Chapter 5 Loops and Relational Expressions

5.1 Expressions and Statements

- Any value or any valid combination of values and operators constitutes an *expression*. A expression can be evaluated, yielding a value. For example, `10`, `x`, `27 + 15`, `x = 20`.
  - The value of an assignment statement is the value of member on the left, namely, the assignee, after the assignment takes place. The fact that assignment expressions have values permits statements such as `a = (b = 4) +3;`
  - The assignment operator associates right-to-left, so in the expression `x = y = z = 0;`, 0 is first assigned to `z` and last to `x`.
  - If the very act of evaluating an expression changes the value of data in memory (i.e. assignment), we say the evaluation has a *side effect* - the primary effect is evaluating the expression, from the standpoint of how the language works. Not all expression have side effects, of course. For example, `x + 5` calculates a new value, but it does not change the value of `x`.
  - The comma operator `,` merges multiple expressions into one.
    - Subexpressions are evaluated from left to right.
    - The value of the merged expression equals to the value of the last-evaluated subexpression, in other words, the rightmost subexpression.
    - The comma operator has the lowest precedence of all operators, and associates from left to right.
    - However, a comma `,` does NOT always serve as the comma operator - for instance, in `int a, b;` the comma is used to separate identifiers.
- An expression or a combination of expressions trailed by a semicolon `;` constitutes a *statement*, or more precisely, an *expression statement*.
  - In C/C++, semicolon `;` is NOT an operator, and is NOT part of expressions.
  - Statements can not be evaluated (thus has no "value"). Only expressions can.
  - Removing `;` from a statement does not necessarily convert it to an expression. For example, removing `;` from the *declaration statement* `int a;` or `int a = 0;` does not produce an expression, and the resulting fragment `int a` or `int a = 0` does not have a value. It makes code like this invalid: `b = int a;`, `b = (int a = 0) + 1;`.
    - Rules can be bent sometimes. Now that `int a = 0` is not an expression, `for(int i = 0; ...;...) statement` should be outlawed. However, it provides convenience in a `for` loop, so the standard legitimized such code, but ONLY in a `for` statement. In short, as an exception to the foregoing rules, this code is permitted.

      > `for`, `while`, and `do-while` loop statements

```
for (exprInit; exprTest; exprStep) statement
while (exprTest) statement
do statement while(exprTest);
// expressions can be empty, but ';' are mendatory.
// Loops continue as long as the value of exprTest is true.
```

    - Note than `int a = 0;` should not be syntactically regarded as `int a, a = 0;` because the comma `,` here is not interpreted as a comma operator, but as an identifier separator, and that statement is invalid because the compiler thinks you declared the same variable `a` twice.

- Apart from expression statements and declaration statements, there are *comound statement*s, or *block*s. A compound statement (block) is constructed by one or more statements included in a pair of braces `{}`.
    - A compound statement, a.k.a. block, counts as a single statement.
    - Automatic variables declared within a block exists only as long as the block is being executed, and therefore the variable is known only within the block.

    > The comma operator `,` merges expressions into one, and the brace-pair `{}` merges statements into one.

- Sequence points
    - A side effect is an effect that occurs when evaluating an expression modifies data in memory, such as a value stored in a variable. A *sequence point* is a point in program execution at which all side effects are guaranteed to have taken place before going on the next step.
    - In C++, a semicolon `;` marks a sequence point, which means all changes made by assignment operators and increment/decrement operators in a statement must take place before a program proceeds to the next statement. The comma operator `,` and some other operators, also have sequence points. And the end of any full expression has a sequence point.
        - *Full expression*: an expression that is not a subexpression of a larger expression. For example, expression trailed by `;`, and the condition-test expression in the `while` loop (note that it does not end with `;`)
        - A example of full expressions

        ```
        while (i++ < 10) std::cout << i << std::endl; // expression 1
        y = (4 + x++) + (6 + x++);                    // expression 2
        ```

        In expression 1, because the test condition for the `while` loop statement is a full expression, so after the `<` is evaluated, the `++` takes place, well before the `cout` statement is executed.

In expression 2, because `4 + x++`, along as `6 + x++`, is a subexpression of the entire assignment expression, C++ does NOT guarantee that `x` will be incremented immediately after `4 + x++` is evaluated. The full expression here is the entire assignment expression, so all that C++ guarantees is that `x` will have been incremented twice by the time the program moves to the next statement. C++ does NOT specify whether `x` is incremented after each subexpression is evaluated or only after all subexpressions are evaluated.

> Prefixed `++` and postfixed `++`: prefixing (`++x`)means "incrementing before using"; and postfixing (`x++`) means "using before incrementing". The rule is similar for `--`.
>
> A side note: in C++, prefix increment/decrement operator the dereference operator `*` share the same precedence, which is lower than that of postfix increment/decrement:
>
> postfix `++`, `--` > prefix `++`, `--` = `*`
>
> And their associativity is right-to-left. For instance,
>
> `x = *++pt` <=> `x = *(++pt)` `y = ++*pt` <=> `y = ++(*pt)`
>
> `z = *pt++` <=> `z = *(pt++)`. But keep in mind that postfix `++` means "using before incrementing", so it actually means "assign `*pt` to `z`, and then increment the pointer: `pt = pt + 1`."

- C++11 standard has dropped the term "sequence point", however. Because the term does not carry over well while discussing multiple threads of execution. Instead, it adopted a descriptive approach where descriptions are framed in terms of sequencing, with some events being described as being sequenced before other events. The term is dropped; the rules are still valid.

5.2 The range-based `for` Loop (C++11)

- C++11 adds a new form of loop called the *range-based* `for` loop. It simplifies one common loop task - that of doing something with each element of an array, or more generally, of one of the container classes such as `vector` and `array`.
- Basic Usage:
  - If elements in the set (say, an array or a `vector` object) are NOT to be modified within the loop, the syntax is `for (type varName : setName) statement`.

```
double prices[] = {4.99, 10.99, 6.00, 7.99, 5.99};
for (double x : prices)
{
  std::cout << x * 0.8 << std::endl; // does not modify the element
  x = x * 0.8;  // no error, but does not modify the set
}
```

Any change made to `x` will NOT be done to the set.

- If the elements are to be (partially or entirely) modified, the syntax is different: `for (type &varName : setName) statement`. This form is more general.

```
double prices[] = {4.99, 10.99, 6.00, 7.99, 5.99};
for (double &x : prices) // x is a reference variable
{
  x = x * 0.8;                       // modifies the element's value
  std::cout << x << std::endl;
}
```

Here `x` is a reference variable, which has the address of the element it refers to. So any change to `x` is actually done to the set.

□

Chapter 6 Branching Statements and Logical Operators

6.1 The branching statements

- The `if` and `if-else` statement: an `if` statement lets a program decide whether a particular statement or block is executed, and an `if-else` lets a program decide which of two statements or blocks is executed.
  - Syntax:

```
// if
if (exprCondition) statement
// if-else
if (exprCondition) statement1 else statement2
// if-else in a more friendly way
if (exprCondition)
  statement1
else
  statement2
```

  - Because the statements within the `if` or `if-else` statement can be any statement, so there is a handy way of writing multiple-branching statements. Take 3-branching as an example:

```
if (exprConditionA)
  statement1
else if (exprConditionB)
       statement2
     else
       statement3
```

In a more common fashion,

```
if (exprConditionA)
  statement1
else if (exprConditionB)
  statement2
else
  statement3
```

- The `switch-case` statement: unlike `if-else`, this kind of statement is inherently born for multiple-branching statements.
  - Syntax:

```
switch(integer_exprTest) // must returns an integer
{
  case label1 : statement1 // break; (may be needed)
  case label2 : statement2 // break; (may be needed)
  ...
  default     : statement  // not necessary
}
```

  - The test expression `integer_exprTest_int` in the pair of parenthesis must an expression that reduces to an integer value. Also each label must be an integer constant expression. Most often, labels are simple `int` or `char` constants such as `1` or `q`, or enumerators.
  - Each case label functions only as a line label, NOT as a boundary. That is, after a program jumps to a particular line in a switch, it then sequentially executes all the statements following that line in the `swtich-case`, unless it encounters a `break;`. So you must include a `break;` statement at the right places to if you want to confine execution to a particular portion of a `switch-case` statement.
  - Statements can be empty. In this case, the program proceeds.

```
switch(ternaryNumber)
{
  case -1: // empty
  case  0: std::cout << "non-positive" << std::endl; break;
  case  1: std::cout << "pisitive" << std::endl; break;
  // the default case can be dropped if you are confident
  // that ternaryNumber can only be -1, 0, 1.
}
```

- The `goto` statement: it is strongly advised that you do not use it because it may mess up the control flow, making the code hard to maintain, if it is used unwisely. Its syntax is `goto label`.

```
char ch; std::cin >> ch;
if (ch == 'P') goto paris; // directs the program to the "paris" label
...
paris: std::cout << "Welcome to Paris!" <<std::endl; break;
```

6.2 Logical expressions

- Logical operators: logical AND `&&`, logical OR `||`, logical NOT `!`. Their alternative representations are `or`, `and`, `not`, respectively. These alternatives are reserved keywords in C++ so that programmers with non-American keyboards can use them. In C, these alternatives as operators, provided that the header `iso646.h` is included.
- Precedence: logical NOT `!` > relational operators > logical AND `&&` > logical OR `||`.
- Logical OR `||` and logical AND `&&` operators have a lower precedence than relational operators, while logical NOT operator has a higher precedence. So `x > 5 && x < 10` is equivalent to `(x > 5) && (x < 10)`, and `!x > 5` is equivalent to `(!x) > 5`, which is not `!(x > 5)`.

  > Relational operators: less than `<`, less than or equal to `<=`, greater than `>`, greater than or equal to `>=`, equal to `==`, not equal to `!=`.
  >
  > Their precedence: `<`, `<=`, `>`, `>=` are higher than `==`, `!=`.

- *Short-circuit* mechanism: C++ guarantees that when a program evaluates a logical expression, it evaluates it from left to right, and stops evaluation as soon as it knows the answer.

  > true || any == true (1 || x == 1), false && any == false (0 && x == 0).

6.3 The `cctype` library of character functions

- C++ has inherited from C a handy package of character-related functions, prototyped in the `cctype` header file. The frequently used functions are functions that return Boolean values (`true` or `false`), and some are listed below:

Function name	It returns `true` if the argument...
`isalum()`	...is alphanumeric (that is, a letter or a digit).
`isalpha()`	...is alphabetic.
`isblank()`	...is a space or a horizontal tab.
`iscntrl()`	...is a control character.
`isdigit()`	...is a decimal digit character (`0` - `9`)
`isgraph()`	...is any printing character other than a space.
`islower()`	...is a lowercase letter.
`isprint()`	...is any printing character, including a space.
`isspace()`	...is a standard white-space character (that is, a space, formfeed, newline, carriage return, horizontal tab, vertical tab)
`isupper()`	...is an uppercase letter.
`isxdigit()`	...is a hexadecimal digit character (`0` - `9` , `a` - `f` , `A` - `F`)

Plus, `tolower()` return the lowercase version of the argument if it is an uppercase letter, or otherwise returns it unaltered; `toupper()` returns the uppercase version of the argument if it is a lowercase letter, or otherwise returns it unaltered.

6.4 The conditional operator `? :`

- The `? :` can be used as more concise but less obvious form of the `if-else` statement. It is the only ternary operator in C/C++, unless the standard-making committee decides to introduce something crazy in the future release.
- Syntax: `exprTest ? exprTrue : exprFalse`. If `exprTest` is true, then the whole expression's value is the value of `exprTrue`. Otherwise it is the value of `exprFalse`.
- The `? :` can be nested within one another, which is preferred by weirdos. This mild example gives you a taste:

```
const char x[2][20] = {"Jason", "at your service\n"};
const char *y = "Quillstone";
for (int i = 0; i < 3; i++)
  std::cout << ((i < 2)? !i ? x[i] : y : x[1]) << std::endl;
// output:
// Jason Quillstone at your service
```

□