n times) done more efficiently
  - Similarly `(a >> n )` shifts the bits in `a` by `n` positions to the right; `n` least significant bits fall off, and `n` most significant bits are set to `0` (for unsigned or non-negative a) or 1 (for negative signed a)
    - Essentially `(a >> n)` is the same as `a/(2*..*2)` (n times) if a is unsigned or non-negative; for negative a, division rounds towards 0, while >> rounds away from 0.

Example

Demo

```
int x, numbits = sizeof(int)*8;
cout << "Enter integer to be printed in binary: ";
cin >> x;
cout << x << " in binary: ";
for (int i=0; i<numbits; i++) {
   unsigned int y = (1 << (numbits-1));
   y &= x;
   cout << (y?'1':'0');
   x <<= 1;
}
cout << endl;
```

A Checklist of C++ Concepts So far

- Data types
  - `bool`, `char`, `int`, `short`, `long long`, `float`, `double`, double long, string
  - `signed` (mostly default) and `unsigned`. `const`.
- Expressions
  - Constants, variables, and using operators. Can use `( )` to enforce order of operations.
- Operators
  - Arithmetic operators: `+ - * / %`
  - Operators used for input and output: `<<, >>` (also used for bit-shift)
  - Logical operators: `&&, ||, !`
  - Ternary conditional operator: `? :`
  - Comparisons: `<, <=, >, >=, ==, !=`
  - Bitwise operators: `&, |, ^, ~`
  - (Compound) assignment operators: `=, ++, --, +=, -=, *=, /=. %=, &=, |=, ^=`
  - Comma operator: `,`
  - Casting from one type to another

A Checklist of C++ Concepts So far

- Statements
  - *Declaration (possibly with initialisation)* ;
  - *Expression* ;
  - `if` (*condition*) { *body1* } `else` { *body2* }
  - `while` (*condition*) { *body* } *and* `do` { *body* } `while` (*condition*)
  - `for` (*init*; *condition*; *update*) { *body* }
  - `break and continue`
- Some Ideas Encountered
  - Nested loops
  - Boolean algebra using logical operators (and their left-to-right evaluation in C++)
  - Chain of if-else conditions
  - Turtle drawing examples: polygon, star, inscribed squares, dashed lines, two-turtles in a box
  - Other examples: reading digits into a number, factorisation, changing case

A Checklist of C++ Concepts So far

- Some Examples of Common Errors
  - *Writing* `if(i=1)` *instead of* `if(i==1)`
  - *Off-by-one errors in loops (use small examples to check)*
  - *Not enclosing a body with multiple statements in { }*
  - *Dangling else:* `if (`*condition1*`) if (`*condition2*`) {`*body1*`} else {`*body2*`}`
    - *Results in* `else` *being attached to inner* `if`*.*
    - *When using* `if (condition1) if (condition2)` *form, always use explicit braces to be clear. (But can avoid the braces for* `else if(condition)` *)*
  - *Forgetting to initialise variables (they will have "garbage" values)*
  - *Using integer division, when floating point is intended*
  - *Invalid inputs: A good program should detect such inputs and report them to the user (and if meaningfully possible, recover from the error)*
  - *Corner cases, leading to division (or %) by zero, often reported as Floating Point Exception. (But in float, division by zero is <u>not</u> an error; it yields* `±inf`*, or* `nan for 0./0.`*)*
  - *If a long list of compiler errors, fix the first one and try again!*
  - *Also pay attention to compiler warnings.*

Exercise

- Inspect the sample program explain.cpp accompanying last lecture.

  For each part, try to find out why the program behaves the way it does.

- Simulate a projectile's trajectory, given initial x and y velocity. A sample solution is provided. Modify it to add a second projectile, and detect near collisions.

  Continue by ignoring collision; report the same collision event only once.