Topic 4

1. The `while` loop
2. Problem solving: hand-tracing
3. The `for` loop
4. The `do` loop
5. Processing input
6. Problem solving: storyboards
7. Common loop algorithms
8. Nested loops
9. Problem solving: solve a simpler problem first
10. Random numbers and simulations
11. Chapter summary

The `do{ } while()` Loop

The **`while()`** loop's condition test is the first thing that occurs in its execution.

The **`do`** loop (or **`do-while`** loop) has its condition tested only after at least one execution of the statements. The test is at the bottom of the loop:

```
do
{
   statements
}
while (condition);
```

The do Loop

This means that the **do** loop should be used only when the statements must be executed before there is any knowledge of the condition.

This also means that the **do** loop is the least used loop.

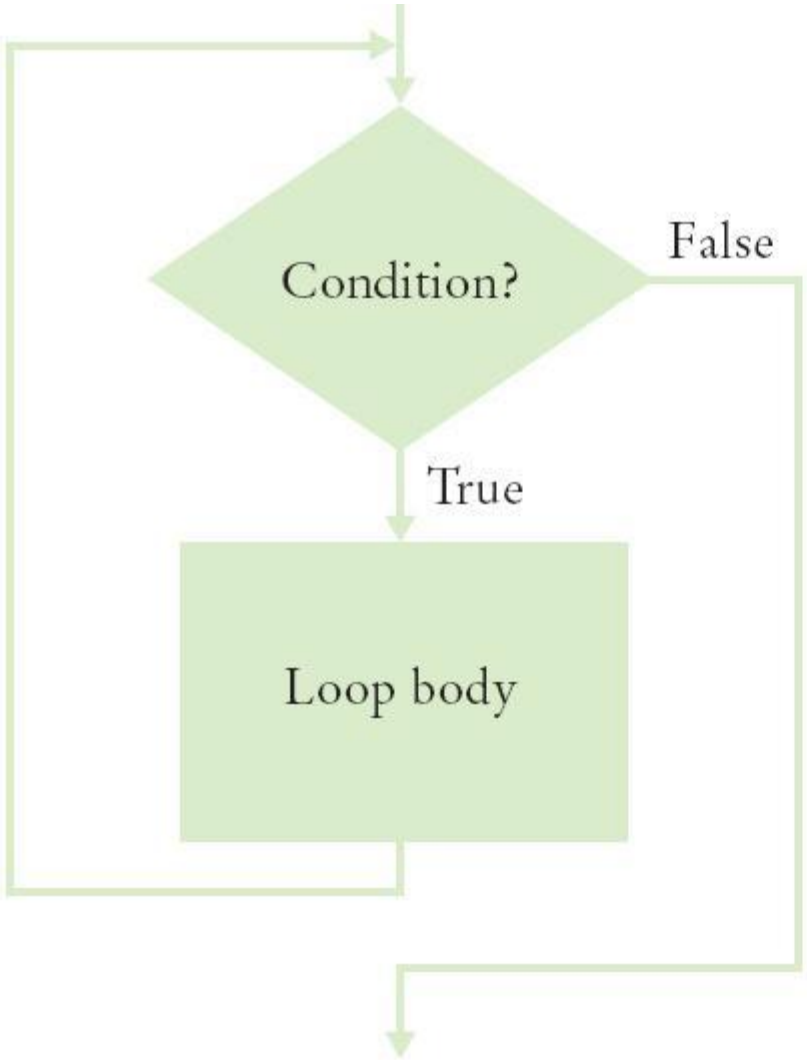
do { } Loop Code: getting user input Repeatedly

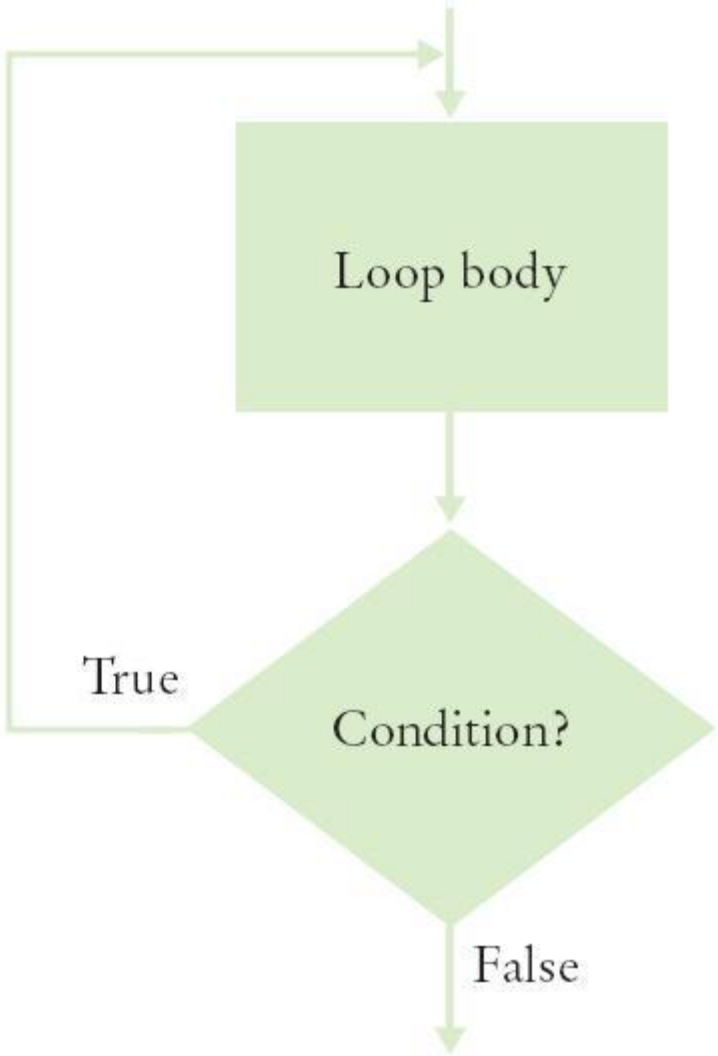
Code to keep asking a user for input until it satisfies a condition, such as non-negative for applying the `sqrt()`:

```
double value;
do
{
 cout << "Enter a number >= 0: ";
 cin >> value;
}
while (value < 0);

cout << "The square root is " << sqrt(value)
<< endl;
```

Flowcharts for the `while` Loop and the `do` Loop





Practice It: Example of `do...while`

- What output does this loop generate?

```
int j = 1;
do
{
   int value = j * 2;
   j++;
   cout << value << ", ";
} while (j <= 5);
```

Topic 5

1. The `while` loop
2. Problem solving: hand-tracing
3. The `for` loop
4. The `do` loop
5. Processing input
6. Problem solving: storyboards
7. Common loop algorithms
8. Nested loops
9. Problem solving: solve a simpler problem first
10. Random numbers and simulations
11. Chapter summary

Processing Input – When and/or How to Stop?

- We need to know, when getting input from a user, when they are done.

- One method is a **sentinel** (a *value* whose meaning is STOP!)
  - For example, when user is entering salary values, a negative number would indicate the end (since legitimate salaries cannot be negative)

Sentinel and a Salary Average Program (part 1)

ch04/sentinel.cpp

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

int main()
{
   double sum = 0;
   int count = 0;
   double salary = 0;
   // get all the inputs
   cout << "Enter salaries, -1 to finish: ";
   while (salary != -1)
   {
      cin >> salary;
      if (salary != -1)
      {
         sum = sum + salary;
         count++;
      }
   }
```

The Salary Average Program (part 2)

```
   // process and display the average
   if (count > 0)
   {
      double average = sum / count;
      cout << "Average salary: " << average << endl;
   }
   else
   {
      cout << "No data" << endl;
   }

   return 0;
}
```

A program run:

Enter salaries, -1 to finish: 10 10 40 -1
Average salary: 20

Using Failed Input for Processing

- Sometimes it is easier to ask the user to “Hit Q to Quit” instead of requiring the input of a sentinel value.
- In the previous chapter, we used **`cin.fail()`** to test if the most recent input failed.
- Note that if you intend to take more input from the keyboard after using failed input to end a loop, you must reset the keyboard with **`cin.clear()`**.
- Use a bool variable to keep track of the status, and use `cin.fail()` to test for the input of a **non-numeric** when expecting a number:

Code Example: Testing `cin.fail()`

```
cout << "Enter values, Q to quit: ";
int value;
bool more = true;
while (more)
{
   cin >> value;
   if (cin.fail())
   {
      more = false;
   }
   else
   {
      // process value here
   }
}
cin.clear(); // reset if more input needed
```

The Loop and a Half Problem

Those same programmers who dislike loops that are controlled by a `bool` variable have another reason: the actual test for loop termination is in the *middle* of the loop. Again it is not really a top or bottom test.

This is called a loop-and-a-half.

The Loop and a Half Problem and the `break` Statement

If we test for a failed read, we can stop the loop *at that point*:

```
while (true)
{
   cin >> value;
   if (cin.fail())
   { break; }
   // process value here
}
cin.clear() // reset if more input is to be taken
```

The **`break`** statement breaks out of the enclosing loop, independent of the loop condition.

Using Failed Input in the Loop Test

- Using a **`bool`** variable in this way is disliked by many programmers.

  Why?

- **`cin.fail`** is set *when* **`>>`** fails
  It is not really a top or bottom test.

  If only we could use the input itself to control the loop – we can!

- An input >> operation that does not succeed returns **`false`**, so it can be used in the **`while`**'s test.

Failed Input Loop Control – No `cin.fail()` needed

To avoid the need for `break` and testing `cin.fail`, you can use the input statement as the condition of the `while()` loop:

```
cout << "Enter values, Q to quit: ";
while (cin >> value)
{
   // process value here
}
cin.clear();
```

Redirection of Input and Output to Files

- To avoid having to type all the input to your program every time you re-test it, you can save the input in a text file, and run your program with "input redirection" via the < sign, as:

```
myprogram < myinput.txt
```

  - This assumes you have compiled an ".exe" file from your code called myprogram.exe, and have typed the above in a command line window
- Likewise, to store the output from your program, you can redirect it to a file instead of the screen by using >

```
myprogram > myoutput.txt
```

- And you can do both input and output from files:

```
myprogram < myinput.txt > myoutput.txt
```