Chapter 5

REPETITION STRUCTURES
Outline

- Overview
- *while* loops
- *for* loops
- *do–while* Loops
- Interactive *while* loops
- Nested loops
- Break and Continue statements
Overview

- C++ provides three different forms of repetition structures:
  1. **while** structure
  2. **for** structure
  3. **do-while** structure

- Each of these structures requires a condition that must be evaluated.

- The condition can be tested at either (1) the beginning or (2) the end of the repeating section of code.

- If the test is at the beginning of the loop, the type of loop is a **pre-test loop**.

- If the test is at the end of the loop, the type of loop is a **post-test loop**.
Fixed count loop and variable condition loop

- In addition to where the condition is tested, repeating sections of code are also classified.

- In a *fixed count loop*, the condition is used to keep track of how many repetitions have occurred. In this kind of loops, a fixed number of repetitions are performed, at which point the repeating section of code is exited.

- In many situations, the exact number of repetitions are not known in advance. In such cases, a *variable condition loop* is used.

- In a *variable condition loop*, the tested condition does not depend on a count being achieved, but rather on a variable that can change interactively with each pass through the loop. When a specified value is encountered, regardless of how many iterations have occurred, repetitions stop.
while loops

The *while* statement is used for repeating a statement or series of statements as long as a given conditional expression is evaluated to true.

The syntax for the *while* statement:

```
while (condition expression) statement
```

Programming Fundamentals
Example 5.2.1

// This program prints out the numbers from 1 to 10
#include <iostream.h>
int main()
{
    int count;
    count = 1;       // initialize count
    while (count <= 10){
        cout << count << " ";
        count++;       // increment count
    }
    return 0;
}

The output of the above program:

1 2 3 4 5 6 7 8 9 10
In the above program, the loop incurs a *counter-controlled repetition*. Counter-controlled repetition requires:

1) the name of a control variable (the variable *count*)
2) the initial value of the control variable (*count* is initialized to 1 in this case)
3) the condition that tests for the final value of the control variable (i.e., whether looping should continue);
4) the *increment* (or *decrement*) by which the control variable is modified each time through the loop.
The *for* statement is used for repeating a statement or series of statements as long as a given conditional expression evaluates to true.

One of the main differences between *while* statement and *for* statement is that in addition to a condition, you can also include code in the *for* statement
- to initialize a counter variable and
- changes its value with each iteration

The syntax of the *for* statement:

```plaintext
for ( initialization expression; condition; update expression) statement
```
Enter the *for* statement

Initialization expression

- test the condition?
  - false
  - Execute the statement(s)
  - true
  - Exit the *for* statement

Execute the update statement
Example 5.4.1
// This program prints the even number from 2 to 20
#include <iostream.h>
int main()
{
    int count;
    for (count = 2; count <= 20; count = count + 2)
        cout << count << " ";
    return 0;
}

The output of the above program:

2  4  6  8  10 12  14  16  18  20
- **do..while** statement is used to create **post-test** loops.
- The syntax:
  ```
  do {
  statements
  } while (conditional expression);
  ```
// This program prints the odd number from 1 to 19
#include <iostream.h>
int main()
{
    int count = 1;
    do {
        cout << count << " ";
        count += 2;
    } while (count < 20);
    return 0;
}

The output of the above program:

1 3 5 7 9 11 13 15 17 19
Exercise
Combining interactive data entry with a loop statement produces very adaptable and powerful programs.

Example 5.3.1

```c
#include <iostream.h>
int main()
{
    int total,        // sum of grades
        gradeCounter, // number of grades entered
        grade,        // one grade
        average;      // average of grades
    total = 0;
    gradeCounter = 1;                   // prepare to loop
    while (gradeCounter <= 10) {       // loop 10 times
        cout << "Enter grade: ";          // prompt for input
        cin >> grade;                     // input grade
        total = total + grade;            // add grade to total
        gradeCounter = gradeCounter + 1;  // increment counter
    }
}
```
// termination phase
average = total / 10;           // integer division
cout << "Class average is " << average << endl;
return 0;
}

The output of the above program:
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
Example of interactive *do while* loop

do {
    cout<< "\nEnter an identification number:”;
    cin >> idNum;
} while (idNum < 1000 | | idNum> 1999);

- Here, a request for a new id-number is repeated until a valid number is entered.
A refined version of the above program:

```c
    do {
        cout << "Enter an identification number:";
        cin >> idNum;
        if (idNum < 1000 || idNum > 1999) {
            cout << "An invalid number was just entered\n";
            cout << "Please reenter an ID number\n";
        } else break;
    } while (true);
```
In many situations, it is convenient to use a loop contained within another loop. Such loops are called nested loops.

Example 5.4.1
#include <iostream.h>
int main()
{
    const int MAXI = 5;
    const int MAXJ = 4;
    int i, j;
    for(i = 1; i <= MAXI; i++) // start of outer loop
    {
        cout << endl;
        for(j = 1; j <= MAXJ; j++) // start of inner loop
            cout << j << endl;
    } // end of outer loop
cout << endl;
return 0;
}

The output of the above program:

i is now  1
  j = 1  j = 2  j = 3  j = 4
i is now  2
  j = 1  j = 2  j = 3  j = 4
i is now  3
  j = 1  j = 2  j = 3  j = 4
i is now  4
  j = 1  j = 2  j = 3  j = 4
i is now  5
  j = 1  j = 2  j = 3  j = 4
In programming, data values used to indicate either the start or end of a data series are called *sentinels*. The sentinels must be selected so as not to conflict with legitimate data values.

Example 5.3.2
#include <iostream.h>
const int HIGHGRADE = 100; // sentinel value
int main()
{
    float grade, total;
    grade = 0;
    total = 0;
    cout << endlTo stop entering grades, type in any number"
         "greater than 100." << endl
;
cout << "Enter a grade: ";
    cin >> grade;
    while (grade <= HIGHGRADE)
    {
        total = total + grade;
        cout << "Enter a grade: ";
        cin >> grade;
    }
    cout << "\nThe total of the grades is " << total << endl;
    return 0;
}

- In the above program, the sentinel is the value 100 for the entered grade.
The `break` statement causes an exit from the innermost enclosing loop.

Example:
```cpp
while (count <= 10)
{
    cout << "Enter a number: "; cin >> num;
    if (num > 76) {
        cout << "you lose!\n";
        break;
    }
    else
        cout << "Keep on trucking!\n";
    count++;
}
//break jumps to here
The `continue` statement halts a looping statement and restarts the loop with a new iteration.

```cpp
while(count < 30) {
    cout << "Enter a grade: ";
    cin >> grade;
    if (grade < 0 || grade > 100)
        continue;
    total = total + grade;
    count++;
}
```

In the above program, invalid grades are simply ignored and only valid grades are added to the total.
All statements must be terminated by a semicolon. A semicolon with nothing preceding it is also a valid statement, called the *null statement*. Thus, the statement

```
;
```

is a null statement.

**Example:**

```
if (a > 0)
  b = 7;
else ;
```

The null statement is a do–nothing statement.
Exercise

cuu duong than cong. com

cuu duong than cong. com