Chapter 4: Program Control

* A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true.
* Two means of repetition:
    1. Counter-controlled repetition: is sometimes called definite repetition because we know in advance exactly how many times the loop will be executed.
    2. Sentinel-controlled repetition: is sometimes called indefinite repetition because it is not known in advance how many times the loop will be executed.
* In counter-controlled repetition, a control variable is used to count the number of repetitions.
* Sentinel values are used to control repetition when:
    1. The precise number of repetitions is not know in advance.
    2. The loop includes statements that obtain data each time the loop is performed.
* The sentinel value indicates "end of data" which must be distinct from regular data items.

Counter-Controlled Repetition

* Counter-controlled repetition requires:
    1. The name of a control variable (or loop counter).
    2. The initial value of the control variable.
    3. The increment (or decrement) by which the control variable is modified each time through the loop.
    4. The condition that tests for the final value of the control variable.
* E.g.

```
#include <stdio.h>
main()
{
    int counter = 1;
    while (counter <= 10)
    {
        printf("%d\n", counter);
        ++counter;
    }
    return 0;
}
```

* Or,

```
#include <stdio.h>
main()
{
    int counter = 0;
    while (++counter <= 10)
        printf("%d\n", counter);
    return 0;
}
```

The "For" Repetition Structure

* The "for" repetition structure handles all the details of counter-controlled repetition automatically.
* E.g.

```
#include <stdio.h>
main()
{
    int counter;
```

```
            for (counter = 1; counter <= 10; counter++)
                printf("%d\n", counter);
            return 0;
        }
```
* Note that the sequence of instructions executed is exactly the same as the above "while" example.

* The general format of the "for" structure is

```
        for (expression1; expression2; expression3)
            statement
```

    1. expression1: initializes the loop's control variable
    2. expression2: is the loop-continuation condition
    3. expression3: increments the control variable.

* In most cases the "for" structure can be represented with an equivalent "while" structure:

```
        expression1;
        while (expression2)
        {
            statement
            expression3;
        }
```

* Often, "expression1" and "expression2" are comma-separated lists of expressions. E.g.

```
        for (num=2; num <= 100; sum+=num, num+=2)
```

* The three expressions are optional:
    1. If expression2 is omitted, C assumes that the condition is true, thus creating an infinite loop.
    2. Expression1 might be omitted if the control variable is initialized elsewhere in the program.
    3. Expression3 might be omitted if the increment is calculated by statements in the body, or if no increment is needed.

The "For" Structure: Notes and Observations
---

* The initialization, loop-continuation condition, and increment can contain arithmetic expression. E.g.

```
        for (j=x; j< 4*x*y; j+=y/x)
```

* The "increment may be negative. E.g.

```
        for (i=10; i>=0; i--)
```

* If the loop-continuation condition is initially false, the body portion of the loop is not performed.
* It is common to use the control variable for controlling repetition while never mentioning it in the body of the loop.
* Although the value of the control variable can be changed in the body of a "for" loop, this can lead to subtle errors. It is best not to change it.

* E.g. A person invests $1000.00 in a saving account yielding 5 percent interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:

$$a = p(1+r)^n$$

where p is the principal, r is the annual interest rate, n is the number of years, a is the amount on deposit at the end of the nth year.

```
        #include <stdio.h>
        #include <math.h>
```

```
        main()
        {
            int year;
            double amount, principal = 1000.0, rate = 0.05;

            printf("%4s%21s\n", "Year", "Amount on deposit");
            for (year=1; year<=10; year++)
            {
                amount = principal * pow(1.0 + rate, year);
                printf("%4d%21.2f\n", year, amount);
            }
            return 0;
        }
```

* The type "double" is a floating-point type much like "float", but a variable of type "double" can store a value of much greater magnitude with greater precision than "float".
* Although C does not include an exponentiation operator, we can use the standard library function "pow" for this purpose.
* Note that the header file "math.h" should be included whenever a math function such as "pow" is used.

The Switch Multiple-Selection Structure

* Occasionally, an algorithm will contain a series of decisions in which a variable or expression is tested separately for each of the constant integral values it may assume, and different actions are taken.
* The "switch" structure consists of a series of "case" labels, and an optional "default" case.
* E.g.

```
        #include <stdio.h>
        main()
        {
            int grade;
            int aCount = 0, bCount = 0, cCount = 0,
                dCount = 0, fCount = 0;
            printf("Enter the letter grades.\n");
            printf("Enter the EOF character to end input.\n");

            while ((grade = getchar()) != EOF)
            {
                switch (grade)
                {
                    case 'A': case 'a':
                        ++aCount;
                        break;
                    case 'B': case 'b':
                        ++bCount;
                        break;
                    case 'C': case 'c':
                        ++cCount;
                        break;
                    case 'D': case 'd':
                        ++dCount;
                        break;
                    case 'F': case 'f':
                        ++fCount;
                        break;
                    case '\n': case ' ':
```

```
                    break;
                default:
                    printf("Incorrect letter grade entered.");
                    printf(" Enter a new grade.\n");
                    break;
            }
        }
        printf("\nTotals for each letter grade are:\n");
        printf("A: %d\n", aCount);
        printf("B: %d\n", bCount);
        printf("C: %d\n", cCount);
        printf("D: %d\n", dCount);
        printf("E: %d\n", eCount);
    }
```

* Characters must be enclosed within single quotes to be recognized as character constants (its ASCII code).
* The function "getchar" (from the standard input/output library) reads one character from the keyboard and store that character in integer variable "grade".
* Characters are normally stored in variables of type "char".
* We can treat a character as either an integer or a character depending on its use. E.g.

```
printf("The character (%c) has the value %d.\n",'a','a');
```

* Assignment statements have value can be useful for initializing several variables to the same value. E.g.

```
a = b = c = 0;
```

* We use 'EOF' (which normally has the value -1, which is an integer value) as the sentinel value of "end of file".

* The keyword "switch" is followed by the variable name "grade" in parentheses. This is called the controlling expression.
* The value of this expression is compared with each of the "case" labels.
* If a match occur, the statements for that "case" are executed.
* The "switch" structure is exited immediately with the "break" statement.
* If "break" is not used anywhere in a "switch" structure, then each time a match occurs in the structure, the statements for all the remaining "case"s will be executed.
* If no match occurs, the "default" case is executed.
* The "switch" structure is different from all other structures in that braces are not required around multiple actions in a "case".

* Reading characters one at a time can cause some problems. To have the program read the characters, they must be sent to the computer by pressing the return key on the keyboard.
* This causes the newline character to be placed in the input after the character we wish to process.
* often, this newline character must be specially processed to make the program work correctly.

* When using the "switch" structure, remember that it can only be used for testing a constant integral expression.

The "Do/While" Repetition Structure

* The "do/while" structure tests the loop-continuation condition after the loop body is performed, therefore the loop body will be

executed at least once.
* E.g.

```
#include <stdio.h>
main()
{
    int counter = 1;

    do
    {
        printf("%d  ",counter);
    } while (++counter <= 10);
    return 0;
}
```

The Break and Continue Statements
---
* The "break" statement, while executed in a loop or "switch" structure, causes immediate exit from the structure.
* Common uses of the "break" statement are to escape early from a loop, or to skip the remainder of a "switch" structure.
* E.g.

```
#include <stdio.h>
main()
{
    int x;

    for (x=1; x<=10; x++)
    {
        if (x==5)
            break;
        printf("%d ",x);
    }
    printf("\nBroke out of loop at x == %d\n", x);
    return 0;
}
```

* The "continue" statement, when executed in a loop structure, skips the remaining statements in the body of that structure, and performs the next iteration of the loop.
* E.g.

```
#include <stdio.h>
main()
{
    int x;

    for (x=1; x<=10; x++)
    {
        if (x==5)
            continue;
        printf("%d ",x);
    }
    printf("\nUsed continue to skip printing the value 5\n",
                                    x);
    return 0;
}
```

Logical Operators
---
* C provides logical operators that may be used to form more complex

conditions by combining simple conditions.
* The logical operators are
    1. && - logical AND
    2. || - logical OR
    3. ! - logical NOT

* Logical AND ensures that two conditions are both true before we choose a certain path of execution.
* E.g.

```
        if (gender == 1 && age >= 65)
            ++seniorFemales;
```

* Truth table for the &&:

expression1	expression2	expression1 && expression2
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

* C accepts any nonzero value as true.

* Logical OR ensures that either or both of two conditions are true before we choose a certain path of execution.
* E.g.

```
        if (semesterAverage >= 90 || finalExam >= 90)
            printf("Student grade is A\n");
```

* Truth table for the ||:

expression1	expression2	expression1 || expression2
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

* The logical negation operator is placed before a condition when we are interested in choosing a path of execution if the original condition is false.
* E.g.

```
            if (!(grade == sentineValue))
                printf("The next grade is %f\n", grade);
```

* Truth table for the !:

expression1	!expression1
0	1
nonzero	0