Learning Objectives

In this chapter you will:

- Learn about repetition (looping) control structures, i.e. while, for and do...while
- Explore how to construct and use (1) count-controlled, (2) sentinel-controlled, (3) flag-controlled, and (4) EOF-controlled repetition structures
- Examine break and continue statements
- Discover how to form and use nested control structures

Flow of Control (Part II):

Repetition while, for & do..while

Why Is Repetition Needed?

- **Repetition** (a.k.a. **Loop**)
  - allow you to efficiently use variables
  - can input, add, and average multiple numbers using a limited number of variables
- For example, to add five numbers:
  - declare a variable for each number, input the numbers and add the variables together
  - create a loop that reads a number into a variable and adds it to a variable that contains the sum of the numbers
A First Look of Loops

- 3 types of loop structures in C++
  1. while
     - most flexible
     - no restrictions
  2. do-while
     - least flexible
     - always execute loop body at least once
  3. for
     - natural counting loop

while Loop Example

- Example:
  [...]
  count = 0;    // Initialization
  while (count < 3) // Loop Condition
  {
    cout << "Hi "; // Loop Body
    count++;       // Update expression
  }
  [...]
  - Loop body executes how many times?

do-while Loop Example

- Example:
  [...]
  count = 0;    // Initialization
  do
  {
    cout << "Hi "; // Loop Body
    count++;       // Update expression
  } while (count < 3); // Loop Condition
  [...]
  - Loop body executes how many times?
  - do-while loops always execute body at least once!

for Loop Example

- Example:
  for (count=0; count<3; count++)
  {
    cout << "Hi "; // Loop Body
  }

  - How many times does loop body execute?
  - Initialization, loop condition and update are all built into the for-loop structure!
  - A natural "counting" loop
### The while Loop (1/2)

- Syntax of the while statement is:
  
  ```
  while (entry_cond) 
  statement_while;
  ```

- Statement `statement_while` can be either simple or compound ⇒ body of the loop

- Expression `entry_cond` acts as a decision maker and is usually a Boolean expression

- The parentheses (...) are part of the syntax

### The while Loop (2/2)

- Expression `entry_cond` provides an entry condition

- Statement `statement_while` executes if the expression initially evaluates to true

- Loop condition `entry_cond` is then reevaluated

- Statement continues to execute until the expression `entry_cond` is no longer true

- Infinite loop: continues to execute endlessly – can be avoided by including statements in the loop body that assure exit condition will eventually be true

### while Loop Example (1/2)

**Example:**

```cpp
... idx = 0;
while (idx < 10) {
    cout << 2*idx << " ";
    idx += 2; // idx = idx + 2
} cout << endl;
... Sample Run:
> ./a.out
0 4 8 12 16
```

### while Loop Example (2/2)

```cpp
... idx = 20;
while (idx < 20) //never satisfy entry_cond
{
    cout << idx << " ";
    idx += 5;
}
cout << endl;
...
```

**Sample Run:**

```
> ./a.out
0 4 8 12 16
```
**while (1): Counter-Controlled**

- If you know exactly how many pieces of data need to be read, the while loop becomes a counter-controlled loop
- Example of counter-controlled loop:
  ```
  counter = 0; // initialize loop control
  while (counter < Limit) // test loop control
  {
      statements;
  ...
  counter++; // update loop control
  }
  ```

**while (2): Sentinel-Controlled**

- Sentinel variable is tested in the condition and loop ends when sentinel is encountered
- Example of sentinel-controlled loop:
  ```
  cin >> target; // initialize loop control
  while (target != sentinel) // test loop control
  {
      statements;
  ...
  }
  ```

**while (3): Flag-Controlled**

- A flag-controlled while loop uses a bool variable to control the loop
- Example of flag-controlled loop:
  ```
  loop_stop = false; // initialize loop control
  while (!loop_stop) // test loop control
  {
      ...
      if (expression) // update loop control
      loop_stop = true;
      ...
  }
  ```

**while (4): EOF-Controlled**

- An EOF-controlled while loop uses a EOF (End-of-File) to control the loop
- The logical value returned by `cin` can determine if the program has ended input
- Example of EOF-controlled loop:
  ```
  cin >> var; // initialize loop control
  while (cin) // test loop control
  {
      ...
      if (cin.eof())
      {
          // cin does not reach EOF
          Or use while (!cin.eof())
      }
      ...
  }
  ```
**The eof Function**

- When a program is reading from a disk file, `eof` can determine the end of file status
  - When reading from input devices like keyboard, use `ctrl+z` or `ctrl+d` denotes `eof`
- `eof` is a member of data type `iostream`
  - `iostream` includes header files of `istream`, `ostream` and others
- Syntax for the function `eof` is: `istreamObj.eof()`
  - `istreamObj` is an input stream variable, such as `cin`
- Example:
  ```cpp
  cin.eof()
  ```
  ```cpp
  cin.clear();  // to reset eof and restart reading from input devices
  ```

**while Pitfalls: Misplaced ;**

- Watch the misplaced `;` (semicolon)
  ```cpp
  while (response!= 0):
  {
    cout << "Enter val: ";
    cin >> response;
  }
  ```
  - Notice the `;` after the while condition!
- Result here: INFINITE LOOP!

**while Pitfalls: Infinite Loops**

- Loop condition must evaluate to false at some iteration through loop
  - If not ⇒ infinite loop
- Example:
  ```cpp
  while (1) {
    cout << "Hello ";
  }
  ```
  - a perfectly legal C++ loop ⇒ always infinite!
- Sometimes, infinite loops can be desirable
  - e.g., "Embedded Systems"

**The for Loop**

- Syntax of `for` statement:
  ```cpp
  for (init_stmt;lp_cond;update_stmt) statement_for;
  ```
  - `init_stmt`: initialize ctrl variable
  - `lp_cond`: compare ctrl variable
  - `update_stmt`: update ctrl variable
- Action statement `statement_for` can be a compound statement
  - much more typical
for versus while (1/2)

- Rewrite for statement in while loop:

  ```c
  init_stmt;
  while (lp_cond) {
    stmt_for;
    update_stmt;
  }
  ```

- Rewrite while statement in for loop:

  ```c
  for ( ; entry_cond; ) {
    stmt_while;
  }
  ```

for versus while (2/2)

- Components of for statement are optional but semicolons ";" must always be present
  - omissions ⇒ doing nothing in init_stmt and update_stmt or being true in lp_cond
  - EX: `for ( ; ; ) { statements; }` is valid

- Differentiate for from while:
  - *initialization statements* grouped as first set of items init_stmt within the for’s ()
  - *loop condition* and loop statements are fixed: no change in for
  - *update statements* as the last set of items update_stmt within the for’s ()

for Loop Example 1 (1/3)

Example 1:

```c
...for (count=2; count<=20; count+=2) {
  cout << count << " ";
}
...
```

Modified Example 1a:

```c
...count = 2;
for ( ; count<=20; count+=2) {
  cout << count << " ";
}
...
```

Sample Run:

```
>./a.out
  2 4 6 8 10 12 14 16 18 20
```

for Loop Example 1 (2/3)

Example 1:

```c
...for (count=2; count<=20; count+=2) {
  cout << count << " ";
}
...
```

Modified Example 1b:

```c
...count = 2;
for ( ; count<=20; ) {
  cout << count << " ";
  count +=2;
}
...
```
for Loop Example 1 (3/3)

Example 1:
```cpp
for (count=2; count<=20; count+=2)
    { cout << count << " "; }
```
Modified Example 1c:
```cpp
    count = 2;
for (count=2; count<=20; count+=2)
    { cout << count << " "; }
```

for Loop Example 2 (1/2)

Example 2:
```cpp
for (idx=1; idx<=5; idx++)
    { cout << "Hello!" << endl;
      cout << "*" << endl;
    }
```
Example 2a:
```cpp
for (idx=1; idx<=5; idx++)
    cout << "Hello!" << endl;
    cout << "*" << endl;
```
What will happen?

for Loop Example 2 (2/2)

Example 2:
```cpp
for (idx=1; idx<=5; idx++)
    { cout << "Hello!" << endl;
      cout << "*" << endl;
    }
```
Example 2b:
```cpp
for (idx=1; idx<=5; idx++)
    { cout << "Hello!" << endl;
      cout << "*" << endl;
    }
```
What will happen?

Comments on for loop

- If loop condition lp_cond is initially false ⇒ do nothing
  - Ex: for (idx=0; idx > 0; idx++)
- Update statement update_stmt eventually sets the value of lp_cond to false ⇒ otherwise, this for loop will run forever.
  - Ex: for (idx=0; idx<100; idx--)
- If ctrl variable is float/double, then different computers may yield different results on ctrl variable ⇒ should avoid using such variable.
  - Ex: for (double idx=0.1; idx<1.2; idx+=0.05)
The do…while Loop (1/2)

- Syntax of do…while statement:
  
  ```
  do
      stmt_dowhile;
  while (rep_cond);
  ```

- Statement stmt_dowhile executes first, and then rep_cond is evaluated

- If rep_cond evaluates to true, the statement stmt_dowhile runs again

The do…while Loop (2/2)

- As long as rep_cond in a do…while statement is true, the statement stmt_dowhile executes

- To avoid an infinite loop, the loop body stmt_dowhile must contain a update statement that makes rep_cond to be false

- The statement stmt_dowhile can be simple or compound

- If compound, it must be in braces `{ ... }`

- do…while loop typically has an exit condition and iterates at least once (unlike for and while)

do…while Loop Example

Example 1:

```
... idx = 1;
do {
   cout << idx << "  ";
   idx *= 2;
} while (idx <= 20);
...
```

Answer: 1 2 4 8 16

do…while Loop vs while Loop

Example 2a:

```
... idx = 15;
do {
   idx %= 8;
} while (idx > 15);
cout << idx << "  ";
...
```

Answer: 7

Example 2b:

```
... idx = 15;
do {
   idx %= 8;
} while (idx > 15);
cout << idx << "  ";
...
```

Answer: 15
break & continue in Repetition

- Flow of Control
  - Recall how loops provide "graceful" and clear flow of control in and out
  - In RARE instances, can alter natural flow
- break
  - force the loop to exit immediately.
- continue
  - skip the rest of loop body
- These statements violate natural flow
  - only used when absolutely necessary!

break in Loop

- break: forces immediate exits from structures:
  - in switch statements:
    ⇒ the desired case is detected/processed
  - in while, for and do...while statements:
    ⇒ an unusual condition is detected
- Example:
  ```c++
  for (idx=10; idx<=50; idx+=2) {
    if (idx%9 == 0)
      break; cout << idx << “ “ ;
  }
  ```
  What will be displayed on screen?
  10 12 14 16 ←

continue in Loop

- continue: cause the next iteration of the loop to begin immediately
  - execution transferred to the top of the loop
  - apply only to while, for and do...while statements
- Example:
  ```c++
  idx = 0; while (idx < 100) {
    idx++; if (idx == 50) continue; cout << idx << endl;
  }
  ```

Nested Loops (1/2)

- A loop contained within another loop
- Example: Print 9x9 multiples
  ```c++
  for (idx=1; idx<=9; idx++) //outer loop
  {
    cout << idx << “-multiples: " << endl;
    for (jdx=1; jdx<=9; jdx++) //inner loop
    {
      cout << idx*jdx << “ “ ;
    }
    // end of inner loop
  }
  // end of outer loop
  ```

What will be displayed on screen?
1 2 48 49 51 52...98 99
Nested Loops (2/2)

- **Outer** (first) Loop:
  - controlled by value of `idx`
- **Inner** (second) Loop:
  - controlled by value of `jdx`
- **Rules:**
  - For each single trip through outer loop, inner loop runs through its entire sequence
  - Different variables to control each loop
  - Inner loop statements contained within outer loop

Summary (1/3)

- C++ has three looping (repetition) structures: `while`, `for` and `do...while`
- `while`, `for` and `do...while` are reserved words
- `while` and `for` loops are called pre-test loops
- `do...while` loop is called a post-test loop
- `while` and `for` may not execute at all, but `do...while` always executes at least once

Summary (2/3)

- **while**: expression is the decision maker, and the statement is the body of the loop
- In a counter-controlled while loop,
  - Initialize counter before loop
  - Body must contain a statement that changes the value of the counter variable
- A sentinel-controlled `while` loop uses a sentinel to control the `while` loop
- An EOF-controlled `while` loop executes until the program detects the end-of-file marker

Summary (3/3)

- **for**: simplifies the writing of a count-controlled while loop
- Executing a `break` statement in the body of a loop immediately terminates the loop
- Executing a `continue` statement in the body of a loop skips to the next iteration
- After a `continue` statement executes in a `for` loop, the update statement is the next statement executed