

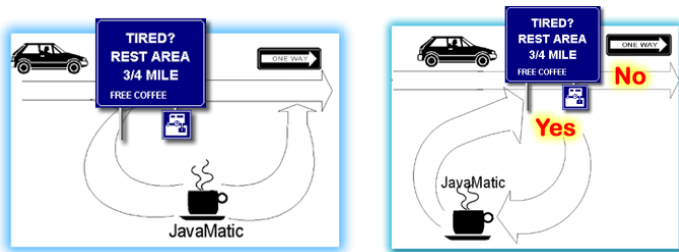
Introducing Loops

An old CS joke. Why did the Computer Science student starve to death in the shower? He was following the instructions on the bottle of shampoo. To really "get" the joke, you have to learn about **iteration**, a Computer Science term that 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 to 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.



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.

Guarded & Unguarded Loops

C++ has four loops:

```
1 while (condition) { statements; }
2 do { statements; } while (condition);
3 for (initializer; condition; update) { statements; }
4 for (element : collection) { statements; }
```

Each loop 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, illustrated on the right loop, tests its condition **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 three loop types check the test condition **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**.



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.

Classifying 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++.

- 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.



Sometimes you 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.

- 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.

- Range-based** 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 informal name for a range loop is a **foreach** loop. The range-based **for** loop looks like this:

```
for (declaration : collection)
    statement
```

where *collection* is an object of a type that represents a sequence (such as a **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 **collection**.

Now, let's look at different kinds of **indefinite** loops.



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.

Indefinite Loop Categories

Indefinite loops are those that wait until an event occurs at runtime to complete. You might wonder, "what kind of event could that be?". Here are three kinds of indefinite bounds; each uses a different sort of bounds:

Data bound loops are those that keep **reading from input** until there is **no more data** to be read. Data loops are 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 the input file object named `in`, and prints each one on its own line:

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

Sentinel bound loops look for the presence of a special value, **contained within its input**, to determine when to quit. If your problem is "read characters until you encounter a period", then 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 bound loops 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;
}
```



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.

Range-based Loops

As mentioned earlier, range-based loops were added in

C++11. A range loop will visit each element in a collection, setting the range variable to each value in the collection, in turn. These loops are very similar to the simplified **for** loop added in Java 5, and the **for in** loops in Python.



Let's look at the three variations of range-based loops which are offered in C++. We'll start with **value iteration**, which is the closest to the version used in Java.

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 a **copy** of the next character in the **string snake**. 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"*.

With value iteration, changes to the variable **c** have no effect on the **string snake**. If you **want** to change the **string** itself, then use **reference iteration**. Here's a second example which does that:

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

Finally, if the items you are iterating over are very large, and you want to make sure you don't change them, then you'd use **const-ref iteration** like this (made up) example.

```
Album photos(get_phone_photos());
for (const Image& img : photos)
{
    if (is_cute_cat(img))
    {
        display(img);
    }
}
```

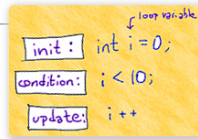
Because each picture in your photos library might be very large, you **wouldn't** want to copy them with value iteration. Similarly, since you wouldn't want the **CuteCats** app to have the ability to modify (or perhaps erase) your cute cat photos, the loop should access each variable as a **const Image&**.



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.

Definite Loops with *for*

C++ has a loop designed for repeating an operation a fixed 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.



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 type in this section. These variables have **statement scope**; they are not available after the loop body. The initialization section ends with a semicolon.
- The **test condition** is first 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. The update expression must change one of the variables 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 conventional. If you want others to understand your code, you'll conform to this convention.

The loop shown here goes from **0** to **less-than times**, so we say that this loop uses **asymmetric bounds**. This means the **lower** bounds is **included** while the **upper** bound is **excluded**.



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.

Sequences & Symmetric Bounds

A second idiomatic variation of the *for* loop is used to **generate sequential data**, such as counting from **start** to **finish**:

```
for (int var = start; var < finish; update-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**.

Because we include both ends, we say that the bounds used in this loop are **symmetric**. Use this loop to count from **1** to **100** like this:

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

Here's another example. 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;
}
```

As you can see, this example uses the loop to implement the **factorial function**, the product of the integers between **1** and **n**. The *for* loop variable **i** goes from **1** to **n** (inclusive). The body of the loop updates **result** by multiplying it by **i**.

Counting Down

As the *factorial* example shows, the update expression need only move the loop variable closer to the loop bounds. It must **advance** the loop; it doesn't need to increment. Here's an example:



Here we want to **start the loop at a large number**, and **decrease the index by four** on each iteration. In other words, we want to **count down** rather than counting up. Here's what this looks like in code:

```
for (int i = 19; i >= 0; i -= 4)
    cout << i << " ";
cout << endl;
```

*Don't use conditions like **i != 0**, unless you are certain that the condition will be met. Because we are decrementing by four, we will never set **i** to **0** and so we would have an **infinite** or **endless** loop.*




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**.

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);    // now, process c in some way
}
```

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 (size_t len = 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 loop that is written correctly:

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

Alternatively, **you can store str.size() in an int variable**, as long as you:

- cast the returned value from **str.size()** to an **int** like this:

```
for (int i = static_cast<int>(str.size()); i >= 0; --i) ...
```
- make sure that the **string** you are processing is no longer than the positive range of an **int**. If you do this, your loop will **not work** if you have a large string.



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.

Some Bad Habits

You may see the idiomatic loop written like this:

```
for (size_t i = 0; i < str.size(); ++i)
    // do something with str[i] or str.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. For instance, when working with C-style strings, using the equivalent `strlen()` function is very expensive. **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 the type of `i` to an **unsigned size_t**. 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:

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

*Should you use `i++` or `++i` in your loop update expression? With `int` or `size_t` indexes, it makes no difference. The effect is the same either way. However, I prefer the prefix version (`++i`) because I want to "train my fingers" for more the more advanced **iterator** loops you'll work with in CS 250. With iterators, often the `i++` version is much slower, or even non-existent.*



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.

Hands On: Counting Vowels

Now it's your turn. On the next page you'll find a function which should **count the number of lower-case vowels** in its **string** parameter.

► *Watch the solution video*

Look at the hints I've supplied here, and give it a try.

1. I have already written a portion of the function skeleton (or stub) 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.

After you've given it a try, come back here and watch the solution video.



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.

Hands On: Counting Substrings

Processing substrings requires a little more work than processing individual characters. On the next page you'll find a function, `countSubs()`, which counts the number of times that one string, `s2`, appears inside another, `s1`.

► *Watch the solution video*

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;
}
```

Next, to extract a **substring** from `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. Compare the substring to `s2`, and, if it matches, count it.

```
if (subs == s2) { ++count; }
```

Go ahead and complete the exercise on the next page, and then come back here and check your work with the solution below.



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.

More Efficient Substrings

Here's an illustration of the loop we just wrote, using it to count the number of times that **mo** appears in **moon**.

► [Watch the video](#)

You'll notice that the last two iterations of the loop are not necessary, since we're comparing **mo** to **on**, and then a last time to **n**.

You may have encountered this situation in Java, where you'd solve it by changing the loop like this:

```
for (size_t i = 0, len = s1.size() - (s2.size() - 1); i < len; ++i)
```

Now, your code correctly works for the example shown here, only completing the first two repetitions, and skipping the two extraneous ones at the end.

However, what if the string **s1** contained **m** instead of **moon**. In Java, the expression `s1.size() - (s2.size() - 1)` would result in a negative number (`1 - (3 - 1) -> -1`), and the loop would work correctly. In C++, **that's not the case**, since the types are **unsigned** numbers, so the result "wraps around" to a very large number, and your program crashes.

That means, to get this to work correctly for **any** input in C++, you need to add a **loop guard** like this:

```
size_t len1 = s1.size(), len2 = s2.size();
if (len2 <= len1) {
    for (size_t i = 0, len = len1 - (len2 - 1); i < len; ++i) {
        if (s1.substr(i, len2) == s2) { count++; }
    }
}
```

0	1	2	3
m	o	o	n
m	o	o	
	m	o	o
		m	o
			m
			o
			o



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.

Substrings Redux

Instead of using a loop guard, let's **think about the loop in a different way**.

You need to loop through **s1**, extracting each 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**.

► Watch the video

0	1	2	3	4	5	6	7
c	a	t	a	p	u	l	t

 ↑ ↑

A picture might help. Suppose **s1** is "catapult" and **s2** is "tap", here is how that looks. Your loop becomes very easy to write, and you don't need any extra **if** statements:

```
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 );
    if (subs == s2) { ++count; }
}
return count;
```

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 in Java.



This course content is offered under a [CC Attribution Non-Commercial](#) license. Content in this course can be considered under this license unless otherwise noted.