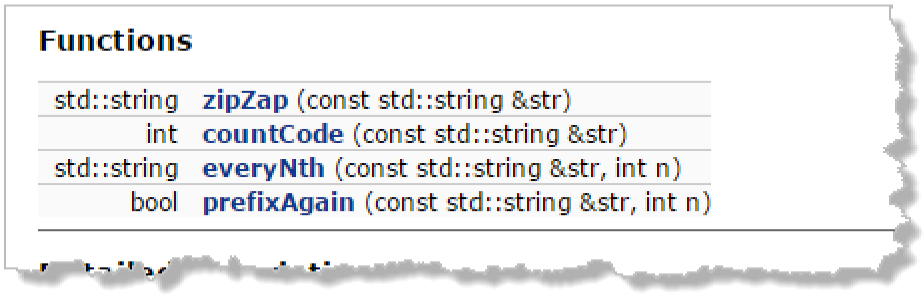
Writing String Functions

Are you ready to try your hand at writing some string processing functions? Upload the starter code into your IDE. There, you'll find four **fun-fun-functions** from the hand of Nick Parlante[1] similar to the short CodingBat[2] style programs you may have encountered in CS 170.

Open the header file to find the prototypes, along with the documentation for each of the functions. Here are the four functions that you'll be implementing.



Functions

std::string	**zipZap** (const std::string &str)
int	**countCode** (const std::string &str)
std::string	**everyNth** (const std::string &str, int n)
bool	**prefixAgain** (const std::string &str, int n)

Put your implementation (that is, the **function definitions**), in the implementation (**.cpp**) file. Once you have added the actual functions, **build** to do some simple testing. You'll find sample output embedded in the documentation comments in the interface (**.h**) header file.

Writing the Stubs

You should be able to **stub out** each of the functions, by following these four steps.

1. **Copy the prototypes** from the header to the implementation file. You may, if you want, copy the comments as well, but you don't need to.

[1] http://cs.stanford.edu/people/nick/

[2] http://codingbat.com/java

2. Remove the semicolons at the end of each of the prototypes. While you are doing this, you may also remove the `std::` qualifications in front of each of the library type names. While required in the header file, you don't need them in the implementation file, where you have `using namespace std` to qualify the names.
3. Add braces to provide a body for each function.
4. Return a placeholder. Look at the return type for each function. Inside the body, create a variable of that type and then `return` it at the end of the function. Make sure that the variable you return is the same type as the return type of the function. Particularly be careful that you don't return the integer `0` from a function that expects a `string` result.

Please memorize this mechanical process, so that it becomes second nature, part of your "muscle memory", like using a clutch or fingering the keys on a piano. You don't want to think about it when programming.

Iteration Patterns

These **iteration patterns** are worth memorizing so that you don't waste time thinking about them. When you recognize that you need to cycle through the characters in a `string`, some part of your nervous system between your brain and your fingers, should translate that idea effortlessly into the following line:

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

Sometimes, you will need to modify the basic iteration pattern to start or end the iteration at a different index position. For example, the following function checks to see whether a `string` begins with a particular prefix:

```
bool startsWith(const string& str, const string& prefix)
{
    if (str.size() < prefix.size())
        return false;
    for (size_t i = 0, len = prefix.size(); i < len; ++i)
    {
        if (str.at(i) != prefix.at(i))
            return false;
    }
    return true;
}
```

For right now, just look at the code in the body. It begins by checking to make sure that `str` is not shorter than `prefix` (in which case, the result must certainly be `false`) and then iterates only through the characters in `prefix` rather than the `string` as a whole.

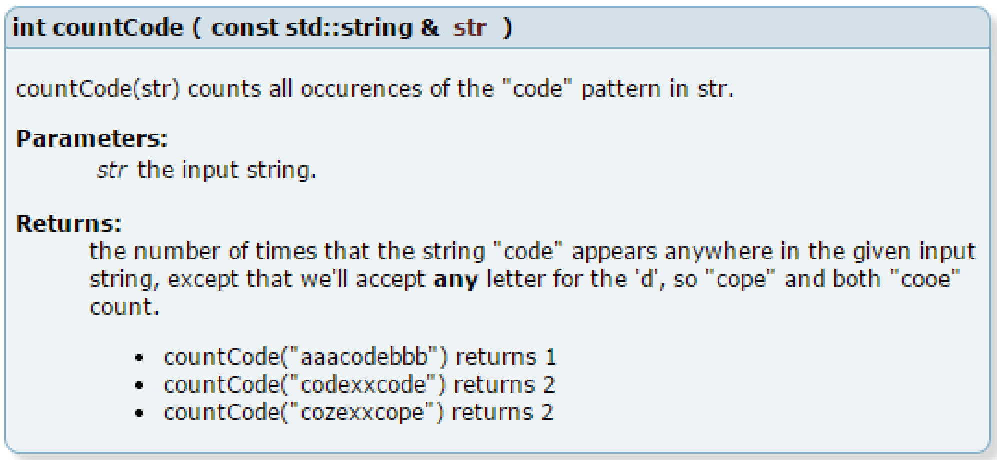
As you read the code for **startsWith()**, pay attention to the placement of the two **return** statements. The code returns **false** as soon as it discovers the first difference between the **string** and the **prefix**.

The code returns **true** from outside the loop, after it has checked every character in the prefix without finding any differences. You will encounter examples of this basic pattern over and over again.

The **startsWith()** function is very useful, even though it is not part of the standard **<string>** library in C++. It is an ideal candidate for inclusion in a library of **string** functions that you create for yourself. You'll learn how to do that later.

Counting Code

Here's the processed documentation for the **countCode()** function.



int countCode (const std::string & str)

countCode(str) counts all occurences of the "code" pattern in str.

Parameters:
str the input string.

Returns:
the number of times that the string "code" appears anywhere in the given input string, except that we'll accept **any** letter for the 'd', so "cope" and both "cooe" count.

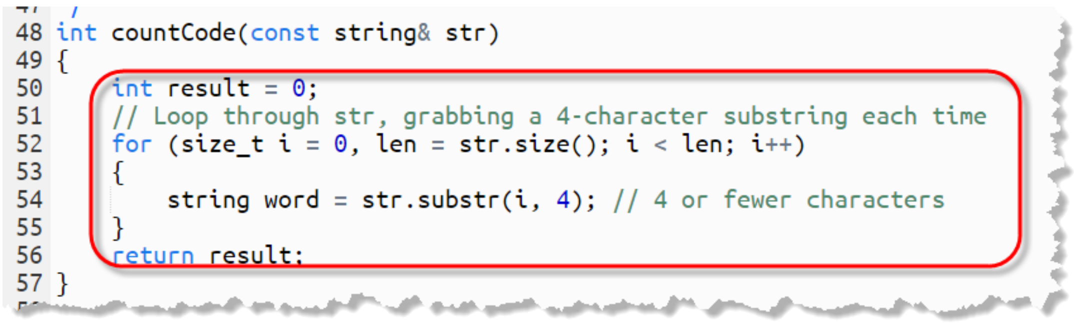
- countCode("aaacodebbb") returns 1
- countCode("codexxcode") returns 2
- countCode("cozexxcope") returns 2

You've already written the function stub, so let's plan out an implementation. This should be pretty straight forward:

```
1. Extract first four characters (substring) of the input.
2. If the string starts with "co" and it ends with 'e'
      Then add one to the counter
3. Repeat for the next four characters. (This is a loop.)
```

Step 1 – the Basic Loop

Because you need to examine "chunks" of the **string**, not individual characters, the simpler C++11 range-based loop isn't really appropriate. Instead, you must use manual iteration. Your first pass should look something like this:



```
int countCode(const string& str)
{
    int result = 0;
    // Loop through str, grabbing a 4-character substring each time
    for (size_t i = 0, len = str.size(); i < len; i++)
    {
        string word = str.substr(i, 4); // 4 or fewer characters
    }
    return result;
}
```

This is legal in C++, because the substring start index (**`i`**) is always inside the **`string`**. However, the substring **`word`** is **not always** four characters long. When **`i`** approaches the end of the **`string`**, **`substr()`** returns **less than four characters**. This is quite different than the way Java works (where the code would crash).

You can ensure that **`word`** always contains exactly four characters, by adjusting the loop's upper bounds, like this:

```
// Loop through the string grabbing a 4-char substring
for (size_t i = 0, len = str.size(); i < len - 3; i++)
```

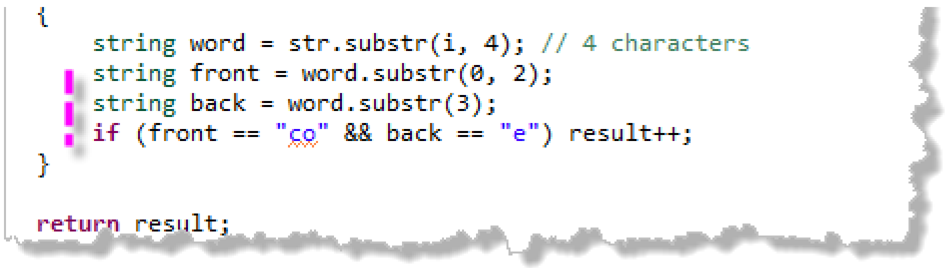
If you want a substring that is **`4`** characters long, use **`-3`** for the adjustment. If you want a substring that is **`10`** characters long, use **`-9`**, and so on.

Step 2 - Loop Processing

The second step is where you do the work; everything up to this is just mechanics—making sure the loop goes around the correct number of times. The logic for this section reads:

```
2. If the string starts with "co" and it ends with 'e'
      Then add one to the counter
```

Use an **`if`** statement along with **`substr()`** or **`at()`** get the individual parts. Here's one implementation:

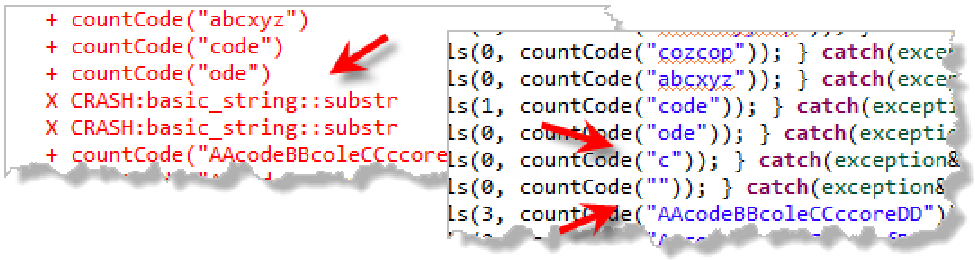


```
{
    string word = str.substr(i, 4); // 4 characters
    string front = word.substr(0, 2);
    string back = word.substr(3);
    if (front == "co" && back == "e") result++;
}

return result;
```

Tracing a Bug

If you test this, you'll see that 86% of the tests work correctly, but two of them crash! Can you figure out why?



```
+ countCode("abcxyz")
+ countCode("code")
+ countCode("ode")
X CRASH:basic_string::substr
X CRASH:basic_string::substr
+ countCode("AAcodeBBcoleCCccore
```

```
ls(0, countCode("cozcop")); } catch(exce
ls(0, countCode("abcxyz")); } catch(exce
ls(1, countCode("code")); } catch(excepti
ls(0, countCode("ode")); } catch(excepti
ls(0, countCode("c")); } catch(exception&
ls(0, countCode("")); } catch(exception&
ls(3, countCode("AAcodeBBcoleCCccoreDD")
```

I've placed the output on the left and the source code from the tests on the right. From this, you see that the function crashes when calling **`substr()`** if the **`countCode()`** function is called with **`"c"`** or with the empty string **`""`**. In those cases, you should not enter the body of the loop and should never call the **`substr()`** function.

Why is the function entering the loop? Here's what you **expect** the loop conditions to be; as you can see, it **should never enter the loop** in these cases!

Input	i	len	len - 3	i < len - 3
"c"	0	1	-2	false
""	0	0	-1	false

To see what the actual problem is, you can use a debugger or just print some values in the console window. Here's the code I used:

```
size_t i = 1;
cout << "1 - 3->" << i - 3 << endl;
i = 0;
cout << "0 - 3->" << i - 3 << endl;
```

And here's the output I got when I ran it:

```
1 - 3->4294967294
0 - 3->4294967293
```

Of course! I forgot about the unsigned numbers. In this case **`len - 3`** is not **`-2`**, but **`4294967294`**, which is certainly greater than **`i`**. The fix is to put your **`for`** loop inside an **`if`** statement, so you only iterate when **`str.size() > 3`**.

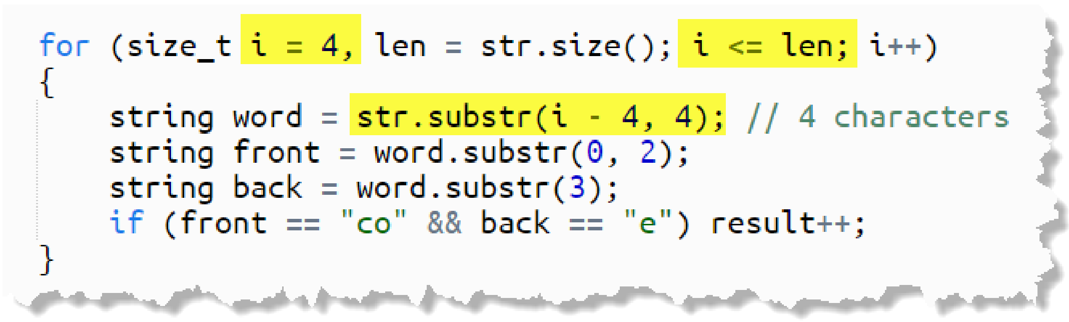
Rethinking the Problem

Whenever you find yourself with a solution that requires you to add extra **`if`** statements to handle "special cases", it's worth spending a few minutes revisiting the problem and seeing if it can be handled in a different way.

In our case, we only correctly handle strings where the length is at least **`4`**. That happens because the condition **`i < len - 3`** is only legal when **`len`** is at least **`4`**. If **`len`** is **`3`**, then **`len - 3`** is **`0`** and since **`i`** is **`unsigned`**, no possible value for **`i`** could ever be less than **`0`**. This statement is undefined for some input values.

We can fix this by making sure that the statement could never be undefined. To do this, instead of starting **`i`** at **`0`**, start **`i`** at **`4`** and change the condition to **`i <= len`**. Then, if we have a string that is too small, we'll never enter the loop. The only other change that's needed is to use **`str.substr(i - 4, 4)`** when extracting the **`word`**.

Here's the modified loop:



```
for (size_t i = 4, len = str.size(); i <= len; i++)
{
    string word = str.substr(i - 4, 4); // 4 characters
    string front = word.substr(0, 2);
    string back = word.substr(3);
    if (front == "co" && back == "e") result++;
}
```

The Growing a string Pattern

Another important pattern to memorize as you learn how to work with strings involves creating a new string one character at a time. The structure will depend on the application, but the general pattern for creating a **`string`** by concatenation is this:

```
string result;
for (whatever loop header line fits the application)
{
   result += the next substring or character;
}
```

Here's a simple function, ***`repeatStr()`***, that might be used when you want to generate a section separator in console output. The **`string`** class already has a constructor that does this for an individual character, it doesn't contain version that allows you to repeat a **`string`** as ***`repeatStr()`*** does.

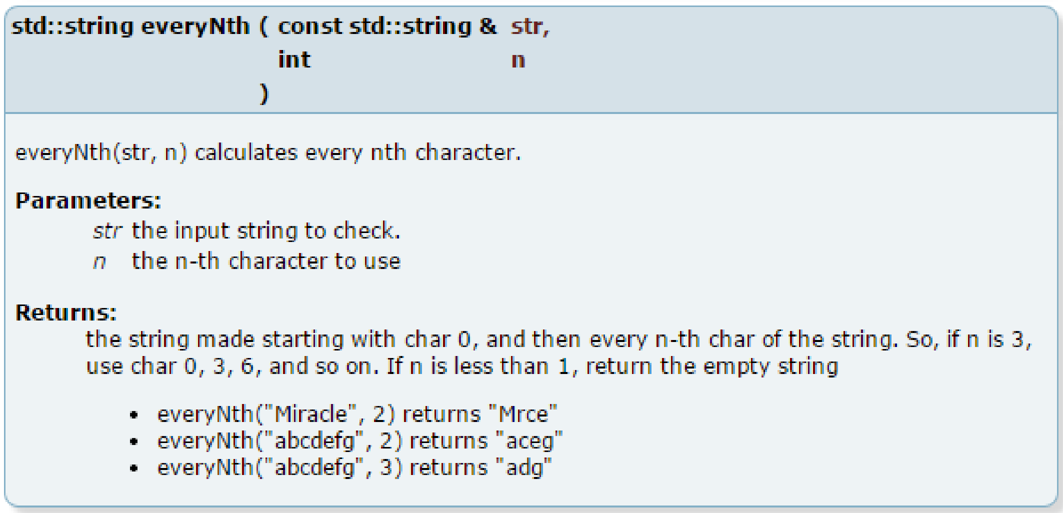
```
string repeatStr(int n, const string& pat)
{
    string result;
    for (int i = 0; i < n; i++)
    {
        result += pat;
    }
    return result;
}
```

Many string-processing functions use both the iteration and concatenation patterns together. For example, the following function **reverses** its argument so that, for example, calling **`reverse("desserts")`** returns **`"stressed"`**:

```
string reverse(const string& str)
{
    string result;
    for (auto c : str)
        result = c + result;
    return result;
```

Every Nth

Here's the documentation for this function.



std::string everyNth (const std::string & str,
int n
)

everyNth(str, n) calculates every nth character.

Parameters:
str the input string to check.
n the n-th character to use

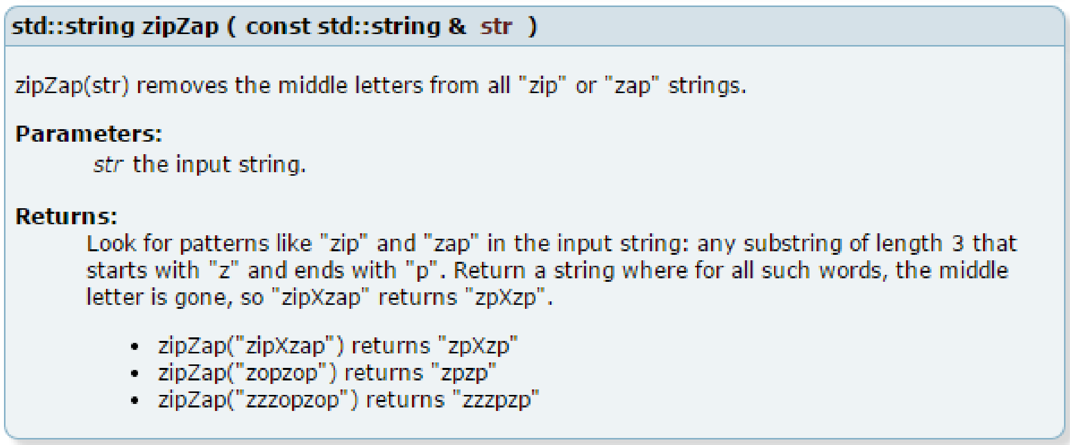
Returns:
the string made starting with char 0, and then every n-th char of the string. So, if n is 3, use char 0, 3, 6, and so on. If n is less than 1, return the empty string

- everyNth("Miracle", 2) returns "Mrce"
- everyNth("abcdefg", 2) returns "aceg"
- everyNth("abcdefg", 3) returns "adg"

The "growing a string" pattern is what we need to solve this. We can simply translate the pattern almost directly. The only change you have to make to the **loop update expression** is to move forward **n** characters, instead of moving forward by just one.

The zipZap Function

Here's the documentation for the function:



std::string zipZap (const std::string & str)

zipZap(str) removes the middle letters from all "zip" or "zap" strings.

Parameters:
str the input string.

Returns:
Look for patterns like "zip" and "zap" in the input string: any substring of length 3 that starts with "z" and ends with "p". Return a string where for all such words, the middle letter is gone, so "zipXzap" returns "zpXzp".

- zipZap("zipXzap") returns "zpXzp"
- zipZap("zopzop") returns "zpzp"
- zipZap("zzzopzop") returns "zzzpzp"

Many students, when they look at the first line, "removes the middle letters...", start looking for a **delete()**, **erase()** or **replace()** method. The **string** class **does** have a few methods like that, but unfortunately, it's very difficult to solve this problem using them. You'll learn about removing items from a container, using iterators, later.

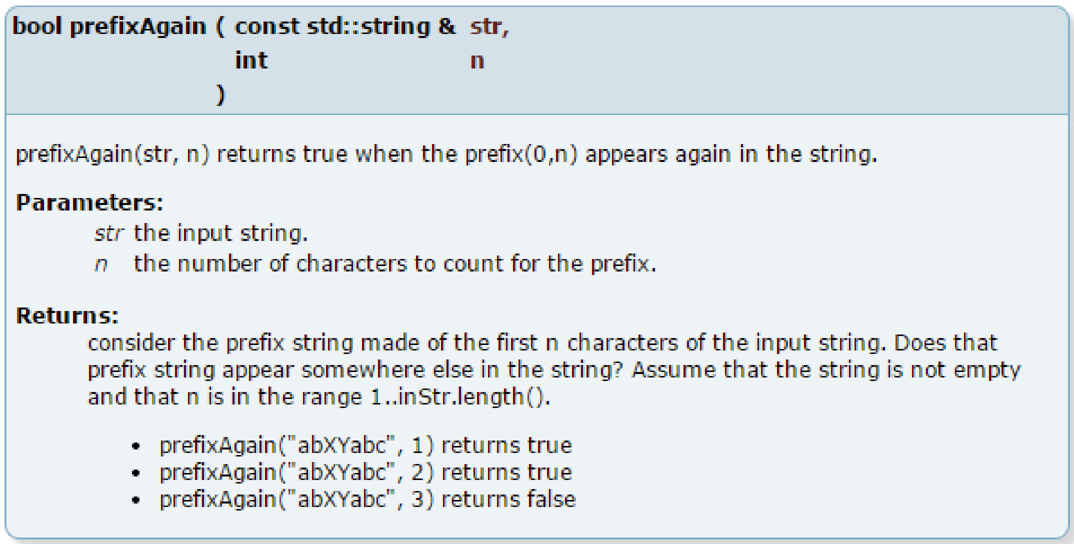
The simplest way to solve all such problems is to use the same "growing a string" pattern that you used for ***everNth()***. For this problem, it will be a little simpler to use a **while** loop, instead of **for**. You'll also need a few **if** statements for special cases.

Here's a processing plan:

```
1. Save the length (len) and create an index (i = 0)
2. If len is less than 3, then we just return the input str
3. Write a while loop that goes while i is less than len - 2
    4. Inside the loop, extract a substring of 3 (word)
    5. If word starts with 'z' and ends with 'p' then:
       - add "zp" to the output string
       - move forward three characters (to skip the "ip")
   6. Otherwise
       - add the first character of word to the output string
      - move forward one character
7. After the loop, add the remaining characters, to the output
```

The prefixAgain Function

Here is the specification for this function:



bool prefixAgain (const std::string & str,
int n
)

prefixAgain(str, n) returns true when the prefix(0,n) appears again in the string.

Parameters:
str the input string.
n the number of characters to count for the prefix.

Returns:
consider the prefix string made of the first n characters of the input string. Does that prefix string appear somewhere else in the string? Assume that the string is not empty and that n is in the range 1..inStr.length().

- prefixAgain("abXYabc", 1) returns true
- prefixAgain("abXYabc", 2) returns true
- prefixAgain("abXYabc", 3) returns false

The real problem with this function is that you **don't need to use a loop at all**. Since you know how to use the `string::find()` function, you could write the implementation in one line, like this:

```
return str.substr(1).find(str.substr(0, n)) != string::npos;
```

Of course, this chapter is on writing loops, so let me ask, how could we write the function without using `find()`? Here's a plan:

```
1. Extract the first n characters into prefix
2. Start looping at 1, extracting n characters into word
    - if word is equal to prefix, return true
3. If we get through the loop, return false
```

Finish Up

Go ahead and implement that. Then do `make test`. If you haven't made any mistakes, `make submit` and you're done with this assignment.