Programming in C/C++

Chapter 5 – Repetition/Loops

Kristina Shroyer

Objectives

You should be able to describe:

- The **`while`** Loop
  - **`cin`** within a **`while`** Loop
  - Sentinels
  - Input validation
  - We will start by using while loops as count controlled loops even though for loops are best for count controlled loops
- The **`do while`** Loop
  - Menus
  - Input Validation
- The **`for`** Loop
  - Count Controlled loops
- Common Programming Errors

Increment and Decrement Operators ++ --

- Increment and Decrement Operators are **unary** arithmetic operators that add and subtract 1 from their operands

- **Incrementing a Value in C++**
  - We have already gone over the first two ways of incrementing a value by 1 in C++:
    - `number = number + 1;`
    - `number += 1;`

  - The third way of incrementing a value by 1 in C++:
    - `number++;`
      - Shortcut Operator

- **Decrementing a Value in C++**
  - `number = number - 1;`
  - `number -= 1;`
  - `number--;`

- **Often used in loops/Repetition:**
  - **Has the form**: `variable = variable + fixedNumber;`
  - **Counting**

Increment and Decrement Operators ++ --

- Example: (Example 1 - IncDec.cpp)

```
#include <iostream>
using namespace std;

int main()
{
    int number = 2;

    number++;
    cout << "After executing number++, number = " << number << endl;

    number--;
    cout << "After executing number--, number = "<< number << endl;

    system("PAUSE");
    return 0;
}
```

Increment & Decrement Operators

- Can be used in prefix mode (before) or postfix mode (after) a variable
  - **PostFix Mode:**
    - This means the operator (`--` or `++`) comes after the variable

```
number++;
number--;
```

  - **Prefix Mode:**
    - The increment decrement operator comes before the variable

```
++number;
--number;
```

  - Many times (but not all) both postfix and prefix mode produce the same result in your code
    - Next Example – Same program using Prefix Mode instead of Postfix

Increment and Decrement Operators ++ --

- Example: (Example 2 - IncDec2.cpp)

```
#include <iostream>
using namespace std;

int main()
{
    int number = 2;

    ++number;
    cout << "After executing ++number, number = " << number << endl;

    --number;
    cout << "After executing --number, number = "<< number << endl;

    system("PAUSE");
    return 0;
}
```

Postfix and Prefix Mode: ++ --

- **So Why have both Postfix and Prefix Mode? What's the difference?**
- **Prefix and Postfix Mode produce different results when the -- and ++ operators are used in combination with other operators**
- Let's look at an Example:
  - `int value = 1;`
  - `cout << value++;`
    - The second statement does two things
      - It increments value
      - It prints value
    - The question here is what happens first in the second statement?
      - Does value get incremented before it displays on the screen?
        - In this case value would display as 2
      - Or Does value display on the screen before it gets incremented?
        - In this case value would display as 1
    - **The answer depends on if the increment operator was used in Postfix or Prefix mode**
- **Rule1:** When a **Postfix** operator is used in combination with other operators (or expressions, like `cout`, the other expressions are evaluated or executed BEFORE the increment or decrement operator)
  - So in the above Postfix Example what would print on the screen?

Postfix and Prefix Mode: ++ --

- **Prefix and Postfix Mode produce different results only when the – and ++ operators are used in combination with other operators**

- So let's change our example so the increment operator is in Prefix mode
  - `int value = 1;`
  - `cout << ++value;`

- **Rule 2:** When a **Prefix** operator is used in combination with other operators (or expressions, like `cout`, the other expressions are evaluated or executed AFTER the increment or decrement operator)

  - So in the above Prefix Example what would print on the screen?

- Let's look at some examples

Postfix and Prefix Mode: ++ --

- Example: Postfix Operator with `cout` (Example 3 - Postfix1.cpp)

```
#include <iostream>
using namespace std;

int main()
{
    int number = 2;

    cout << "Postfix cout Example" << endl;
    cout << number++ << endl;

    system("PAUSE");
    return 0;
}
```

Postfix and Prefix Mode: ++ --

- Example: Prefix Operator with **cout** (Example 4 - Prefix1.cpp)

```
#include <iostream>
using namespace std;

int main()
{
    int number = 2;

    cout << "Prefix cout Example" << endl;
    cout << ++number << endl;

    system("PAUSE");
    return 0;
}
```

Postfix and Prefix Mode: ++ --

- Let's look at some more examples
- Example:
  - `int x = 1;`
  - `int number;`
  - `number = x--;`

  - What is the value of number?

- **Example Program: (Example 5 - Postfix2.cpp)**

```
#include <iostream>
using namespace std;

int main()
{
    int x = 1;
    int number;
    number = x--;

    cout << "Postfix Assignment Example" << endl;
    cout << "number = " << number << endl;

    system("PAUSE");
    return 0;
}
```

Postfix and Prefix Mode: ++ --

- Let's look at some more examples
- Example:
  - `int x = 1;`
  - `int number;`
  - `number = --x;`

  - What is the value of number?

- **Example Program: (Example 6 - Prefix2.cpp)**

```
#include <iostream>
using namespace std;

int main()
{
    int x = 1;
    int number;
    number = --x;

    cout << "Prefix Assignment Example" << endl;
    cout << "number = " << number << endl;

    system("PAUSE");
    return 0;
}
```

Postfix and Prefix Mode: ++ --

- Why do they have this differentiation between Postfix and Prefix?
- When would you ever use it?

Repetition - Loops

- Loops are control structures that allow repetition to be used in programs
  - A loop causes a statement or group of statements to repeat

- C++ has 3 looping structures we will look at in detail
  - The **`while`** loop
  - The **`do-while`** loop
  - The **`for`** loop

The `while` Statement

- A general repetition statement
- Format:

```
while (expression)
{
    statement(s);
}
```

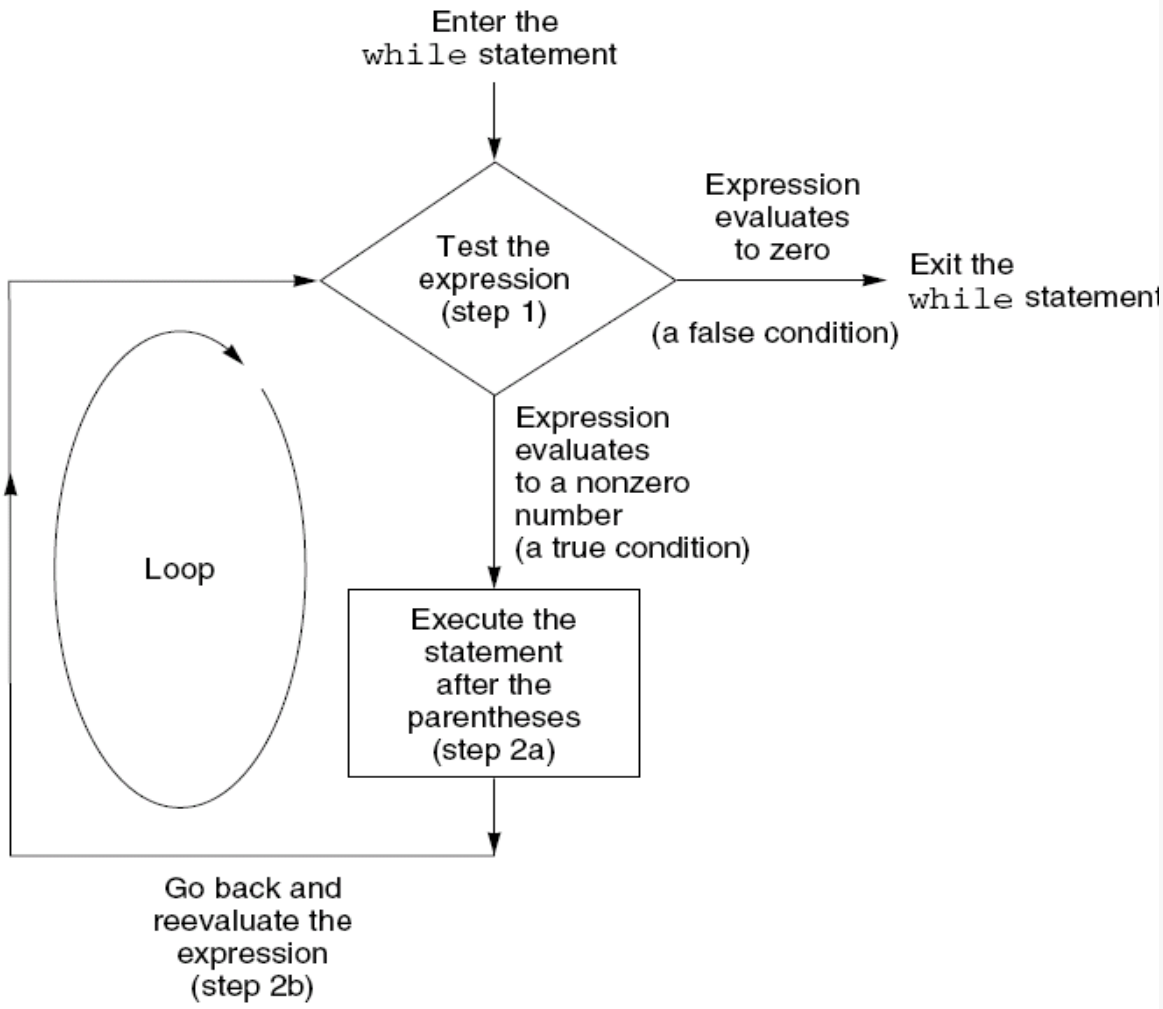
- Steps in execution of **`while`** statement:
  1. Test the expression
  2. If the expression has a nonzero (true) value
     - Execute the statement(s) following the curly braces
     - Go back to step 1

     otherwise
     - Exit the **`while`** statement

- The while loop is a **pre-test** loop
  - It's possible that its statements will never get executed
  - Why?

The `while` Statement

FIGURE 5.1 *Anatomy of a* `while` *Loop*



The `while` Statement - Example

- **Example (Example 7 - While1.cpp)**

```
#include <iostream>
using namespace std;

int main()
{
   int value = 5;
   while (value >= 0)
   {
     cout << value << " ";
     value--;
   }

   cout << '\n';
   system("PAUSE");
   return 0;
}
```

- How many times will this loop execute?
- This is an example of a **count controlled loop**
  - What does that mean?
    - Counter, max/min value, condition comparing the counter and max value
  - Note that inside the loop you must have some way of eventually making the condition false so the loop eventually ends, I used the `value--` to do that
  - While loops are usually not used for count controlled loops
- The while loop is a **pretest loop**
  - It is possible that a pretest loop is never executed
  - How could we change this example so the loop is never executed?

The `while` Statement - Example

- **Example (Example 8 - While1a.cpp)**

```
#include <iostream>
using namespace std;

int main()
{
   int count = 1;

   while(count <= 10)
   {
      cout << count << " ";
      count++;
   }

   cout << endl;
   system("PAUSE");
   return 0;
}
```

- How many times will this loop execute?
- This is a more traditional example of a **count controlled loop**
  1. What is the max value?
  2. What is the counter?
  3. What's the condition?
- ***It's easy to accidentally make a loop run infinitely***
  - How could we change this loop to make it run infinitely?
    - There are a couple of mistakes that could make this happen

The `while` Statement - Example

- **Example (Example 9 - While2.cpp)**

```
// This program displays the numbers 1 through 10 and their
//squares.

#include <iostream>
using namespace std;

int main()
{
   int num = 1; // Initialize counter

   while (num <= 10)
   {
       cout << num << " " << (num * num) << endl;
       num++; // Increment counter
    }

    system("PAUSE");
    return 0;
}
```

- How could we change this example so that the user controls how many squares are printed?
  - Have the user control the maximum times the loop will be executed

The `while` Statement - Example

- **Example (Example 10 - While2a.cpp)**

```
// This program displays the numbers 1 through 10 and their squares.
#include <iostream>
using namespace std;

int main()
{
   int num = 1; // Initialize counter
   int numSquares;

   cout << "How many squares would you like to print?";
   cin >> numSquares;

   while (num <= numSquares)
   {
      cout << num << " " << (num * num) << endl;
      num++; // Increment counter
   }

   system("PAUSE");
   return 0;
}
```

The `while` Statement

- **Counter:** variable that is incremented or decremented each time a count controlled loop repeats
  - Can be used to control execution of the loop (count (loop) control variable)
  - Must be initialized before entering loop
  - May be incremented/decremented either inside the loop or in the loop test
- **Fixed-count Loop:** tested expression is a counter that checks for a fixed number of repetitions
- **Variation:** Counter is incremented by a value other than 1

The `while` Statement

- Example: Counter Increment Other than 1 (Example 11 - While3.cpp)

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{

   double celsius = 5;           // starting Celsius value
   double fahren;

   while (celsius <= 50)
   {
       fahren = (9.0/5.0) * celsius + 32.0;
       cout << setw(4) << celsius
               << setw(13) << fahren << endl;
       celsius = celsius + 5;
   }

   system("PAUSE");
   return 0;
}
```

The `while` Loop – Pretest Loop

- **`while` loop format review:**
  - Loop: part of program that may execute > 1 times (i.e., it repeats)
  - **while** loop format:

```
while (expression)
{ statement(s);
}
```

  - If there is only one statement in the body of the loop, the `{}` can be omitted

- **`while`** is a pretest loop (***expression*** is evaluated before the loop executes)
  - If the expression is initially false, the statement(s) in the body of the loop are never executed
  - If the expression is initially true, the statement(s) in the body continue to be executed until the expression becomes false

The `while` Loop – Exiting the Loop

- The loop must contain code to allow ***expression*** to eventually become **false (evaluate to zero)** so the loop can be exited
- Otherwise, you have an infinite loop (i.e., a loop that does not stop)
  - Example infinite loop:

```
x = 5;
while (x > 0) //infinite loop because x is always > 0
   cout << x;
```

  - **CURLY BRACES ARE VERY IMPORTANT**
    - **Forgetting them can cause an infinite loop**

```
x = 5;
while (x > 0) //infinite loop because x is always > 0
   cout << x;
   x--;
```

    - **This is still an infinite loop**

Using the `while` Loop to Keep a Running Total

- Many programming tasks require you to calculate the total of a series of numbers provided as input
  - A number is entered by the user
  - An accumulating statement adds the number the user entered to total

```
total = total + num;
```

  - A **`while`** statement repeats the process

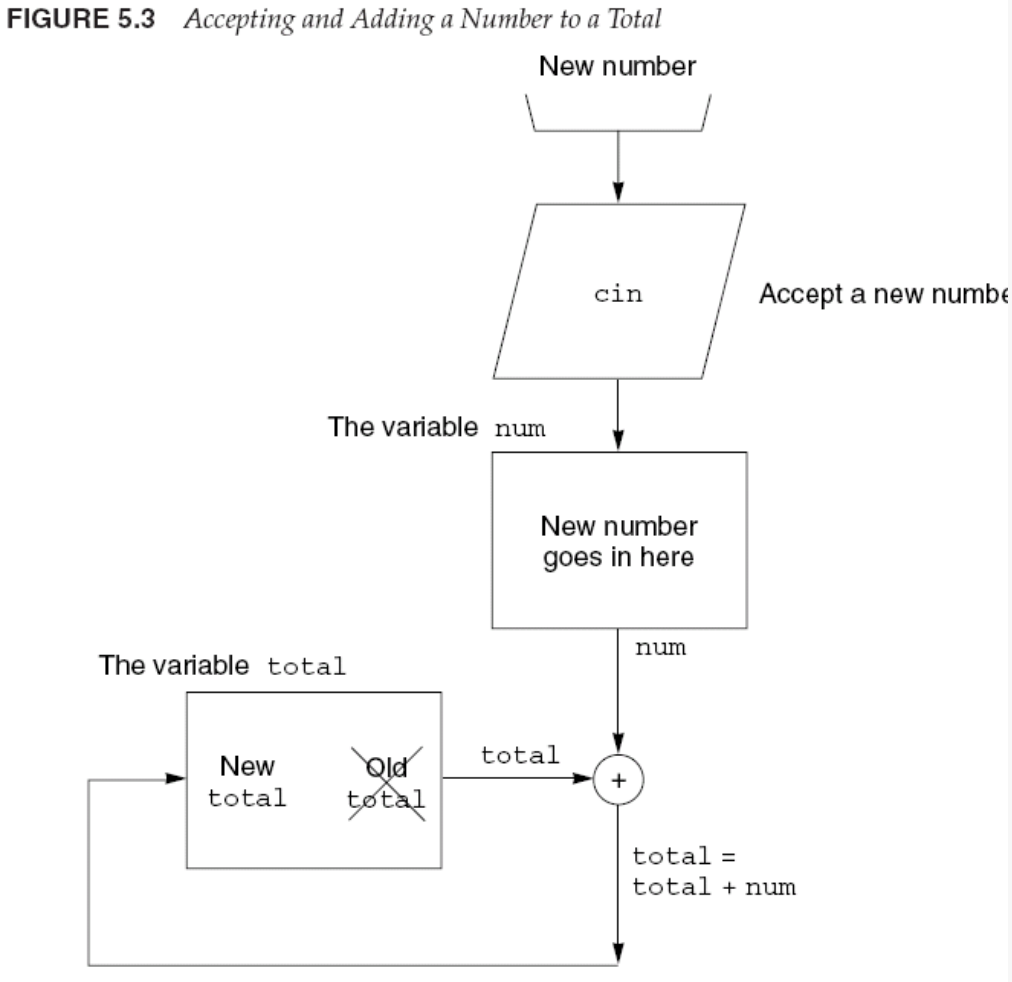
- Definitions
  - **running total:** accumulated sum of numbers from each repetition of loop
  - **accumulator:** variable that holds running total

Using the `while` Loop to Keep a Running Total

- **Example (Example 12 - WhileEx4Accum.cpp)**

```
#include <iostream>
using namespace std;

int main()
{
   const int MAXNUMS = 4;
   int count = 1;
   double num;
   double total = 0;  //initialize the accumulator

   cout << "This program will ask you to enter " << MAXNUMS << " numbers.\n";

   while(count <= MAXNUMS)
   {
      cout << "\n\nEnter a number: ";
      cin >> num;
      total += num;  //same as total = total + num;
      cout << "The total is now " << total;
      count++;
   }

   cout << '\n';

   system("PAUSE");
   return 0;
}
```

`cin` within a `while` Loop to Keep a Running Total



`break` and `continue` statements

- **`break:`** forces immediate exit from structures:
  - Used in **`switch`** statements:
    - The desired case has been detected and processed
  - ***Rarely (NEVER IN THIS CLASS)*** used in **`while`**, **`for`** and **`do-while`** statements:
    - ***An unusual condition is detected – must be very unusual***
    - **Use sparingly if at all – makes code harder to understand**
      - ***DO NOT USE BREAK IF YOU CAN INSTEAD USE LOGIC TO EXIT A LOOP – THERE IS NO REASON TO USE BREAK IN THIS CLASS OTHER THAN IN switch STATEMENTS (points off!!)***
      - **In this class you should *NEVER* NEED A BREAK in a loop**
      - **DO NOT MAKE INFINITE LOOPS and "break" out, use logic whenever possible to increase readability**
  - When used in an inner loop, terminates that loop only and goes back to outer loop
- Format:

```
break;
```

- **`continue;`** causes the next iteration of the loop to begin immediately
  - Execution transferred to the top of the loop
  - Applies only to **`while`**, **`do-while`** and **`for`** statements
    - **while** and **do-while** loops go to test and repeat the loop if test condition is true
    - **for** loop goes to update step, then tests, and repeats loop if test condition is true
  - **Use sparingly if at all – makes code harder to understand**
    - ***DO NOT USE CONTINUE WHEN LOGIC CAN BE USED (there is no reason to use continue in this class – points off!!)***
    - **In this class you should *NEVER* NEED A CONTINUE**
- **Format:**

```
continue;
```

The `while` Loop - Sentinels

- All of the loops we have done so far have required that we know how many times we want our loop to execute (how many repetitions we want it to have)
  - Sometimes the user may not know how many times they want the loop to iterate
    - Maybe there is a potentially very long list of data to input and the exact number of input items isn't known
  - In this case instead of using a counter to terminate the while loop, a **sentinel** can be used

- ***`while` loops in general are not used for count controlled loops because we have a better construct for that, the `for` loop***
  - **`while`** loops instead are going to be used with condition controlled loops, usually done with sentinels

- ***Sentinel:*** a special value in a list of values that indicates end of data
  - This special value that cannot be confused with a valid value,
    - *e.g.*, **-999** for a test score would work as a sentinel since a test score cant be negative
  - Used to terminate input when user may not know how many values will be entered

Sentinels - Example

- **Example : (Example 13 - Sentinel.cpp)**

```
#include <iostream>
using namespace std;

int main()
{
   int points;
   int total = 0;

   cout << "Enter points earned "
        << "(or -1 to quit): ";
   cin >> points;

   while (points != -1) // -1 is the sentinel
   {
      total += points;
      cout << "Enter points earned: ";
      cin >> points;
   }

   cout << "The total points earned are " << total << endl;

   system("PAUSE");
   return 0;
}
```

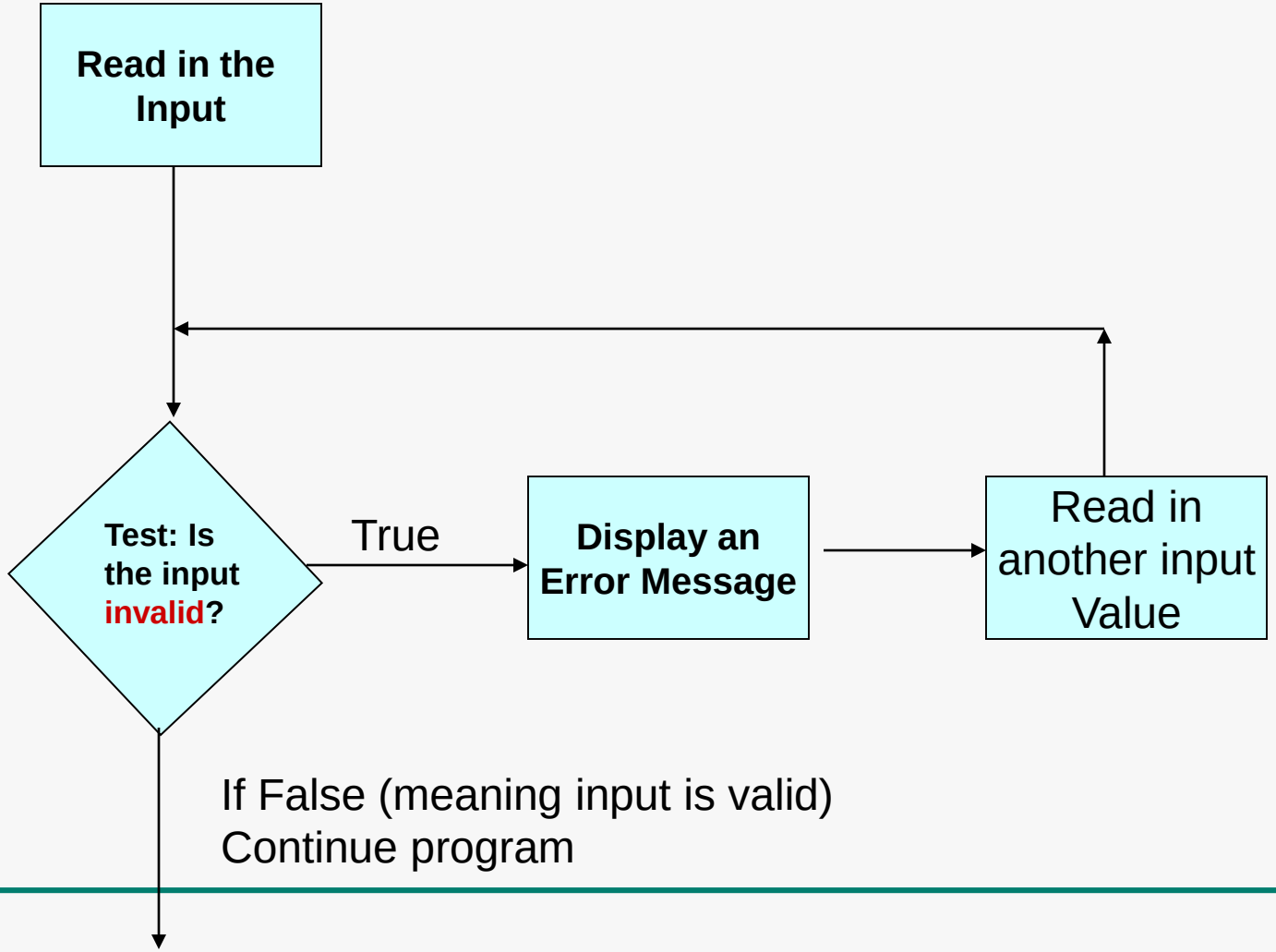
The `while` Loop – Input Validation

- You can use the while loop for ***input validation*** in a program
  - This is another example of a loop not using a counter but waiting for the user to enter a specific value (a sentinel) to terminate the loop

- How it works:
  - The *BooleanExpression* in the while loop will check for *invalid* data entry
    - If the *BooleanExpression* evaluates to **false (zero)**, the while loop will notify the user and ask them to re-enter the data so the loop can test if the new data they entered was valid
    - This type of while loop keeps executing until the user enters valid data

The `while` Loop – Input Validation

- **Logic of the While Loop When Used for Input Validation**



The `while` Loop – Input Validation

- **Example: (Example 14 - InputValid1.cpp)**

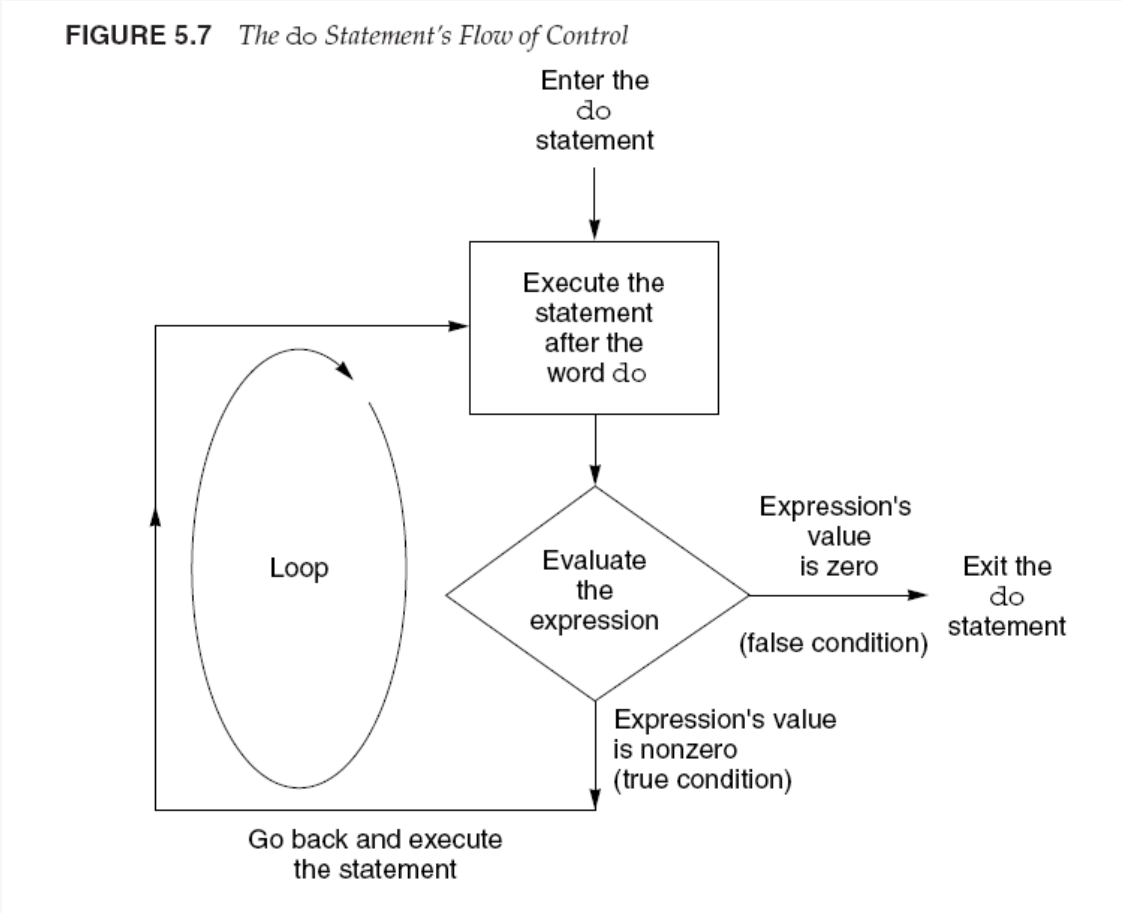
```
#include <iostream>
using namespace std;

int main()
{
    int age;

    cout << "Please enter your age: ";
    cin >> age;

    //prompt the user for an age and only accept ages > 0
    //loop is a pre-test loop, will not enter if they enter a valid age the first time (this is ok
    //and intentional)
    //loop continues as long as invalid data is entered
    while(age <= 0)
    {
         cout << "The age you entered is INVALID. Try again." << endl;
         cout << "Please enter your age: ";
         cin >> age;
    }

    cout << "Your age is " << age << endl;

    system("PAUSE");
    return 0;
}
```

The `do-while` Loop

- **do-while** is a repetition statement that evaluates an expression at the end of the statement
  - Allows some statements to be executed before an expression is evaluated
- **do-while**: a posttest loop (***expression*** is evaluated after the loop executes)
  - `for` and `while` loops evaluate an expression at the beginning of the statement (pretest loops)
  - **This allows you to have an while statement that always executes at least once**
- **Format:**

```
do
{
   statement(s);
}while (expression); // don't forget final ;
```

The `do-while` Loop



The `do-while` Loop : Notes

- Loop always executes at least once
- Execution continues as long as ***expression*** is **true**;
  - the loop is exited when ***expression*** becomes **false**
- Useful in menu-driven programs to bring user back to menu to make another choice
- Can be used for validity checks input

The do-while Loop : Menu - Example 15 – menuProg.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    char selection;
    int numMilky = 0;
    int numSnick = 0;
    int numSkit = 0;
    int numStar = 0;

    do
    {
          cout << endl;
          cout<< "CANDY MACHINE MENU" << endl;
          cout << "a) Milky Way" << endl;
          cout << "b) Snickers" << endl;
          cout << "c) Skittles"  << endl;
          cout << "d) Starburst" << endl;
          cout << "p) Print Daily Report" << endl;
          cout << "q) Quit Program" << endl;
          cout << "Please enter your selection: ";
          cin >> selection;

          //you could also use a switch statement here
          if(selection == 'a')
          {
             numMilky++;
             cout << "A Milky Way was purchased" << endl;
          }

          else if(selection == 'b')
          {
             numSnick++;
             cout << "A Snickers was purchased" << endl;
          }
```

NOTE: I often see variations of this with a lot of EXTRA LOOPS in the code, extra loops cause your code to be less efficient and your algorithm to be slower, you should NEVER use an EXTRA LOOP when a selection statement can be used to logically exit the one loop – notice I only have one loop here!!!

Always make your logic as simple as possible – simple programs > complex ones

Let's redo this code so the user can enter uppercase or lowercase letters as menu choices

• **Continued on next slide**

The do-while Loop : Menu Example 15

- **Continued from previous slide**

```
      else if(selection == 'c')
       {
          numSkit++;
          cout << "A bag of Skittles was purchased" << endl;
       }

       else if(selection == 'd')
       {
          numStar++;
          cout << "A package of Starburst was purchased" << endl;
       }

       else if(selection == 'p')
       {
          cout << endl;
          cout << "STATISTICS REPORT" << endl << endl;
          cout <<"CANDY          \t" << setw(15) << "Number Sold" << endl;
          cout <<"---------------\t" << setw(15) << "-----------" << endl;
          cout << "Milky Way\t" << setw(15) << numMilky << endl;
          cout << "Snickers\t" << setw(15) << numSnick << endl;
          cout << "Skittles\t" << setw(15) << numSkit << endl;
          cout << "Starburst\t" << setw(15) << numStar << endl;
       }

       else if(selection != 'q')
       {
          cout << "INVALID SELECTION" << endl;
       }

   }while(selection != 'q');

   system("PAUSE");
   return 0;
}
```

The `for` Loop

- Pretest loop that executes zero or more times
- **Designed for a *counter (count)-controlled loop***
  - **It's almost always better (*in this class it is ALWAYS better and required*) to use a for loop for a counter controlled loop than a while loop**
    - **This is very true when using loops to navigate arrays**
  - All of the components needed for a counter controlled look are in the first line of the for loop

- **Format:**

```
for( initialization; test; update )
{
  1 or more statements;
}
```

- **Components:**
  1. **Initializing list:** Initial value of expression (counter)
  2. **Expression:** a valid C++ expression (compares counter to a max or a min)
  3. **Update:** statements executed at end of each `for` loop to alter value of expression

The `for` Loop – Running Total Example

- **Example (Example 16 - ForLoopEx1.cpp)**

```
/*this for loops sums the numbers 1-10*/
#include <iostream>
using namespace std;

int main()
{
   int sum = 0; //accumulator

   for (int num = 1; num <= 10; num++) //num is the counter for the for loop
   {
      sum += num;
      cout << "Sum of numbers 1 - " << num << " is "
           << sum << endl;
   }

  system("PAUSE");
  return 0;
}
```

Question: What is the scope of `num`?

The `for` Loop

```
for(initialization; test; update)
{
      statement(s);
}
  // {} may be omitted if loop body contains only 1 statement
```

- **For loop steps:**

1. Perform ***initialization***
2. Evaluate ***test*** expression:
   - If **true (non zero)**, execute ***statement(s)***
   - If **false (zero)**, terminate loop execution
3. Execute ***update***, then re-evaluate ***test*** expression

The `for` Loop – Running Total Example

- **Example (Example 17 - ForLoopEx2.cpp)**
  - **Let the user control the maxNum**

```
/*this for loops sums the numbers from 1 to whatever the user input*/
#include <iostream>
using namespace std;

int main()
{
   int sum = 0; //accumulator
   int maxNum;

   cout << "Please enter the maximum number: ";
   cin >> maxNum;

   for (int num = 1; num <= maxNum; num++) //num is the counter for the for loop
   {
      sum += num;
      cout << "Sum of numbers 1 - " << num << " is "
           << sum << endl;
   }

   system("PAUSE");
   return 0;
}
```

The `for` Loop – Another Counting Loop

- **Example (Example 18 - ForLoopEx3.cpp)**

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    const int MAXNUMS = 10;

    //print the number, it's square and its cube, 1 up thru MAXNUMS
    cout << endl;
    cout << "NUMBER   SQUARE   CUBE" << endl;
    cout << "------   ------   ----" << endl;

    for(int num = 1; num <= MAXNUMS; num++)
    {
       cout << setw(3) << num << "      "
            << setw(4) << num * num << "      "
            << setw(4) << num * num * num << endl;
    }

    system("PAUSE");
    return 0;
}
```

`for` Loop Modifications

- Can define variables in initialization code
  – Their scope is the **for** loop
  - **Example:**

```
for (int num = 1; num <= 10; num++)
{
    sum += num;
}
```

  - Why define num like this?
    - It's useful scope really is only the for loop
    - You'll see most of my for loops are done like this

- Initialization code, test, or update code can contain more than one statement
  - Separate with commas
  - **Example:**

```
for (int sum = 0, num = 1; num <= 10; num++)
{
    sum += num;
}
```

    - **Now for this loop the scope of sum is probably more than just in the for loop so I wouldn't recommend this in most cases**

`for` Loop Modifications

- **THESE ARE NOT RECOMMENDED (DO NOT USE IN THIS CLASS!!)**
  - Can omit ***initialization*** if already done

```
int sum = 0;
int num = 1;
for(; num <=10; num++)
  sum += num;
```

- Can omit ***update*** if done in loop

```
for(sum = 0, num = 1; num <=10;)
  sum+ = num ++;  //this is confusing, BAD
                  //CODE, do not use
```

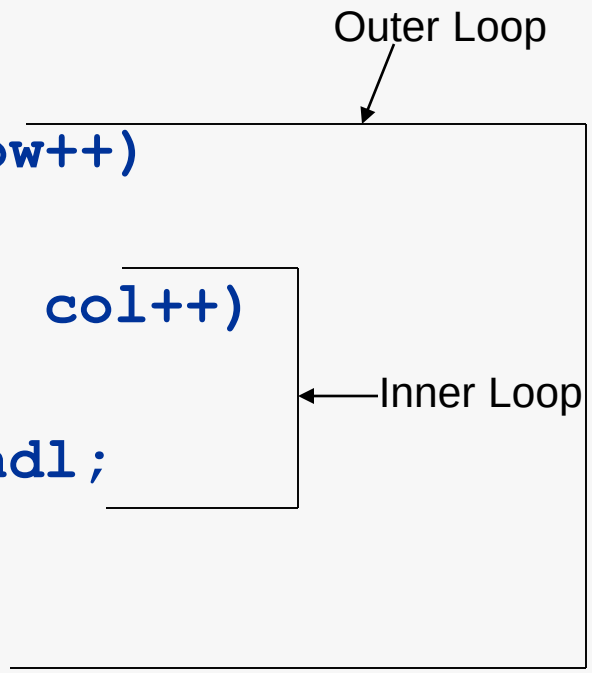
- Can omit ***test*** – may cause an infinite loop

```
for(sum = 0, num = 1;; num++)  //infinte loop
  sum += num;
```

Nested Loops

- A nested loop is a loop inside the body of another loop
  - ***Pay attention this is what part 2 of HW #3 is about***
- Example:

```
for (int row = 1; row <= 3; row++)
{
   for (int col = 1; col <= 3; col++)
   {
       cout << row * col << endl;
   }
}
```



Nested Loops

- Inner loop goes through all its repetitions for each repetition of outer loop
- Inner loop repetitions complete sooner than outer loop
- Total number of repetitions for inner loop is product of number of repetitions of the two loops. In previous example, inner loop repeats 9 times
- In the Previous Example:
  - **Outer (first) Loop:**
    - Controlled by value of `row`
  - **Inner (second) Loop:**
    - Controlled by value of `col`

Nested Loops

- Example (Example 19 - NestedLoop.cpp) – Another good one to look at for HW #3 Part #1

```
#include <iostream>
using namespace std;

int main()
{
   for (int i = 0; i < 4; i++)                  // outer loop
   {
      for (int j = 0; j < 2; j++)              // inner loop
      {
         cout << "i is " << i << " j is " << j << "\n";
      }
   }

   system("PAUSE");
   return 0;
}
```

Using `while` and `do-while` Loops for Data Validation

- **`while and do-while`** Loops are the most appropriate structure for validating user input data

- For a **`while`** loop:
  1. Prompt and read in the data.
  2. Use a pretest loop to test if data is invalid (not valid).
  3. Enter the loop only if data is not valid.
  4. Inside the loop, prompt the user to re-enter the data.
  5. The loop will not be exited until valid data has been entered.

Using `while` Loops for Data Validation - Example

```
cout << "Enter a number (1-100) and"
<< " I will guess it. ";
cin >> number;

while (number < 1 || number > 100)
{
   cout << "Number must be between 1 and 100."
   << " Re-enter your number. ";
   cin >> number;
}

// Code to use the valid number goes here.
```

Using `while` and `do-while` Loops for Data Validation

- **`while` and `do-while`** Loops are the most appropriate structure for validating user input data

- For a **`do-while`** loop:
  1. Enter the loop the first time – remember a do-while loop executes at least one time
  2. Prompt and read in the initial data.
  3. Inside the loop, have some sort of if statement to prompt the user to re-enter data if it is invalid
  4. Use a post test loop to see if the data is invalid
  5. Repeat the loop only if data is not valid.
  6. The loop will not be exited until valid data has been entered.

Using `do-while` Loops for Data Validation - Example

```
do{
   cout << "Enter a number (1-100) and"
        << " I will guess it. ";
   cin >> number;

   if (number < 1 || number > 100)
   {
      cout << "Number must be between 1 and 100."
           << " Re-enter your number. ";
      cin >> number;
   }
  } while (number < 1 || number > 100);

// Code to use the valid number goes here.
```

Deciding Which Loop to Use

- **`while`**: pretest loop (loop body may not be executed at all),
  - Sentinels (menus that repeat until a certain key is entered)
  - Input Validation
- **`do-while`**: posttest loop (loop body will always be executed at least once)
  - Input Validation
  - Menus that repeat (sentinels)
- **`for`**: pretest loop (loop body may not be executed at all); has initialization and update code;
  - is useful with counters or if precise number of repetitions is known

Common Programming Errors

"One-off" errors: loop executes one time too many or one time too few

- Initial and tested conditions to control loop must be carefully constructed

- Inadvertent use of assignment operator, = in place of the equality operator, ==
  - This error is not detected by the compiler

Common Programming Errors

- Using the equality operator when testing double-precision operands
  - Do not test expression **`(fnum == 0.01)`**
  - Replace by a test requiring absolute value of
    `(fnum - 0.01) < epsilon` for very small **`epsilon`**

- Placing a semicolon at end of the `for` statement parentheses:

```
for (count = 0; count < 10; count ++);
   total = total + num;
```

  - Creates a loop that executes 10 times and does nothing but increment **`count`**

Common Programming Errors

- Using commas instead of semicolons to separate items in a for statement:

```
for (count = 1, count < 10, count ++)
// incorrect
```

  - Commas should be used to separate items within the separating and initializing lists

- Omitting the final semicolon from the *do* statement

```
do
   statement;
while (expression); ← don't forget the final ;
```

Summary

- `while`, `for` and `do` statements create loops
  - These statements evaluate an expression
    - On the basis of the expression value, either terminate the loop or continue with it
  - Each pass through the loop is called a repetition or iteration

- `while` checks expression before any other statement in the loop
  - Variables in the tested expression must have values assigned before **`while`** is encountered

Summary

- The `for` statement: fixed-count loops
  - Included in parentheses at top of loop:
    - Initializing expressions
    - Tested expression
    - Expressions that affect the tested expression
    - Other loop statements can also be included as part of the altering list
- The `do` statement checks its expression at the end of the loop
  - Body of the loop must execute at least once
  - `do` loop must contain statement(s) that either:
    - Alter the tested expression's value or
    - Force a break from the loop