Chapter

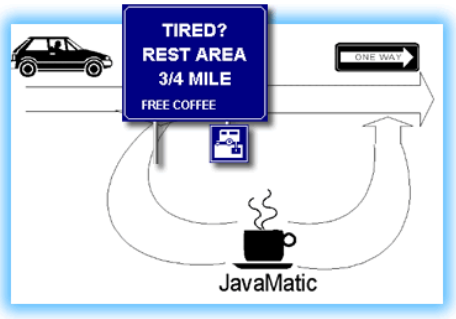
6

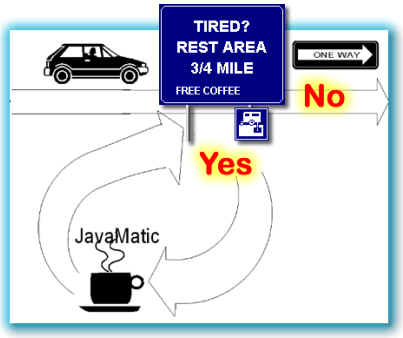
Introducing Loops

Iteration is a Computer Science term that simply means repeating a set of actions. Iteration is also called **repetition** or **looping**. The statements that are used in iteration are called **loops**. Let's start by comparing iteration the `if` statement.

Both iteration and selection are **flow-of-control** statements; they control **which** code is executed inside your program.

- Like the `if` statement, iteration is based upon evaluating a Boolean—`true`/`false`—condition, and then performing a set of actions if the test is `true`, and skipping them if it is `false`.
- Both loops and `if` statements have a **condition part**, (the test), and a **body part**, (the actions that are taken).



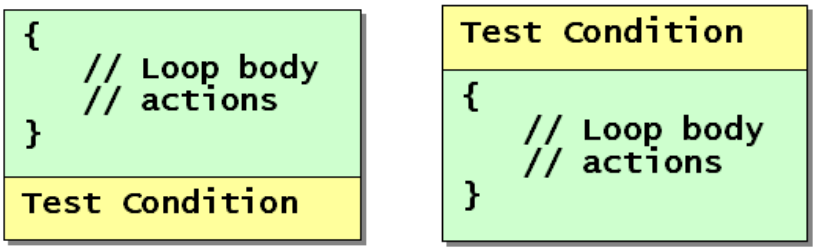


Selection works like the illustration on the left. Driving down a highway, you come to a rest stop pull off. When you leave, you rejoin the highway further down the road. Once you rejoin the highway, **you have no opportunity to go back**, and revisit the rest stop again. Those who bypass the turnoff, skip the rest stop altogether.

A loop looks similar, **but it not the same**. The illustration on the right shows that there is still a test, but, after you've had your break, the exit road "**loops back**" (hence the name), and you rejoin the highway where you initially left it. If you like, you can choose to enter the rest stop once again, even though the highway is one-way. In this sense, a loop works a little like a cloverleaf interchange.

Types of Loops

C++ has four loops: `while`, `do-while`, and two versions of the `for` loop. Each is **designed for a particular purpose**, and each has a place where it is most effective. One difference is **where** the test takes place. The `do-while` loop makes its test **after** it has performed the actions in the loop body **at least once**. This "test at the bottom" loop is also called an **unguarded** loop, because it "leaps before it looks".



The others make their tests **before** performing the actions in the loop body. These are called **guarded** loops, because when the test condition is `false`, then the actions inside the loop body are **never performed at all**.

Counted, Indefinite and Range Loops

Classifying loops according to **where** their condition is tested is not really very useful when it comes to deciding which loop to use. It is much more useful to classify loops by the **kind of bounds** that they employ.

A loop's **bounds** are the conditions under which it will repeat its actions. In a simple, loop, the this might be expressed as "the counter has a value less than ten". In more complex loops, the bounds may be a combination of conditions. There are three **major kinds of loops** that can be built using the basic loop syntax available in C++.

Definite Loops

A **definite** or (**counter-controlled**) loop repeats its actions a fixed number of times--a "gimme fifty pushups" kind of loop. Ideally you can read the code and tell how many times the loop will run. Often, though, you often won't know the **exact** number of repetitions until runtime; it may be based upon the number of characters in a `string`, for instance, or some other number which is not computed until then.

Here is an example of a counter-controlled loop:

```
for (int i = 10; i <= 20; ++i)
{
    cout << i << endl;
}
```

This kind of loop is also known as a **symmetric counter-controlled loop**, because both the **lower** bounds (10) and the **upper** bounds (20) are included in the iteration.

Indefinite Loops

With an **indefinite loop** you can **never** tell how many times the loop will repeat by examining the code. An indefinite loop is a loop that tests for the occurrence of a particular event, not a count of the number of repetitions.

"Read characters until you encounter a period" is an indefinite loop. The bounds may be reached after reading three characters, or, after reading three-thousand. It's also possible that the period might be the first character or might not occur at all.

Indefinite Categories

Here are three kinds of indefinite bounds; each uses a different sort of bounds:

- **Data bounds**: a loop that keeps **reading** until there is **no more data**. Used when processing files or network data. We'll work extensively with data loops when we get to **streams**. Here is an indefinite data loop that reads all of the words from a file, represented by input file object named in, and prints each one on its own line:

```
string word;
while (in >> word)
{
    cout << word << endl;
}
```

- **Sentinel bounds**: looks for the presence of a special marker **contained within its input**. With "read characters until you encounter a period", the period is the sentinel. Sentinel loops are often used in searching, but have other uses as well. This sentinel loop finds the position of the first period entered.

```
int position = 0; char c;
cin >> c;
while (c != '.') // Loop sentinel
{
    position++;
    cin >> c;
}
```

- **Limit bounds**: end when another repetition of the loop won't get you any closer to your goal. They are often used in scientific calculations and other numeric algorithms, when stating a precise termination condition is not possible. Often this involves monitoring the difference between two variables, and stopping the loop when the difference passes a predetermined threshold. Here is a limit-loop example which counts the number of odd digits in an integer **n**:

```
int count = 0;
while (n != 0) // Loop limit
{
    if (n % 10 % 2 == 1) { count++; }
    n = n / 10;
}
```

Range Loops

Range (or **iterator**) loops were added to the language in C++11. Range loops iterate over a collection of elements, such as a **string**, **array** or **vector**. The shorthand name for a range loop is a **foreach** loop. The **range-based for loop** which looks like this:

```
for (declaration : expression)
   statement
```

where **expression** is an object of a type that represents a sequence (such as **string**), and **declaration** defines the variable that is used to access the underlying elements in the sequence. On each cycle of the loop, the variable in **declaration** is initialized from the value of the next element in **expression**.

Here's a short example, which prints each character in a **string** on a line by itself:

```
string snake{"Ouroboros"};
for (char c : snake)
{
    cout << c << endl;
}
```

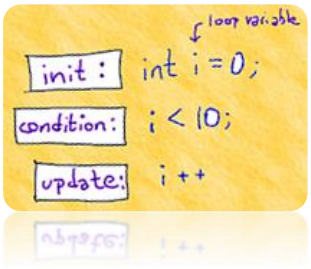
On each loop cycle, the variable **c** is initialized with the **next character** in the **string**. When all of the characters have been processed, the loop stops. Thus, you can read this loop as saying **for each character in snake**, do something. Here's a second example:

```
string str{"one two three"};
int numSpaces{0};
for (char c : str)
{
    if (c == ' ') { ++numSpaces; }
}
cout << str << " contains " << numSpaces
    << " spaces." << endl;
```

If you want to **change** the characters in the **string**, then use a **reference** variable

```
string str{"one two three"};
for (char& c : str)
{
    if (c == ' ') { c == '_'; }
}
cout << str << endl;
```

Definite Loops with *for*



C++ has a loop designed for repeating an operation a definite number of times: the ***for*** loop. You already met the **range-based** ***for*** loop in the last section. In this section we'll cover the traditional **classic counter-controlled version**.

A Definite Loop Pattern

The **pattern** used when you simply want to **repeat an action n times** is this:

```
times <- times to repeat
i <- 0
While i is less than times
   Do action
   i <- i + 1
```

Here is this pattern translated into C++ using the ***for*** loop.

```
int times = 5; // repeat 5 times
for (int i = 0; i < times; ++i)
{
    cout << "Hello" << endl;
}
```

The traditional for loop has **three sections** inside its parentheses:

- The **initialization expression** is evaluated once before the loop is entered. It creates and initializes the **loop control variable**, often named **i**. You may create other variables of the same time. These variables have **statement scope**; they are not available after the loop body. It ends with a semicolon.
- The **test condition** is evaluated after the initialization. If **true**, the body is entered; if **false**, it is skipped. The condition also ends in a semicolon.

- The **update expression** is evaluated **after** the loop body is completed. It does not end in a semicolon. It must change something in the condition, which is evaluated again, immediately following the update.

Often, the **index** or **loop control variable** is not used inside the body of the loop; it simply controls the number of repetitions. Single letter names such as `i` and `j` are common. The here goes from `0` to less-than `n`, so we say that this loop uses **asymmetric bounds**.

Sequences

A **second idiomatic variation** is used to **count from one value to another**:

```
for (int var = start; var <= finish; ++var)
```

In this `for` loop the actions in the body are executed with the variable `var` set to each value between `start` and `finish`, **inclusive**. We say that the bounds are **symmetric**. Use this loop to count from `1` to `100`:

```
for (int i = 1; i <= 100; ++i)
```

In this pattern, the loop variable is used to **produce a sequence of data**.

```
int factorial(int n)
{
    int result = 1;
    for (int i = 1; i <= n; ++i) { result *= i; }
    return result;
}
```

This example uses the loop to implement the **factorial function**, the product of the integers between `1` and `n`. The ***for*** **loop variable** `i` counts from `1` to `n`. The body of the loop updates `result` by multiplying it by `i`.

Counting Down

Here's a loop that **counts down, not up**:

```
for (int n = 10; n >= 0; --n)
{
    cout << n << ' ';
}
cout << endl;
```

```
10 9 8 7 6 5 4 3 2 1 0
```

Each of the expressions in a ***for*** statement **is optional**, but the semicolons must appear. If the **initialization** expression is missing, no initialization is performed. If the **test-condition** is missing, it is assumed to be ***true***. If **update-step** is missing, no action occurs between loop cycles.

Processing Strings

One use of the ***for*** loop is to **process strings**. The ***for*** loop, and the **asymmetric bounds pattern** are ideal, because the subscripts use by strings and arrays all begin at **0**, and the last element is always found at **size() – 1**.

Processing Single Characters

The **canonical classic *for*** loop to process every character in a **string**, should look something like this:

```
for (size_t i = 0, len = str.size(); i < len; ++i)
{
    char c = str.at(i);
}
```

Note that the **string size()** member function returns an **unsigned** type. If you are not careful, that can lead to unexpected results like this:

```
for (auto i = str.size() - 1; i >= 0; --i)
```

This loop intended to **count down** from the last character to the first at index **0**. However, if **str** is an **empty** string, then **i** starts at the largest possible **unsigned** number, instead of **-1**, since unsigned numbers "wrap around". Even worse, because **i** is an **unsigned** type, the condition **i >= 0** can never be **false**, so you can **never exit the loop**.

Here is a solution to this problem that performs correctly:

```
for (auto i = str.size(); i > 0; --i)
{
    auto c = str.at(i - 1);
}
```

Alternatively, you can store **str.size()** in an **int** variable, as long as you are sure that the **string** you are processing is no longer than the positive range of an **int**. Avoid this because then you have a potential bug in your code when strings get large.

A Bad Habit

You may see the idiomatic loop written like this:

```
for (size_t i = 0; i < str.size(); i++)
    // do something with s[i] or s.at(i)
```

This is a **bad habit** which **assumes** that calling **`size()`** is "free"—that is, it executes in constant time and there is no overhead for calling the function. This is close to true for **`string::size()`**, but **it is not true** for all functions. **Don't train your fingers** to do that.

Even worse is combining this bad habit with **`int`** indexes, like this:

```
for (int i = 0; i < str.size(); i++)…
```

The comparison **`i < str.size()`**automatically converts **`i`** to **`unsigned`**. If **`i`** ever becomes negative, it is compared **as if it were a very large positive number**. Your compiler may warn you if you mix signed and unsigned numbers like this, but it's easier to remember: **Just don't do it!**

Since **`size()`** never changes in the loop, **store the length in a variable**, and **use the variable** in your test. Here is an example using **`int`**, but you should do the same thing even if you use **`size_t`** (which is better).

```
for (int i = 0, len = str.size(); i < len; i++) …
```

A minor problem with using **`int`**, is that, theoretically, strings can be very large; larger than the largest possible **`int`**.

Counting Vowels

As an example, let's **count the number of lower-case vowels** in a **`string`**. Click the "running-man" link to open the **CodeCheck** editor and give it a try.

1. I have already written a portion of the function skeleton for you. Since the function returns an **`int`**, you just need to create the **`result`** variable, initialize it to zero, and **`return`** it at the end of the function.
2. Next, use the **canonical classic *`for`*** loop (from the previous page) to process and retrieve every character in the **`string`**. You should **memorize** this pattern so that it is always at your fingertips when required.
3. Finally, check the character to see if it is one of **`aeiou`**. If it is, then count it.

Processing Substrings

Processing substrings requires a little more work than processing individual characters. Let's complete a function, **countSubs()**, which counts the number of times that one **string**, **s2**, appears inside another, **s1**. Click the "running man" to try it in **CodeCheck**.

Complete the **stub** by adding a variable to hold the **count**, and then **return** it. Your code should now compile and run. You might even get a few correct.

```
int countSubs(const string& s1, const string& s2)
{
   int count{0};
   . . .
   return count;
}
```

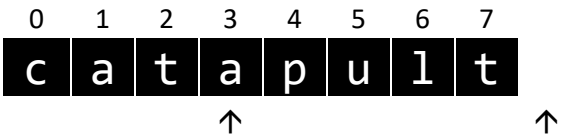
To extract a **substring** from a **string str**, you use the code:

```
string subs = s1.substr(index, count);
```

The variable **index** is the position you want to start extracting from, and **count** is the number of characters to extract. However, if **index** is on the last character, and **count** is greater than **1**, you won't extract the correct number of characters.

Instead, let's **think about the loop in a different way**. You need to loop through **s1**, extracting each three-character substring, and comparing it to **s2**. Rather than writing a ***for*** loop with **index** refer to the **beginning** of the substring, you can have it point to **the element following the substring**, and then extract the characters **preceding index**.

If we suppose **s1** is "**catapult**" and **s2** is "**tap**", here is how that looks:



Your loop becomes very easy to write:

```
int count{0};           // times s2 found in s1
size_t slen{s2.size()}; // size of string looking for
for (size_t i = slen, len = s1.size(); i <= len; ++i)
{
   string subs = s1.substr(i - slen, slen);
   . . .
}
```

Things to notice about this loop:

- The loop index **(i)** starts at **slen**, where **slen** is the size of the substring you intend to extract. It **does not** start at **0**.
- When calling **substr**(**index**, **count**), the **index** is **i-slen**, which means you are extracting the characters **before** **i**, not **after** it.
- Since **i** points to the first position **past** your substring, the loop condition is not **i < len**, but **i <= len**. (Make sure you don't confuse **len** and **slen**).

All that's left to do is to compare **subs** to **s2** and update your counter. With C++ strings, you can use **==** ; you don't need to use an **equals()** method as you would with Java.

Sentinel Loop Patterns

How do you handle problems where the loop **reads data from the user** until some special value, or **sentinel**, is found to signal the end of the input? This is called a **sentinel loop** and its logical structure is:

```
Read a value
If the value is equal to the sentinel Then
    Exit the loop
Process the value
```

There is no easy test at the beginning of the loop; you don't know when the sentinel is encountered **until you've read a value**. There are three ways to **rearrange** the statements to handle this situation.

The Primed Loop Pattern

The **primed-loop pattern** is named after the old-west water pump which required users to pour water down the well to establish suction, before pumping began.

```
Prompt user and read in a value.
While value is not the sentinel
   Process the data value.
   Prompt user and read next value
```

This is the **classic way** to process sentinel data. The code used to read each data value is **duplicated, before the loop and at the end of the loop**.

Here's a **primed-sentinel-loop** that sums a sequence, using **0** as the sentinel:

```
cout << "Add integers. Enter 0 when done." << endl;

int total = 0, value = -1;
cout << "> ";
cin >> value;                // Read before the loop
while (value != 0)           // Check for the sentinel
{
    total += value;          // No sentinel? Process
    cout << "> ";            // Prompt and read next item
    cin >> value;
}
cout << "Total: " << total << endl;
```

The loop-and-a-half Pattern

Another way to write a sentinel loop is to **use the break statement**, which has the effect of immediately terminating the nearest enclosing loop. With **break** you code the loop in a form that **follows the natural structure**: the **read-until-sentinel** pattern:

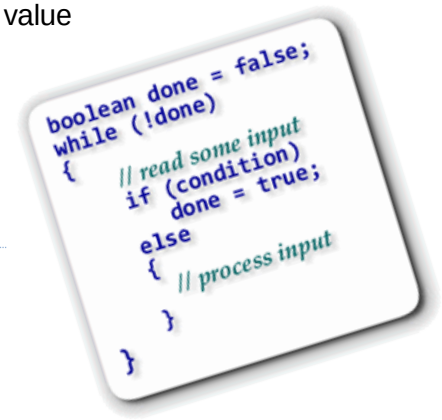
```
While True
   Prompt user and read value
   If value == sentinel Then
      break out of the loop
   Process the value
```

This is an endless loop, where the only way to exit is by executing the **break** statement. Here's the same problem, using the **loop-an-a-half pattern**:

```
while (true)                 // Endless loop
{
    cout << "> ";            // Prompt and read item
    cin >> value;
    if (value == 0) break;   // Sentinel? Leave loop
    total += value;          // No sentinel? Process
}
```

The Flag-controlled Pattern

A third way to implement the **read-until-sentinel** pattern is to use a **flag-controlled** loop, where you introduce an additional **Boolean** variable just before the loop starts and set it to **false**. Inside the loop you read a data value and check the sentinel, just as in the loop-and-a-half.



Instead of a **break** statement, set your flag variable to **true** when the sentinel is read. Otherwise, you process that data value as normal:

```
Let flag be False
While flag is False
    Prompt user and read value
    If value == sentinel Then
        Let flag be True
    Else
        Process the value
```

Which version **should** you use? Eric Roberts[1] suggests that empirical studies demonstrate that students are more **likely to write correct programs if they use the loop-and-a-half version** than if they are forced to use some other strategy.

If you're interested, follow the link below to find Roberts' paper.

Finish Up

- Complete the **reading exercises** (REX) for this chapter.
- Complete the **homework** using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using **`make submit`**.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.

[1] http://www-cs.stanford.edu/people/eroberts/papers/SIGCSE-1995/LoopExits.pdf