Control Flow (order of execution)

• So far, control flow in our programs has included:
  ‣ sequential processing (1st statement, then 2nd statement…)
  ‣ branching (conditionally skip some statements).

• Chapter 5 introduces loops, which allow us to conditionally repeat execution of some statements.
  ‣ while loop
  ‣ do-while loop
  ‣ for loop

5.2 The while loop

• As long as the relational expression is true, repeat the statement

while syntax and semantics

• The while statement is used to repeat statements:

```markdown
while (expression)  
  statement
```

• How it works:
  ‣ expression is evaluated:
    ‣ If it is true, then statement is executed, then it starts over (and expression is evaluated again).
    ‣ If it is false, then statement is skipped (and the loop is done).
while example

• Example:

```cpp
int number = 1;
while (number <= 3)
{
    cout << "Student" << number << endl;
    number = number + 1;
} 
cout << "Done" << endl;
```

• Output

| Student1 | Student2 | Student3 | Done |

5.3 Using while for input validation

• Inspect user input values to make sure they are valid.
• If not valid, ask user to re-enter value:

```cpp
int number;
cin >> number;
while (number < 1 || number > 10) { 
    cout << "Please enter a number between 1 and 10: ";
    cin >> number;
} 
// Do something with number here 
```

Input Validation

• Checking for valid characters:

```cpp
char answer;
cin >> "Enter the answer to question 1 (a,b,c or d): ";
cin >> answer;
while (answer != 'a' && answer != 'b' &&
    answer != 'c' && answer != 'd')
{
    cout << "Please enter a letter a, b, c or d: ";
    cin >> answer;
}
// Do something with answer here
```

5.4 Counters

• Counter: a variable that is incremented (or decremented) each time a loop repeats.
• Used to keep track of the number of iterations (how many times the loop has repeated).
• Must be initialized before entering loop!!!
Counters

- Example (how many times does the user enter an invalid number?):

```cpp
int number;
int count = 0;
cout << "Enter a number between 1 and 10: ";
cin >> number;
while (number < 1 || number > 10) {
    count = count + 1;
    cout << "Please enter a number between 1 and 10: ";
    cin >> number;
}
cout << count << " invalid numbers were entered." << endl;
// Do something with number here
```

Counters

- Example, using the counter to control how many times the loop iterates:

```cpp
int num = 1; // counter variable
while (num <= 8) {
    cout << num << "   
    cout << num << "   
    num = num + 1; // increment the counter
}
```

Output:
```
Number  Number Squared
------  --------------
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
```

5.5 The do-while loop

- Execute the statement(s), then repeat as long as the relational expression is true.

```
do statement
 while (expression);
```

- How it works:
  - statement is executed.
  - expression is evaluated:
  - If it is true, then it starts over (and statement is executed again).
  - If (when) it is false, the loop is done.

- statement always executes at least once.
do-while example

• Example:

```cpp
int number = 1;
do {
cout << "Student" << number << endl;
number = number + 1;
} while (number <= 3);
cout << "Done" << endl;
```

• Output

| Student1 | Student2 | Student3 | Done |

Different ways to control the loop

• Conditional loop: body executes as long as a certain condition is true
  ‣ input validation: loops as long as input is invalid

• Count-controlled loop: body executes a specific number of times using a counter
  ‣ actual count may be a literal, or stored in a variable.

• Count-controlled loop follows a pattern:
  ‣ initialize counter to zero (or other start value).
  ‣ test counter to make sure it is less than count.
  ‣ update counter during each iteration.

5.6 The for loop

• The for statement is used to easily implement a count-controlled loop.

```
for (expr1; expr2; expr3)
statement
```

• How it works:
  1. expr1 is executed (initialization)
  2. expr2 is evaluated (test)
  3. If it is true, then statement is executed, then expr3 is executed (update), then go to step 2.
  4. If (when) it is false, then statement is skipped (and the loop is done).

do-while with menu

```cpp
char choice;
do {
cout << "A: Make a reservation." << endl;
cout << "B: View flight status." << endl;
cout << "C: Check-in for a flight." << endl;
cout << "D: Quit the program." << endl;
cout << "Enter your choice: ";
cin >> choice;
switch (choice) {
case 'A': // code to make a reservation
  break;
case 'B': // code to view flight status
  break;
case 'C': // code to process check-in
  break;
}
} while(choice != 'D');
```
The for loop flow chart

for (expr1; expr2; expr3)
  statement

expr1

expr2

statement

expr3

The for loop and the while loop

- The for statement

```
for (expr1; expr2; expr3)
  statement
```

- is equivalent to the following code using a while statement:

```
expr1;       // initialize
while (expr2) {
  // test
  statement
  expr3;     // update
}
```

for loop example

- Example:

```cpp
int number;
for (number = 1; number <= 3; number++)
{
  cout << “Student” << number << endl;
}
cout << “Done” << endl;
```

- Output

<table>
<thead>
<tr>
<th>Student1</th>
<th>Student2</th>
<th>Student3</th>
<th>Done</th>
</tr>
</thead>
</table>

Counters: redo

- Example, using the counter to control how many times the loop iterates:

```cpp
cout << “Number” << number << “ Number Squared” << endl;
cout << “--------” << “--------” << endl;
int num = 1;  // counter variable
while (num <= 8) {
  cout << num << “ ” << “ ” << (num * num) << endl;
  num = num + 1;  // increment the counter
}
cout << “Number” << number << “ Number Squared” << endl;
cout << “--------” << “--------” << endl;
```

- Rewritten using a for loop:

```cpp
cout << “Number” << number << “ Number Squared” << endl;
cout << “--------” << “--------” << endl;
int num;
for (num = 1; num <= 8; num++)
{
  cout << num << “ ” << “ ” << (num * num) << endl;
}
cout << “Number” << number << “ Number Squared” << endl;
cout << “--------” << “--------” << endl;
```
Define variable in init-expr

• You may define the loop counter variable inside the for loop’s initialization expression:

```cpp
for (int x = 10; x > 0; x=x-2)
    cout << x << endl;
```

• Do NOT try to access x outside the loop (the scope of x is the for loop statement ONLY)

• What is the output of the for loop?

User-controlled count

• You may use a value input by the user to control the number of iterations:

```cpp
int maxCount;
cout << "How many squares do you want?" << endl;
cin >> maxCount;
```

```cpp
for (int num = 1; num <= maxCount; num++)
    cout << num << "           " << (num * num) << endl;
```

• How many times does the loop iterate?

Loops in C++ (review)

• while

```cpp
while (expression) {
    statement
}
```

¬ if expression is true, statement is executed, repeat

• for

```cpp
for (expr1; expr2; expr3) {
    statement
}
```

¬ equivalent to:

```cpp
expr1;
while (expr2) {
    statement
    expr3;
}
```

• do while

```cpp
do {
    statement
} while (expression);
```

statement is executed. if expression is true, then repeat

Common tasks solved using loops

• Counting
• Summing
• Calculating an average (the mean value)
• Read input until “sentinel value” is encountered
• Read input from a file until the end of the file is encountered
Counting (review)

• set a counter variable to 0
• increment it inside the loop (each iteration)
• after each iteration of the loop, it stores the # of loop iterations so far

```cpp
int number;
int count = 0;
cout << "Enter a number between 1 and 10: ";
cin >> number;
while (number < 1 || number > 10) {
    count = count + 1;
    cout << "Please enter a number between 1 and 10: ";
    cin >> number;
}
cout << count << " invalid numbers entered " << endl;
// Do something with number here
```

5.7 Keeping a running total (summing)

• After each iteration of the loop, it stores the sum of the numbers added so far (running total)
• set an accumulator variable to 0
• add the next number to it inside the loop

```cpp
int days;           //Count for count-controlled loop
float total = 0.0;  //Accumulator
float miles;        //daily miles ridden

cout << "How many days did you ride your bike? ";
cin >> days;
for (int i = 1; i <= days; i++) {
    cout << "Enter the miles for day " << i << " : ";
    cin >> miles;
    total = total + miles;
}
cout << "Total miles ridden: " << total << endl;
```

5.8 Sentinel controlled loop

• sentinel: special value in a list of values that indicates the end of the data
• sentinel value must not be a valid value!
  -99 for a test score, -1 for miles ridden
• User does not need to count how many values will be entered
• Requires a “priming read” before the loop starts
  ‣ so the sentinel is NOT included in the sum
  ‣ the loop can be skipped (if first value is the sentinel)

Keeping a running total

• Output:

```
How many days did you ride you bike? 3
Enter the miles for day 1: 14.2
Enter the miles for day 2: 25.4
Enter the miles for day 3: 12.2
Total miles ridden: 51.8
```

• How would you calculate the average mileage?
5.9 Which Loop to use?

- Any loop can work for any given problem
- while loop:
  - test at start of loop, good for:
    - validating input, sentinel controlled loops, etc.
- for loop:
  - initialize/test/update, good for:
    - count-controlled loops
- do-while loop
  - always do at least once, good for:
    - repeating on user request, simple menu processing

5.10 Nested loops

- When one loop appears in the body of another
- For every iteration of the outer loop, we do all the iterations of the inner loop
- Example from “real life”:
- A clock. For each hour in a day (24), we iterate over 60 minutes.

<table>
<thead>
<tr>
<th>12:00</th>
<th>1:00</th>
<th>2:00</th>
<th>3:00</th>
</tr>
</thead>
<tbody>
<tr>
<td>12:01</td>
<td>1:01</td>
<td>2:01</td>
<td></td>
</tr>
<tr>
<td>12:02</td>
<td>1:02</td>
<td>2:02</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>12:59</td>
<td>1:59</td>
<td>2:59</td>
<td></td>
</tr>
</tbody>
</table>

Print a bar graph

- Input numbers from a file. For each number, output that many asterisks (*) in a row.

```cpp
int number;
ifstream inputFile;
inputFile.open("numbers.txt");
inputFile >> number;  //priming read
while (number!=-1) {
    for (int i = 1; i <= number; i++)
        cout << '*';
    cout << endl;
    inputFile >> number;
}
```

numbers.txt:

```
8
3
6
10
-1
```

Output:

```
********
***
******
**********
```
Calculate grades for a class

For each student, input the test scores from the user and output the average.

```cpp
int numStudents, numTests;
cout << "How many students? ";
cin >> numStudents;
cout << "How many test scores? "; cin >> numTests;
for (int student=1; student <= numStudents; student++) {
    float total = 0, score;
    cout << "Enter the " << numTests << " test scores for student " << student << endl;
    for (int test=1; test <= numTests; test++) {
        cin >> score;
        total = total + score;
    }
    float avgScore = total/numTests;
    cout << "Average for student" << student << " is: " << avgScore << endl;
}
```

Output:
- How many students? 3
- How many test scores? 4
- Enter the 4 test scores for student 1: 88 90.5 92 77.5
  Average for student1 is: 87.0
- Enter the 4 test scores for student 2: 66.5 70.5 80 86
  Average for student2 is: 75.8
- Enter the 4 test scores for student 3: 99 93.5 80 79
  Average for student3 is: 87.9

5.11 More File I/O

- Can test a file stream variable as if it were a boolean variable to check for various errors.
- After opening a file, if the open operation failed, the value of file stream variable is false.

```cpp
ifstream infile;
infile.open("test.txt");
if (!infile) {
    cout << "File open failure!";  
    return 1;  // abort program!
}
```

- Note: after ANY input operation, if it fails, the value of file stream variable will then be false.
Using >> to detect end of file

- stream extraction operation (>>) returns true when a value was successfully read, false otherwise

```cpp
int num;
ifstream inputFile;
inputFile.open("numbers.txt");
bool foundValue = (inputFile >> num);
```

- `inputFile >> num`:
  - tries to read a value into `num`
  - if it was successful, result is true (`foundValue` is true)
  - if it failed (non-number char or no more input), result is false (`foundValue` is false, but the value in `num` does not change!)

Using the result of >>

- Example:

```cpp
int number;
ifstream inputfile;
inputfile.open("numbers.txt");
bool foundValue = (inputFile >> number);
if (foundValue)
  cout << "The data read in was: " << number << endl;
else
  cout << "Could not read data from file." << endl;
```

- Can also use directly as relational expression:

```cpp
if (inputFile >> number)
...
```

Sum all the values in the file without using a count or sentinel value

- Code:

```cpp
int number;
ifstream inputFile;
inputFile.open("numbers.txt");

int total = 0;
while (inputFile >> number) {
    total = total + number;
}

cout << "The sum of the numbers in the file: " << total << endl;
```

- numbers.txt: Output:

<table>
<thead>
<tr>
<th>84</th>
<th>32</th>
<th>99</th>
<th>77</th>
<th>52</th>
</tr>
</thead>
</table>

The sum of the numbers in the file: 344

5.12 Breaking and Continuing

- Sometimes we want to abort (exit) a loop before it has completed.
- The `break` statement can be used to terminate the loop from within:

```cpp
cout << "Guess a number between 1 and 10" << endl;
int number;
while (true) {
    cin >> number;
    if (number == 8)
        break;
}
cout << "You got it." << endl;
```

- Don’t do this. It makes your code hard to read and debug.
Stopping a single iteration

• Sometimes we want to abort an iteration (skip to the end of loop body) before it is done.

• The `continue` statement can be used to terminate the current iteration:

```cpp
for (int i=1; i <= 6; i++) {
    if (i == 4)
        continue;
    cout << i << " ";
}
```

• Output: 1 2 3 5 6

• Don’t do this either. It makes your code hard to read and debug.