CMPUT 201: Practical Programming Methodology

Guohui Lin
guohui@ualberta.ca
Department of Computing Science
University of Alberta
September 2019

Lecture 3: Formatted Input/Output

Agenda:

- `printf()`
  - two most frequently used functions
  - need "`#include <stdio.h>`"
  - `printf(format string, expr1, expr2, ...);`
  - conversion specifications
- `scanf()`
  - most powerful in reading numbers
  - pattern-matching ability
  - inappropriate character push back to the input
  - has a return value
- How integers and floating-point numbers are stored

Reading:

- Textbook: Chapter 3

Formatted reading and writing:

- Warning: complete details later in the term
- Two most frequently used functions
  - `printf()` and `scanf()`
  - need "`#include <stdio.h>`"

`printf`:

- Display the contents of a string
  - a sequence of "characters/symbols" (mostly from your keyboard)

  ```
  printf(string, expr1, expr2, ...);
  ```

- **format string**:
  - with values inserted at specified points
  - one value at a time
  - constants, variables, any complicated expressions or function return values
  - every value needs a **conversion specification**, beginning with "%", which

    internal form (binary) $\rightarrow$ printed form (characters)

printf:

- For example (Page 40),

```
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void) {

    int i;
    float x;

    i = 40;
    x = 839.21f;

    printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
    printf("|%f|%10.3f|%10.3e|%-10g|\n", x, x, x, x);

    return 0;
}
```

- Output:

```
ghlin@ug10:~/CMPUT201_19F>gcc -Wall tprintf.c -o test
ghlin@ug10:~/CMPUT201_19F>./test
|40|   40|40   |  040|
|839.210022|   839.210| 8.392e+02|839.21    |
```

Conversion specifications:

- `%m.pX` or `%-m.pX` (**be fearless to try out!**)
  - `m` is an integer constant (optional): minimum field width
  - `-` for left justification
  - `p` is an integer constant (optional, if omitted, so is the period): precision
  - `X` is a letter (required)
    - `d`: based 10 integer (`p`: minimum number of digits)
    - `e`: floating-point in exponential format (`p`: number of digits after decimal point)
    - `f`: floating-point without an exponent (`p`: number of digits after decimal point)
    - `g`: floating-point in either format (`p`: maximum number of significant digits)

```
    printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);       /* i = 40;      */
    printf("|%f|%10.3f|%10.3e|%-10g|\n", x, x, x, x);  /* x = 839.21f; */

ghlin@ug10:~/CMPUT201_19F>./test
|40|   40|40   |  040|
|839.210022|   839.210| 8.392e+02|839.21    |
```

Escape sequences, beginning with "\":

- Special characters
  - reserved symbols such as `"` (using `\"`)
  - non-printing characters such as "horizontal tab" (using `\t`)
  - for now, we have

alert (bell):	`\a`
backspace:	`\b`
new line:	`\n`
horizontal tab:	`\t`
`"`:	`\"`
`\`:	`\\`
`%`	`%%`

- For example,

```
printf("\"Hello!\"");
```

- Produces:

```
"Hello!"
```

`scanf`:

- Reads input according to a particular format (into variables)

```
scanf(string, &var1, &var2, ...);
```

- **format string** — essentially the same as those used with `printf`
  - but often contains only conversion specifications, which

    printed form (characters) $\rightarrow$ internal form (binary)

- Usually, `&` precedes each variable
  - it means the "memory unit address" of the variable

    missing it $\rightarrow$ program crash!

- That is, more precisely,

```
scanf(string, memory_address1, memory_address2, ...);
```

How `scanf` works:

- Sequentially, for each conversion specification
  - locate an item of the appropriate type
  - skip blank space if necessary
  - stop at an inappropriate character (can't belong to the item)
  - **put this last character back to the input!**
- `if` (successful)
  - continue processing the format string
- `else`
  - return immediately
  - the return value is "the number of values successfully read in"
- **Good practice: always check the return value!**

Return value of scanf:

- For example,

```
/* Converts a Fahrenheit temperature to Celsius */

#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void) {

    float fahrenheit, celsius;

    for (;;) {
        printf("Enter Fahrenheit temperature (non-number to quit): ");
        if (scanf("%f", &fahrenheit) == 1) {
            celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;

            printf("Celsius equivalent: %.1f\n", celsius);
        }
        else break;
    }

    return 0;
}
```

How `scanf` recognizes an integer or a floating-point number:

- For integer:
  - (ignoring white space characters) search for a plus sign, or a minus sign, or a digit
  - then continue reading digits until reaching a non-digit
- For floating point number:
  - (ignoring white space characters) search for a plus sign, or a minus sign
  - then a series of digits possibly containing a decimal point
  - lastly a possible exponent: `e/E` + an optional sign + digits
- For example,

  ```
  scanf("%d%d%f%f", &i, &j, &x, &y);
  ```

  - input: "1 -20 .3 -4.0e3"
  - input: "1\n -20 \t.3 -4.0e3"
  - input: "1-20.3-4.0e3"

How `scanf` works — cares:

- White-space characters
  - essentially, non-symmetric matching :-P

- Other characters, for example,

```
scanf("%d/%d", &i, &j);
```

  - input: “5/␣9” (correct: / matches with /)
  - input: “5␣/9” (incorrect: ␣ mismatches with /)

- You try out

```
scanf("%d\n", &i);
```

- For example of `scanf`’s pattern-matching ability (Page 46–47)
  - math: $\frac{a}{b} + \frac{c}{d} = \frac{a \times d + c \times b}{b \times d}$
  - our C program: `a/b + c/d = (a * d + c * b)/(b * d)`

`scanf`'s pattern-matching ability:

```
/* Adds two fractions */

#include <stdio.h>

int main(void) {

    int num1, denom1, num2, denom2, result_num, result_denom;

    printf("Enter first fraction: ");
    scanf("%d/%d", &num1, &denom1);

    printf("Enter second fraction: ");
    scanf("%d/%d", &num2, &denom2);

    result_num = num1 * denom2 + num2 * denom1;
    result_denom = denom1 * denom2;
    printf("The sum is %d/%d\n", result_num, result_denom);

    return 0;
}
```

- ```
  ghlin@ug10:~/CMPUT201_19F/Week02>gcc -Wall addfrac.c -o test
  ghlin@ug10:~/CMPUT201_19F/Week02>./test
  Enter first fraction: 5/6
  Enter second fraction: 3/4
  The sum is 38/24
  ```
  ```

The integer types (Pages 125–128):

- `int` is "signed" 32-bits long binary representation
- The leftmost is the sign bit: 0 for non-negative
  - given an integer say 1,000 in decimal
  - convert it into base-2 (binary): 100 00000000 - 11000 = 11 11101000
  - left filled with 0's: 0 0000000 00000000 00000011 11101000
- The largest is $2^{31} - 1 = 2{,}147{,}483{,}647$
- Overflow?

The floating types (Pages 132–133):

- You may online "the IEEE standard 754": `https://en.wikipedia.org/wiki/IEEE_754`
- `float` is "signed" 32-bits long binary representation
- The leftmost is the sign bit: 0 for non-negative
- The next 8 bits are exponent e: 00000001 − 11111110
  - in decimal, e ranges from 1 to $2^8 - 2 = 254$, shifted to be from -126 to 127
  - the real exponent = the stored value -127
- The rest 23 bits for fraction (the binary after 1.)
  - given a floating point number say 5.25
  - convert it into base-2 (**binary exponential form**): $101.01 = 1.0101 \times 2^2$
  - exponent is 2 and fraction is 0101
  - stored exponent value is $127 + 2 = 129$, or 01111111 + 10 = 10000001
  - stored fraction (right filled with 0's): 0101000 00000000 00000000
  - as a whole: 0 10000001 0101000 00000000 00000000
- The largest is 3.40282×10^{38} with 6-digit precision (why?)

Agenda:

- Arithmetic operators
  - `+, -, *, /, %`
  - implicit/explicit type conversion
  - precedence, associativity
- Assignment operators
  - `=, +=`
  - lvalue has a memory storage
  - side effect
- Increment/decrement operators
  - `++, --`
- Evaluation
- Expression statements

Reading:

- Textbook: Chapter 4

Expressions:

- Formulas
  - how to compute a value
  - e.g., "`celsius`", "`(fahrenheit - FREEZING_PT) * SCALE_FACTOR`"
  - *recursive definitions* (operands/expressions) using operators
  - : constants, variables
- Operators
  - arithmetic: $+$, $-$, $*$, $/$, %
  - relational: $>$, $<$, $>=$, $<=$, $==$
  - logical: !, &&, $||$
  - ...

Arithmetic operators:

- Unary (requires one operand): $+$, $-$
  - does nothing :-)
  - e.g., `i = +1; /* i has value positive 1 */`
- Binary (requires two operands): $+$, $-$, $*$, $/$, % (remainder)
  - $/$: result `type`-dependent
  - e.g., `1 / 2; /* 1 / 2 has value 0 */`
  - e.g., `1.0 / 2; /* 1.0 / 2 has value 0.5 */`

  - negative operands in $/, \%$ — tricky results (fearless testing)
  - (always) check for zero-divisor
- Precedence
  - $\{\text{unary } +, -\} \prec \{*, /, \%\} \prec \{\text{binary } +, -\}$
  - use of parentheses
- Associativity
  - left: $*$, $/$, %, binary $+$, binary $-$
  - right: unary $+$, unary $-$

Assignment operators:

- Simple operator =:
  - left operand has value of the right operand
  - e.g., `v = e; /* variable v has value e */`
  - type conversion applies when `v` and `e` have different types
- Assignment is an operator
  - e.g., `int` variable `i = 72.99f; /* variable i has value 72 */`
  - the value "`i = 72.99f`" (called **side effect**) is 72 (not 72.99)
- Assignment is right associative
  - "`i = j = k = 72;`" is the same as "`i = (j = (k = 72));`"
  - by **side effect**,

    `/* assign 72 to k, then to j, then to i */`
- An lvalue (L-value) is an object stored in memory
  - left operand must be an lvalue ("`72.99 = i;`"? wrong!)

    (this logically makes sense, agree?)

Compound assignment:

- Use the old value of a variable to compute its new value:
  - e.g., `i = i + 2; /* increase the value of i by 2 */`
  - simplified as `i += 2;`
  - reads "adds 2 to `i`, storing the result in `i`"
- Operators: $+=$, $-=$, $*=$, $/=$, $\%=$
  - precedence lower than $\{$binary $+$, $-\}$
  - what does it mean by "`i *= j + k;`"
  - they are (as assignments) right associative
  - e.g., "`i += j += k;`" is the same as "`i += (j += k);`"

In-/de-crement operators:

- Very often, a variable is "incrementing"
  - i.e., adding 1; e.g.,
  - `i = i + 1;`
  - `i += 1;`
  - `i++;`
- Operators: `++` and `--`
  - postfix: `i++;` — (use `i` value then) increment `i`
  - prefix: `++i;` — increment `i` (then use `i` value)
  - at the end, both versions increment `i`
  - side effects

```
i = 1;
j = 2;
printf("i is %d\n", i);
printf("j is %d\n\n", j);
k = ++i + j++;
printf("i is %d\n", i);
printf("j is %d\n", j);
printf("k = ++i + j++ is %d\n\n", k);
```

Expression evaluation:

- Precedence:

$\{$postfix $++$, $--\} \prec \{$prefix $++$, $--$, unary $+$, $-\} \prec \{*, /, \%\} \prec \{$binary $+$, $-\} \prec \{$assignments$\}$

- Side effects
- Break down multiple in-/de-crements
- Break down chain assignments
- Use parentheses
- Undefined behaviors, e.g.

```
i = 5;
k = (j = i++) + (i = 1);
```

Expression statements:

- Any expression can be a statement
  - does not need to have an lvalue
  - e.g., `i++;`
  - e.g., `(j = i++) + (k = 5);`

    (the ending sum is calculated, then discarded)
- Useful only if side effects
  - e.g., `j * i + k + 1;`

    have no point doing so :-)

Agenda:

- Explanation on Assignment #1
  - concepts: sequence, subsequence, common subsequence
  - goal is to compute an LCS
  - technique: dynamic programming
- Assignment #1 specifications
  - read in an instance and validate
  - print to `stdout`
  - compute an LCS
- Programming with me (no sample code will be posted)

Reading:

- Textbook: Chapters 1–4

Basic concepts

- Alphabet Σ — a set of distinct symbols (called *bases, letters, characters*)
  - English alphabet
  - DNA alphabet $\{A, C, G, T\}$
  - digit alphabet $\Sigma = \{0, 1, 2, \ldots, 9\}$
- Strings over Σ (recursive definition)
  - every $x \in \Sigma$ is a string
  - for x, y being strings, xy is a string
  - ϵ is the empty string
  - Σ^* — the set of all strings over Σ
- A string is also called a sequence
- Length of a string/sequence S is the number of symbols in the string/sequence
- Subsequence (concatenate selected members in the original order)
- Common subsequences of two given sequences
  - the *Longest Common Subsequence* (LCS) problem

Longest common subsequence (LCS) problem:

- Problem description:
  - Input: two sequences X and Y over an alphabet Σ
  - Goal: an LCS T of X and Y
  - *e.g.*, $X = 123456$ and $Y = 2143563$ are two sequences, and an LCS is $T = 2356$
- Questions:
  - Is T a subsequence of both X and Y?
    $2356 = \cancel{1}23\cancel{4}56$
    $2356 = 2\cancel{1}\cancel{4}356\cancel{3}$
  - Is T the longest?
  - How is T obtained?
  - How long does it take to obtain T?

A simpler problem:

- A sequence of coins ($1, 5, 10, 25$ cents)
- For any (position-wise) consecutive/adjacent two coins, you cannot pick both
- Under this only constraint, try to pick coins to maximize your total gain
- For example, $10, 5, 5, 1, 1, 5, 1, 1, 5, 10, 25, 25, 10, \mathbf{10}$
  - ask: what is the previous best choice with the blue 10 being picked?
    say choice_1
  - also ask: what is the previous best choice with the blue 10 not picked?
    say choice_2
  - $\text{choice}_2 + 10$ is the best total with the bold **10** being picked
  - $\max\{\text{choice}_1,\ \text{choice}_2\}$ is the best total with the bold **10** not picked

	10	5	5	1	1	5	1	1	5	10	25	25	10	10
✓ total	10	5	15	11	16									
× total	0	10	10	15	15									

Optimal substructure:

- Let $X = x_1 x_2 \ldots x_n$, $Y = y_1 y_2 \ldots y_m$
  - If x_n is NOT in the LCS T (to be computed), then

    T is an LCS of $x_1 x_2 \ldots x_{n-1}$ and $y_1 y_2 \ldots y_m$ ...
  - Similarly, if y_m is NOT in the LCS T (to be computed), then

    T is an LCS of $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_{m-1}$ ...
  - If x_n and y_m are both in the LCS T (to be computed), then

    $x_n = y_m$ and it MUST be the last letter in T!

    So, we need to compute an LCS of $x_1 x_2 \ldots x_{n-1}$ and $y_1 y_2 \ldots y_{m-1}$;
    and then adding x_n to the end to form the LCS T for the original problem

- Three possibilities, we should go with the **longest** one

- Instead of recording the three LCSes, we switch to record their **lengths**

An algorithm:

- Let $DP[n, m]$ to denote the length of an LCS of $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_m$

- We have a recurrence:

$$DP[n, m] = \max \begin{cases} DP[n-1, m], \\ DP[n, m-1], \\ DP[n-1, m-1] + 1, & \text{if } x_n = y_m \end{cases}$$

  where $DP[i, j]$ denotes the length of an LCS of $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$

- Or the general recurrence is:

$$DP[i, j] = \max \begin{cases} DP[i-1, j], \\ DP[i, j-1], \\ DP[i-1, j-1] + 1, & \text{if } x_i = y_j \end{cases}$$

  for $0 < i \leq n$ and $0 < j \leq m$

- Correctness

  Proof.

Do we really have an algorithm yet?

- How do we compute $DP[n, m]$?
- Solving the recurrence:
  - Simple recursion running time: $\Omega(3^{\min\{n,m\}})$
  - Memoization: $\Theta(n \times m)$
  - Dynamic programming:

    Key idea: when computing $DP[i, j]$, the three DP entries must have been computed!

    Equivalently speaking, a careful planned tabular computation!

- *e.g.*,

		2	1	4	3	5	6	3
	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							
4	0							
5	0						$\searrow$	$\downarrow$
6	0						$\rightarrow$	?

The LCS problem — summary:

- Correctness
- Can return an associated LCS ... trace back (how?)

e.g.,

		2	1	4	3	5	6	3
	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							
4	0							
5	0							
6	0							

- Running time: $\Theta(n \times m)$

  There are $n \times m$ entries each takes constant time to compute.

- Space requirement ... $\Theta(n \times m)$

  Can be reduced to $\Theta(\min\{n, m\})$!

- Applications:
  - Bioinformatics
  - Cheating detection

Redirection:

- Input redirection:

```
>myprogram < sequences.txt
```

- Output redirection:

```
>myprogram < sequences.txt > output.txt
```

Agenda:

- Logical expressions
  - logical/boolean type: `true` or `false` (`<stdbool.h>`)
  - operators: relational, equality, logical
- The `if` statement
  - cascaded form:

```
if ( expression ) {
    statements
}
else if ( expression ) {
    statements
}
else {
    statements
}
```

  - conditional expression:

```
expr1 ? expr2 : expr3
```

  - `switch` statement

Reading:

- Textbook: Chapter 5