CMPUT 201: Practical Programming Methodology

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

Lecture 6: Selection Statements

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

Logical expressions:

- Statements that test the value of an expression
  - "true" or "false" (exactly one of the two)
  - Boolean or logical type
- Relational operators
  - <, >, <=, >=
  - precedence lower than arithmetic operators
  - left associative
  - a relational expression yields either 0 or 1 (integer value)

    e.g. "`1 < 2.5`" has the value 1

    e.g. "`1 < 2.5 < 1.1`" has the value 1 (can you explain?)

Logical expressions:

- Equality operators
  - ==, != (not equal to)
  - precedence lower than relational operators
  - left associative
  - an equality expression yields either 0 or 1 (integer value)

    e.g. "`(i >= j) + (i == j)`" is either 0, 1, or 2 (i $<$ j, i $>$ j, or i == j, respectively)
- Logical operators
  - ! (not), && (and), || (or)
  - ! (unary) precedence the same as unary +, −

    && and || precedence lower than equality operators
  - left associative
  - a logical expression yields either 0 or 1 (integer value)
  - any non-zero operand is "true" (1)
  - short-circuit evaluation

    e.g. "`(i != 0) && (i == j)`"

Precedence:

highest: {postfix ++, −−} $\prec$

{prefix ++, −−, unary +, − unary !} $\prec$

{*, /, %} $\prec$

{binary +, −} $\prec$

{<, >, <=, >=} $\prec$

{==, !=} $\prec$

{&&, ||} $\prec$

lowest: {assignment =, +=, −=, *=, /=, %=}

The `if` statement:

- Form

  ```
  if ( expression ) statement
  ```

  - required use of parentheses to enclose the expression
  - non-zero value is "true"
  - when "true", the statement is executed

- e.g., idioms (testing whether $0 \leq$ `i` $<$ `n`)

  ```
  if (0 <= i && i < n) statement
  ```

  ```
  if (i < 0 || i >= n) statement
  ```

- Compound statements enclosed by braces

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

The `if` statement:

- The general (cascaded) form

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

  - logically, exactly one compound statement executed

Calculating a Broker's commission:

- A broker charges the amounts (Page 81)

Transaction size	*Commission rate*
Under $2,500	$30 + 1.7%
$2,500–6,250	$56 + 0.66%
$6,250–20,000	$76 + 0.34%
$20,000–50,000	$100 + 0.22%
$50,000–500,000	$155 + 0.11%
over $500,000	$255 + 0.09%

  The minimum charge is $39

- Expected appearance:

```
Enter value of the trade: 30000
Commission: $166.00
```

Conditional expressions:

- A special `if` statement
- Form:

  `expr1 ? expr2 : expr3`

- Ternary operator (? and :)
- Read "if `expr1` then `expr2` else `expr3`"

```
i > j ? i-- : j++;
```

```
if (i > j)
    i--;
else
    j++;
```

Boolean values (in c99):

(You will have to define them yourself if using c89)

- Type _Bool
  - is an integer type
  - two possible values 0 and 1 (non-zero)
- `#include <stdbool.h>`
  - provides a macro `bool` for _Bool
  - macros for `true` and `false` (stand for 1 and 0, respectively)

The `switch` statement:

- Special cascaded `if` statement, for example

```
if (grade == 4) {
     printf("Excellent");
}
else if (grade == 3) {
     printf("Good");
}
else if (grade == 2) {
     printf("Average");
}
else if (grade == 1) {
     printf("Poor");
}
else if (grade == 0) {
     printf("Failing");
}
else {
     printf("Illegal grade");
}
```

The `switch` statement:

- Using `switch`, equivalently

```
switch (grade) {
     case 4: printf("Excellent");
             break;
     case 3: printf("Good");
             break;
     case 2: printf("Average");
             break;
     case 1: printf("Poor");
             break;
     case 0: printf("Failing");
             break;
     default: printf("Illegal grade");
             break;
}
```

- Notes:
  - `grade` is the controlling expression, type `int/char`
  - `4` is the case label, one constant-expression only
  - using `break;` to get out of `switch` (otherwise all following statements executed)
  - `default` is like `else` associated for the `if` statement

Printing a date in legal form (Page 89):

- Appearance:

```
Enter data (mm/dd/yy):  02/05/16
Dated this 5th day of February, 2016.
```

- Cares:
  - 1st, 2nd, 3rd, 4th, ...
  - one space between tokens
  - `switch` or cascaded `if`?

Agenda:

- Loop: a repeatedly executed statement
- Binary search (demo)
- The `while` statement
  - `while ( controlling expression ) statement`
- The `do` statement
  - `do { statement } while ( controlling expression );`
- The `for` statement
  - `for ( expr1; controlling expression; expr3 ) statement`
- Exiting from a loop: `break`, `continue`, `goto`
- The null statement
  - `for (;;) { statements }`

Reading:

- Textbook: Chapter 6

Loop:

- A statement itself
  - to repeatedly execute some other statement
  - called *loop body*
- A controlling expression
  - evaluated each time the loop body is executed
  - "true" (non-zero): continue the loop
  - "false" (zero): terminate the loop
- Three iteration statements
  - `while`
  - `do`
  - `for`

The `while` statement:

- Form

  `while ( controlling expression ) statement`

  - if `controlling expression` is false at the first place, loop body is not executed
  - (usually) when loop terminates, `controlling expression` must be false

    e.g., when the following loop terminates, we have `i >= n`:

    ```
    i = 1;
    while (i < n)
        i *= 2;
    ```

- Infinite `while`-loop

  `while (1) statement`

  - have to use loop-exiting statement to terminate

`while (1) statement` 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;

      while (1) { /* replacing previous 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;
  }
  ```
  ```

The `do` statement:

- Form

  ```
  do { statement } while ( controlling expression );
  ```

  - the loop body is executed, then `controlling expression` is evaluated
  - (usually) when loop terminates, `controlling expression` must be false

    e.g., when the following loop terminates, we have `i >= n`:

    ```
    i = 1;
    do {
        i *= 2;
    } while (i < n);
    ```

  - the loop body is executed at least once
  - always use { ... } to enclose the loop body (for better reading)
  - and put `while` after the right brace } — indicating not a `while`-loop
  - and the `;` after the controlling expression?

Example, printing a table of squares:

- Page 102
- Appearance as:

```
This program prints a table of squares starting from 3^2.
Enter the maximum number to be squared in the table: 12
         3         9
         4        16
         5        25
         6        36
         7        49
         8        64
         9        81
        10       100
        11       121
        12       144
```

- Testing `while` and `do` statements

The `for` statement:

- Ideal for loops having a "counting" variable!
- Form

  `for ( expr1; controlling expression; expr3 ) statement`

  e.g.,

```
for (i = 3; i < n; i++)
    printf("%10d%10d\n", i, i * i);
```

  - `expr1` is executed only once, the time entering the `for`-loop — initialization
  - `controlling expression` is evaluated every iteration

    `if` "true", execute the loop body

    `else`, terminate the loop
  - `expr3` is executed at the end of loop body

    if the loop body was executed, of course

Re-writing the `for` statement as a `while` statement:

- The `for`-loop

```
for ( expr1; controlling expression; expr3 ) statement
```

- A `while`-loop

```
expr1;
while ( controlling expression ) {
    statement
    expr3;
}
```

- Be careful of side effects, if merging `controlling expression` and `expr3`

- Re-write as a `do`-loop ?

`while (1) statement` 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;

      while (1) { /* replacing previous 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;
  }
  ```
  ```

Some typical `for` statement idioms:

- Counting up from 0 to $n-1$:

  ```
  for (i = 0; i < n; i++) ...
  ```

- Counting up from 1 to n:

  ```
  for (i = 1; i <= n; i++) ...
  ```

- Counting down from $n-1$ to 0:

  ```
  for (i = n-1; i >= 0; i--) ...
  ```

- Counting down from n to 1:

  ```
  for (i = n; i > 0; i--) ...
  ```

- Cares:
  - use of $<$ or $<=$
  - off-by-1 errors (particularly, array index starting at 0!)
  - omitting/missing either expression(s), for example "`for (;;) ...`"

The comma "," operator:

- We used it earlier in declarations such as "`int i, j;`"
- More generally it can be used to "*glue*" multiple expressions into one
  - similar use of { and } to create a compound statement
- Form

  `expr1, expr2`
  - `expr1` evaluated, its value discarded
  - `expr2` evaluated, its value is the value of the entire expression `expr1, expr2`

  e.g.

  ```
  i = 1, j = 2, i + j, k = i + j;
  ```

- Can be replaced by ";" operator

  ```
  expr1;
  expr2
  ```

- Useful in places where a single expression is allowed!

  ```
  for (i = 0, j = 0; i < n; i++) ...
  ```

Calculating squares:

- **Goal:** calculating squares w/o multiplications
- A sequence of odd numbers: $1, 3, 5, \ldots, a_i = 2i - 1, \ldots (i = 1, 2, 3, \ldots)$
  - a_i is the i-th term
- S_n is the sum of the first n numbers
  - calculated as:

$$S_n = \sum_{i=1}^{n} a_i = n^2$$

- Calculating squares w/o multiplications (Page 110)

$$a_i = a_{i-1} + 2$$

$$S_n = S_{n-1} + a_n$$

Exiting from a loop:

- Normal exiting points: before or after the loop body
  - the value of the controlling expression
- Exiting in the middle?
  - recall the `break` statement inside the `switch` statement
  - it can also be used to "jump out of" a `while/do/for`-loop, e.g.

```
while (1) { /* 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;
}
```

  - care: jumping out of the *innermost* `switch/while/do/for` statement

Exiting from a loop:

- The `continue` statement

```
while (1) { /* 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);
        continue;
    }
    break;
}
```

  - doesn't really exit
  - but ends the current iteration only

    (`continue` not to be used inside a `switch` statement)

Exiting from a loop:

- The `goto` statement

```
identifier: statement
...
...
for ( ; ; ) {
    ...
    goto identifier ;
    ...
}
```

  - jumping to any statement (in the same function, restrictions apply)
  - the statement labelled with "`identifier`"

```
while ( ... ) {
    switch ( ... ) {
        ...
        goto while-loop_done; /* break won't work here */
        ...
    }
}
while-loop_done: ...
```

The null statement:

- By the name, does nothing

```
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;
}
```

  - null initialization

    (null controlling expression — not a statement)

  - null statement after the loop body is executed

  - they are there due to syntax

The null statement:

- Primarily use for writing loops with empty body, e.g. (primality testing)

```
for (d = 2; d < n; d++)
    if (n % d == 0)
        break; /* not a prime */
if (d < n)
    printf("%d is divisible by %d\n", n, d);
```

  - can be further simplified as:

```
for (d = 2; d < n && n % d != 0; d++);
if (d < n)
    printf("%d is divisible by %d\n", n, d);
```

  - common errors: extra ";" resulting empty statements

```
for (d = 2; d < n && n % d != 0; d++);
if (d < n);
    printf("%d is divisible by %d\n", n, d);
```

Instructions (during all exams):

- Read these instructions and wait for the signal to turn this cover-sheet over.
- Use space below/beside the questions to write your solutions legibly.
- Closed book;
  - no electronic devices (make sure your cellphone is OFF),
  - no calculators,
  - no conversations.
- In general, no questions will be answered during the quiz;
  - if unsure, state your best assumptions clearly and proceed;
- We will provide an exam clock.

Problem 1 (5 marks)

- Consider the following C code snippet. What is the output?

```
int x, y = 11;
for (x = 11; --x > 0; x--) {
    if (x-- < 2 && y++ > 0) x++;
    printf("%d %d\n", x, y);
}
printf("%d %d\n", x, y);
```

Problem 2 (5 marks)

- Consider the following declarations:

```
int i = 12;
float y = 2.125;
```

- Convert the two numbers into binary (base-2) format first, and then write out their machine formats (*i.e.*, the binary strings stored in their memory blocks), respectively.

Problem 3 (5 marks)

- Consider the following C code snippet, which is intended to binary search for the fahrenheit equivalent to celsius 36.2^o.

```
float fahrenheit, celsius = 36.2;
float lower = -100.0, upper = 100.0;
for (;;) {
    fahrenheit = (lower + upper) / 2.0;
    if ((fahrenheit - 32.0) * 5.0 / 9.0 > celsius) upper = fahrenheit;
    else if ((fahrenheit - 32.0) * 5.0 / 9.0 < celsius) lower = fahrenheit;
    else {
        printf("Fahrenheit equivalent: %.6f\n", fahrenheit);
        break;
    }
}
```

- What is the (possible, or likely) output? Describe your reasoning.

Problem 1 (5 marks)

- Consider the following C code snippet. What is the output?

```
int x, y = 11;
for (x = 11; --x > 0; x--) {
    if (x-- < 2 && y++ > 0) x++;
    printf("%d %d\n", x, y);
}
printf("%d %d\n", x, y);
```

- Testing points:
  - in-/de-crementer (prefix, suffix),
  - short-circuit,
  - side effect

Problem 1 (5 marks)

- Consider the following C code snippet. What is the output?

```
int x, y = 11;
for (x = 11; --x > 0; x--) {
    if (x-- < 2 && y++ > 0) x++;
    printf("%d %d\n", x, y);
}
printf("%d %d\n", x, y);
```

- Testing points:
  - in-/de-crementer (prefix, suffix),
  - short-circuit,
  - side effect

- Output (yes, it is tracing the program):

```
9 11
6 11
3 11
1 12
-1 12
```

Problem 1 (5 marks)

- Consider the following C code snippet. What is the output?

```
int x, y = 11;
for (x = 11; --x > 0; x--) {
    if (x-- < 2 && y++ > 0) x++;
    printf("%d %d\n", x, y);
}
printf("%d %d\n", x, y);
```

- In general, do not "merge `controlling expression` and `expr3`"

- But, the above can be improved to (for readability):

```
int x, y = 11;
for (x = 10; x > 0; x -= 2) {
    if (x-- < 2 && y++ > 0) x++;
    printf("%d %d\n", x, y);
}
printf("%d %d\n", x, y);
```

Problem 2 (5 marks)

- Consider the following declarations:

```
int i = 12;
float y = 2.125;
```

- Convert the two numbers into binary (base-2) format first, and then write out their machine formats (*i.e.*, the binary strings stored in their memory blocks), respectively.

- Testing points:
  - binary conversion,
  - `int` storage,
  - `float` storage (usual floats, exponent from `00000001` to `11111110`)

Problem 2 (5 marks)

- Consider the following declarations:

```
int i = 12;
float y = 2.125;
```

- Convert the two numbers into binary (base-2) format first, and then write out their machine formats (*i.e.*, the binary strings stored in their memory blocks), respectively.

- Testing points:
  - binary conversion,
  - `int` storage,
  - `float` storage (usual floats, exponent from `00000001` to `11111110`)

- Solution:
  - $12_{10} = \{\ 1100\ \}_2$;
  - $2.125_{10} = \{\ 10.001\ \}_2$;
  - { 0 0000000 00000000 00000000 00001100 };
  - { 0 10000000 0001000 00000000 00000000 }

Problem 3 (5 marks)

- Consider the following C code snippet, which is intended to binary search for the fahrenheit equivalent to celsius 36.2^o.

```
float fahrenheit, celsius = 36.2;
float lower = -100.0, upper = 100.0;
for (;;) {
    fahrenheit = (lower + upper) / 2.0;
    if ((fahrenheit - 32.0) * 5.0 / 9.0 > celsius) upper = fahrenheit;
    else if ((fahrenheit - 32.0) * 5.0 / 9.0 < celsius) lower = fahrenheit;
    else {
        printf("Fahrenheit equivalent: %.6f\n", fahrenheit);
        break;
    }
}
```

- What is the (possible, or likely) output? Describe your reasoning.

- Testing points:
  - binary search :-)
  - `float` precision

Problem 3 (5 marks)

- Consider the following C code snippet, which is intended to binary search for the fahrenheit equivalent to celsius 36.2^o.

```
float fahrenheit, celsius = 36.2;
float lower = -100.0, upper = 100.0;
for (;;) {
    fahrenheit = (lower + upper) / 2.0;
    if ((fahrenheit - 32.0) * 5.0 / 9.0 > celsius) upper = fahrenheit;
    else if ((fahrenheit - 32.0) * 5.0 / 9.0 < celsius) lower = fahrenheit;
    else {
        printf("Fahrenheit equivalent: %.6f\n", fahrenheit);
        break;
    }
}
```

- Solution:
  - If program terminates, the output would be "`Fahrenheit equivalent:  97.160000`";

    (`36.2 * 9.0 / 5.0 + 32.0 = 97.16`)
  - termination requires the calculated celsius by "`(fahrenheit - 32.0) * 5.0 / 9.0`"

    and the input celsius by "`celsius = 36.2`"
  - two floats are exactly equal to each other, which is often unlikely to happen!

    (in our case, `97.160004` vs. `97.160000`)

Agenda:

- Integer types
  - `int`
  - built-in types (no need standard libraries)
  - constants, variables: storage (machine format)
  - type conversion
  - type definitions
  - `sizeof` operator: return #bytes
- Floating types
- Character types
  - ```
    scanf("%c", &ch);
    ch = getchar();
    ch = getc(stdin);
    ```

Reading:

- Textbook: Chapter 7
    ```