# WORD COUNTING

- 3 Things That Counting Words Can Reveal on Your Code
- Implementing a Simple Word Counter
- Extracting words from camelCase symbols
- Parametrizing the Type of Case
- Computing the Span of a Word

Fluent {C++}

# 3 Things That Counting Words Can Reveal on Your Code

Being able to read code and understand it quickly is an invaluable skill for a software developer. We spend way more time reading code that writing it, and being able to make a piece of code expressive to your eyes can make you much more efficient in your daily work.

There is a technique to analyse code that I've been very excited about these days: **counting words in code**. By counting words, I mean:

- calculating the number of occurrences of each word in a given piece of code, for example in a function,
- then seeing where the most frequent words are located,
- use this to deduce information about the function as a whole.

Counting words has been useful to me quite a few times to understand code I didn't know, but the main reason I'm so excited about it is that I'm sure there are plenty of things to discover about it, and I'd like to exchange with you on the subject.

This post presents three things counting words can reveal about a piece of code, and I'd love to hear your reactions afterwards.

## Locating the important objects

Let's take the example of understanding a function. It is likely that the words that occur the most frequently across that function's code have some importance in it.

To illustrate, let's experiment a word count on a function, locate the most frequent words, and see what we can learn from it. We'll use open-source code hosted on GitHub. For example, consider this function from a C++ repository called Classic-Shell.

You don't have to read its code, as our purpose is to perform a word count to start with a high level view of the function.

```
bool CSetting::ReadValue( CRegKey &regKey, const wchar_t *valName )
{
	// bool, int, hotkey, color
	if (type==CSetting::TYPE_BOOL || (type==CSetting::TYPE_INT &&
this[1].type!=CSetting::TYPE_RADIO) || type==CSetting::TYPE_HOTKEY ||
type==CSetting::TYPE_HOTKEY_ANY || type==CSetting::TYPE_COLOR)
	{
		DWORD val;
		if (regKey.QueryDWORDValue(valName,val)==ERROR_SUCCESS)
		{
			if (type==CSetting::TYPE_BOOL)
				value=CComVariant(val?1:0);
			else
				value=CComVariant((int)val);
			return true;
		}
		return false;
	}

	// radio
	if (type==CSetting::TYPE_INT && this[1].type==CSetting::TYPE_RADIO)
	{
		ULONG len;
		DWORD val;
		if
(regKey.QueryStringValue(valName,NULL,&len)==ERROR_SUCCESS)
		{
			CString text;

	regKey.QueryStringValue(valName,text.GetBuffer(len),&len);
			text.ReleaseBuffer(len);
			val=0;
			for (const CSetting *pRadio=this+1;pRadio-
>type==CSetting::TYPE_RADIO;pRadio++,val++)
			{
				if (_wcsicmp(text,pRadio->name)==0)
				{
					value=CComVariant((int)val);
					return true;
				}
			}
		}
		else if
(regKey.QueryDWORDValue(valName,val)==ERROR_SUCCESS)
		{
			value=CComVariant((int)val);
			return true;
		}
		return false;
	}
```

```
        // string
        if (type>=CSetting::TYPE_STRING && type<CSetting::TYPE_MULTISTRING)
        {
                ULONG len;
                if
(regKey.QueryStringValue(valName,NULL,&len)==ERROR_SUCCESS)
                {
                        value.vt=VT_BSTR;
                        value.bstrVal=SysAllocStringLen(NULL,len-1);
                        regKey.QueryStringValue(valName,value.bstrVal,&len);
                        return true;
                }
                return false;
        }

        // multistring
        if (type==CSetting::TYPE_MULTISTRING)
        {
                ULONG len;
                if
(regKey.QueryMultiStringValue(valName,NULL,&len)==ERROR_SUCCESS)
                {
                        value.vt=VT_BSTR;
                        value.bstrVal=SysAllocStringLen(NULL,len-1);

        regKey.QueryMultiStringValue(valName,value.bstrVal,&len);
                        for (int i=0;i<(int)len-1;i++)
                                if (value.bstrVal[i]==0)
                                        value.bstrVal[i]='\n';
                        return true;
                }
                else if
(regKey.QueryStringValue(valName,NULL,&len)==ERROR_SUCCESS)
                {
                        value.vt=VT_BSTR;
                        value.bstrVal=SysAllocStringLen(NULL,len);
                        regKey.QueryStringValue(valName,value.bstrVal,&len);
                        if (len>0)
                        {
                                value.bstrVal[len-1]='\n';
                                value.bstrVal[len]=0;
                        }
                        return true;
                }
                return false;
        }

        Assert(0);
        return false;
}
```

The function is called `ReadValue`. Not being familiar with the project, it's not easy to understand what value is read, and to do what.

Counting the words of this function (which you can do approximately by using online generic tools for counting words in text, or by coding a tool specially designed for counting words in code, which we will explore in future posts) outputs that the word that occurs the most frequently in this function is `value`. Let's highlight the occurrences of `value` in the function:

```
bool CSetting::ReadValue( CRegKey &regKey, const wchar_t *valName )
{
	// bool, int, hotkey, color
	if (type==CSetting::TYPE_BOOL || (type==CSetting::TYPE_INT && this[1].type!=CSetting::TYPE_RADIO) || type==CSetting::TYPE_HOT
KEY || type==CSetting::TYPE_HOTKEY_ANY || type==CSetting::TYPE_COLOR)
	{
		DWORD val;
		if (regKey.QueryDWORDValue(valName,val)==ERROR_SUCCESS)
		{
			if (type==CSetting::TYPE_BOOL)
				value=CComVariant(val?1:0);
			else
				value=CComVariant((int)val);
			return true;
		}
		return false;
	}

	// radio
	if (type==CSetting::TYPE_INT && this[1].type==CSetting::TYPE_RADIO)
	{
		ULONG len;
		DWORD val;
		if (regKey.QueryStringValue(valName,NULL,&len)==ERROR_SUCCESS)
		{
			CString text;
			regKey.QueryStringValue(valName,text.GetBuffer(len),&len);
			text.ReleaseBuffer(len);
			val=0;
			for (const CSetting *pRadio=this+1;pRadio->type==CSetting::TYPE_RADIO;pRadio++,val++)
			{
				if (_wcsicmp(text,pRadio->name)==0)
				{
					value=CComVariant((int)val);
					return true;
				}
			}
		}
		else if (regKey.QueryDWORDValue(valName,val)==ERROR_SUCCESS)
		{
			value=CComVariant((int)val);
			return true;
		}
		return false;
	}

	// string
	if (type>=CSetting::TYPE_STRING && type<CSetting::TYPE_MULTISTRING)
	{
		ULONG len;
		if (regKey.QueryStringValue(valName,NULL,&len)==ERROR_SUCCESS)
		{
			value.vt=VT_BSTR;
			value.bstrVal=SysAllocStringLen(NULL,len-1);
			regKey.QueryStringValue(valName,value.bstrVal,&len);
			return true;
		}
		return false;
	}

	// multistring
	if (type==CSetting::TYPE_MULTISTRING)
	{
		ULONG len;
		if (regKey.QueryMultiStringValue(valName,NULL,&len)==ERROR_SUCCESS)
		{
			value.vt=VT_BSTR;
			value.bstrVal=SysAllocStringLen(NULL,len-1);
			regKey.QueryMultiStringValue(valName,value.bstrVal,&len);
			for (int i=0;i<(int)len-1;i++)
				if (value.bstrVal[i]==0)
					value.bstrVal[i]='\n';
			return true;
		}
		else if (regKey.QueryStringValue(valName,NULL,&len)==ERROR_SUCCESS)
		{
			value.vt=VT_BSTR;
			value.bstrVal=SysAllocStringLen(NULL,len);
			regKey.QueryStringValue(valName,value.bstrVal,&len);
			if (len>0)
			{
				value.bstrVal[len-1]='\n';
				value.bstrVal[len]=0;
			}
			return true;
		}
		return false;
	}

	Assert(0);
```

The first thing we can note is that the occurrences of `value` are spread out across the whole function. This suggests that `value` is indeed an central object of the function. Note that if we had started by reading the code line by line, it would have taken much more time to figure out this piece of information.

We also note that the first time that `value` appears in the function is not via a declaration. This means that `value` is presumably a class member of the class containing the method `ReadValue` (in theory `value` could also be a global variable, but let's be optimistic and assume it's a class member).

Now if we pay a closer look to those occurrences of `value`, we notice that most of them are assignments. We now have a good assumption about the purpose of the function `ReadValue`: to fill the class member `value` (and we also understand the function's name now).

All these deductions are only based on assumptions, and to be 100% sure there are valid we would have to read the whole function. But having a likely explanation of what the function does is useful for two reasons:

- often, we don't have the time to read every line of each function we come across,
- for the functions that we do read in details, starting with a general idea of what the function does helps the detailed reading.

# Understanding how inputs are used

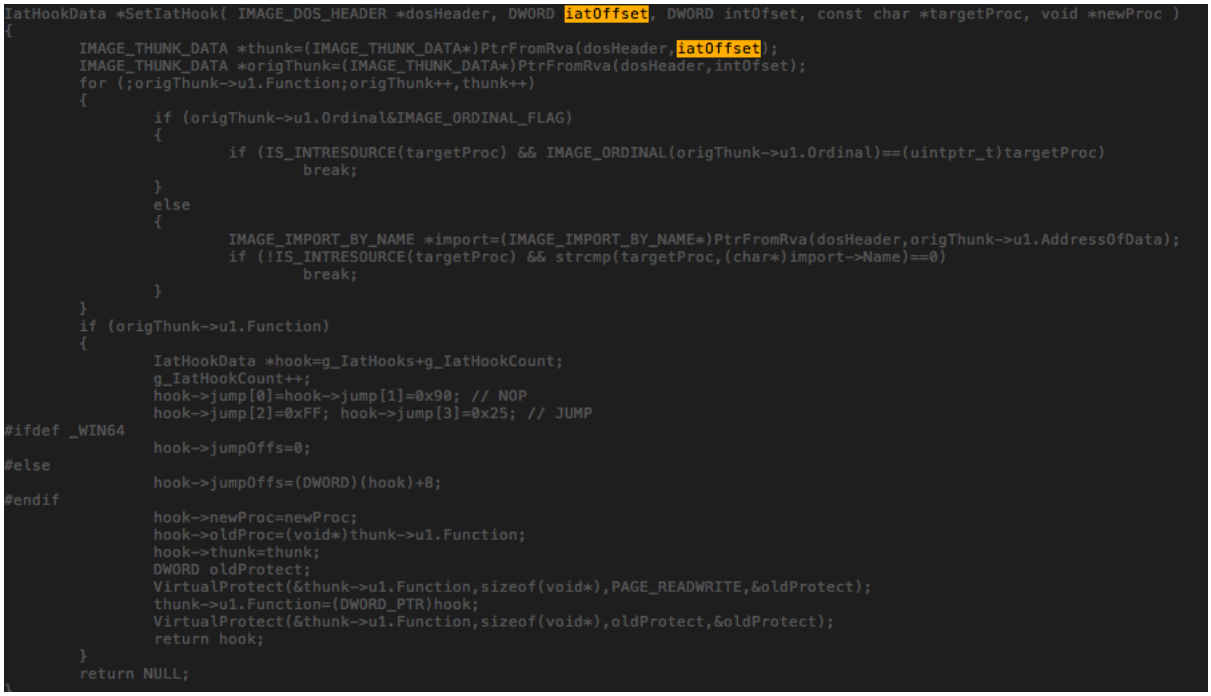
A function takes input and produces outputs. So one way to understand what a function does is to examine what it does with its inputs. On a lot of the word counts I've ran, the function's inputs are amongst the most frequently appearing words in its body.

The `ReadValue` function takes two inputs: `regKey` and `valName`. Let's highlight the occurrences of those words in the function. `regKey` is in orange, `valName` in red:


A pattern jumps out of this highlighting: `regKey` and `valName` are always used together. This suggests that, to understand them, we should consider them together. And indeed, by looking more closely at one of the lines where they are used, we see that `regKey` seems to be some sort of container, and `valName` a key to search into it.

Counting words in code can also provide ideas for refactoring tasks. Since those two objects are always used together in the function, perhaps it could be interesting to group them into one object. Or perhaps, perform the lookup of `valName` in `regKey` before calling `ReadValue`, and make `ReadValue` take only the result of the search as an input parameter.

Sometimes the input parameters are not used extensively in the function though. For example, consider this other function taken from the same codebase:



```
IatHookData *SetIatHook( IMAGE_DOS_HEADER *dosHeader, DWORD iatOffset, DWORD intOfset, const char *targetProc, void *newProc )
{
        IMAGE_THUNK_DATA *thunk=(IMAGE_THUNK_DATA*)PtrFromRva(dosHeader,iatOffset);
        IMAGE_THUNK_DATA *origThunk=(IMAGE_THUNK_DATA*)PtrFromRva(dosHeader,intOfset);
        for (;origThunk->u1.Function;origThunk++,thunk++)
        {
                if (origThunk->u1.Ordinal&IMAGE_ORDINAL_FLAG)
                {
                        if (IS_INTRESOURCE(targetProc) && IMAGE_ORDINAL(origThunk->u1.Ordinal)==(uintptr_t)targetProc)
                                break;
                }
                else
                {
                        IMAGE_IMPORT_BY_NAME *import=(IMAGE_IMPORT_BY_NAME*)PtrFromRva(dosHeader,origThunk->u1.AddressOfData);
                        if (!IS_INTRESOURCE(targetProc) && strcmp(targetProc,(char*)import->Name)==0)
                                break;
                }
        }
        if (origThunk->u1.Function)
        {
                IatHookData *hook=g_IatHooks+g_IatHookCount;
                g_IatHookCount++;
                hook->jump[0]=hook->jump[1]=0x90; // NOP
                hook->jump[2]=0xFF; hook->jump[3]=0x25; // JUMP
#ifdef _WIN64
                hook->jumpOffs=0;
#else
                hook->jumpOffs=(DWORD)(hook)+8;
#endif
                hook->newProc=newProc;
                hook->oldProc=(void*)thunk->u1.Function;
                hook->thunk=thunk;
                DWORD oldProtect;
                VirtualProtect(&thunk->u1.Function,sizeof(void*),PAGE_READWRITE,&oldProtect);
                thunk->u1.Function=(DWORD_PTR)hook;
                VirtualProtect(&thunk->u1.Function,sizeof(void*),oldProtect,&oldProtect);
                return hook;
        }
        return NULL;
}
```

However, it is always interesting to see where a function uses its inputs.

# Intensive uses of an object

Another pattern that comes up often and that teaches a lot about a piece of code is an intensive use of a word in a portion of the code, and very few usages outside of this portion. This can mean that this portion of code is focused on using a particular object, which clarifies the responsibilities of the portion of code.

Let's illustrate it on another example:

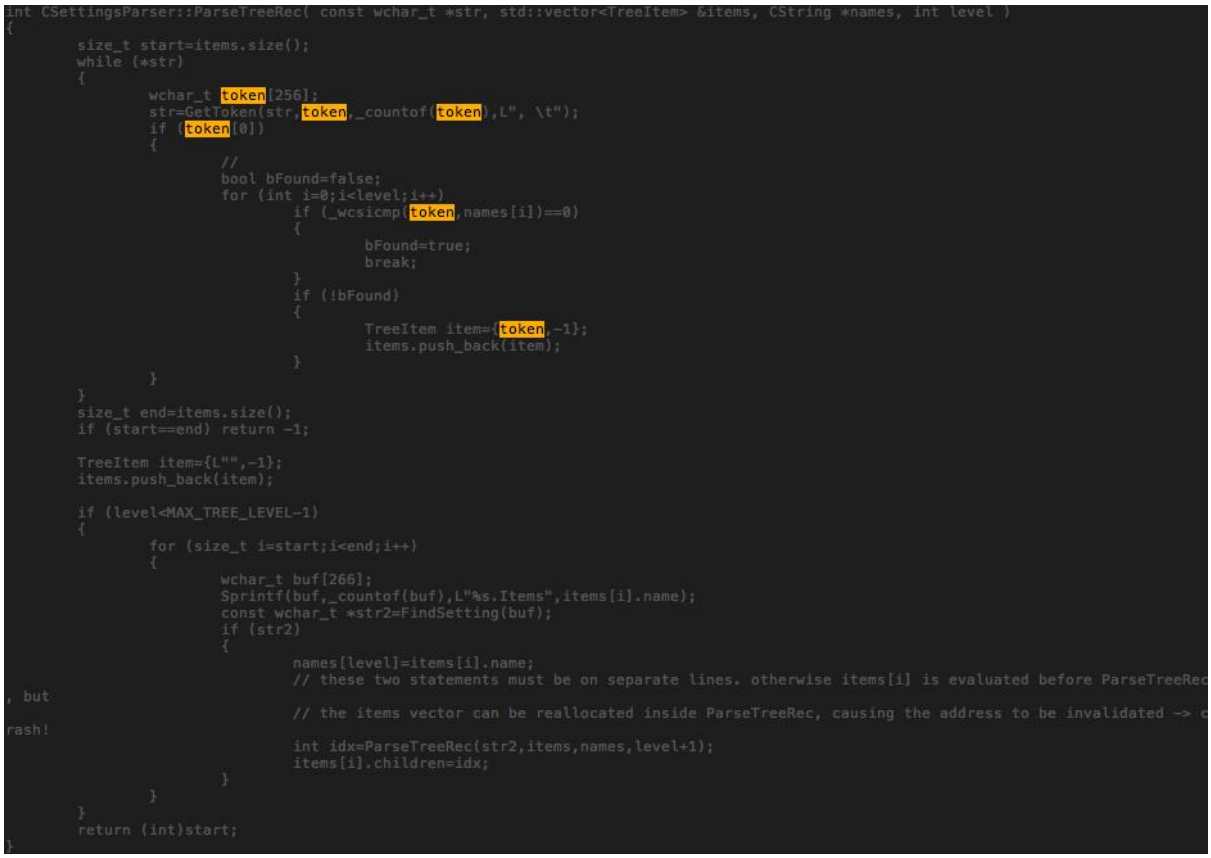
```
int CSettingsParser::ParseTreeRec( const wchar_t *str,
std::vector<TreeItem> &items, CString *names, int level )
{
        size_t start=items.size();
        while (*str)
        {
                wchar_t token[256];
                str=GetToken(str,token,_countof(token),L", \t");
                if (token[0])
                {
                        //
                        bool bFound=false;
                        for (int i=0;i<level;i++)
                                if (_wcsicmp(token,names[i])==0)
                                {
                                        bFound=true;
                                        break;
                                }
                                if (!bFound)
                                {
                                        TreeItem item={token,-1};
                                        items.push_back(item);
                                }
                }
        }
        size_t end=items.size();
        if (start==end) return -1;

        TreeItem item={L"",-1};
        items.push_back(item);

        if (level<MAX_TREE_LEVEL-1)
        {
                for (size_t i=start;i<end;i++)
                {
                        wchar_t buf[266];

        Sprintf(buf,_countof(buf),L"%s.Items",items[i].name);
                        const wchar_t *str2=FindSetting(buf);
                        if (str2)
                        {
                                names[level]=items[i].name;
                                // these two statements must be on separate
lines. otherwise items[i] is evaluated before ParseTreeRec, but
                                // the items vector can be reallocated
inside ParseTreeRec, causing the address to be invalidated -> crash!
                                int
idx=ParseTreeRec(str2,items,names,level+1);
                                items[i].children=idx;
                        }
                }
        }
        return (int)start;
}
```

One of the terms that comes up frequently in the function is `token`. Let's see where this term appears in the function's code:



```
int CSettingsParser::ParseTreeRec( const wchar_t *str, std::vector<TreeItem> &items, CString *names, int level )
{
	size_t start=items.size();
	while (*str)
	{
		wchar_t token[256];
		str=GetToken(str,token,_countof(token),L", \t");
		if (token[0])
		{
			//
			bool bFound=false;
			for (int i=0;i<level;i++)
				if (_wcsicmp(token,names[i])==0)
				{
					bFound=true;
					break;
				}
				if (!bFound)
				{
					TreeItem item={token,-1};
					items.push_back(item);
				}
		}
	}
	size_t end=items.size();
	if (start==end) return -1;

	TreeItem item={L"",-1};
	items.push_back(item);

	if (level<MAX_TREE_LEVEL-1)
	{
		for (size_t i=start;i<end;i++)
		{
			wchar_t buf[266];
			Sprintf(buf,_countof(buf),L"%s.Items",items[i].name);
			const wchar_t *str2=FindSetting(buf);
			if (str2)
			{
				names[level]=items[i].name;
				// these two statements must be on separate lines. otherwise items[i] is evaluated before ParseTreeRec
, but
				// the items vector can be reallocated inside ParseTreeRec, causing the address to be invalidated -> c
rash!
				int idx=ParseTreeRec(str2,items,names,level+1);
				items[i].children=idx;
			}
		}
	}
	return (int)start;
}
```

Since `token` appears many times in the `while` loop, it suggests that it has a central role in that loop. This is good to know if we need to understand what the loop does, and it also suggests a refactoring: why not putting some of the body of the loop in a function that takes `token` as an input parameter?

# There is plenty left to discover

The three above techniques help in the understanding of code by quickly giving high level information about it. This big picture of a piece of code also suggests some refactoring tasks to improve it.

But there is more to word counting. Based on the discussions I had with people around me, I'd like to go further by exploring these ideas:

- counting the individual words inside of a camelCaseSymbol,
- trying word counting with sensitive/insensitive case,
- performing word counts at the level of a module, across multiple files.

Also, in future posts we will build our own program designed to count words in code, which is not quite the same as counting words in just any text. We will use the STL algorithms to code up this program.

Do you think counting words can be useful to understand your codebase? How do you think we should improve the above techniques?

# Word Counting in C++: Implementing a Simple Word Counter

Word counts can reveal information about your code, or make an unknown piece of code more expressive to your eyes.

There are online tools to count words in generic text, but most of those I've come across are designed around counting words in text and SEO (Search Engine Optimization). Since analysing source code is not the same thing as analysing the text of a blog post, let's design a tool fit for our needs of counting words in code. This way, we will be able to make it evolve when we discover new tasks to try with our word counter.

Another reason to write our own word counter is that it will let us practice interface design, and also STL algorithms, which are useful to master for coding in C++.

For this first version of our word counter, the objective will be to put together a working prototype. Then we will improve it over future posts, by adding features and by refining its implementation.

## Code that counts words in code

To make a function that performs the word counting of a piece of code, let's start by designing its interface.

### The interface

One possible interface for a function counting words in code could be this:

```
std::map<std::string, size_t> getWordCount(std::string const& code);
```

The code is input as a `std::string`, and the output word count associates individual words to their number of occurrences. The individual words can be represented as `std::string`, and their number of occurrences is a positive number that can be represented by a `size_t`. It is therefore natural to use a `std::map<std::string, size_t>`.

However, this natural return type may not be exactly what we want: one of the points of a word count is to identify the frequent words, and a map is not designed for doing this. A more appropriate structure would be a `std::vector<std::pair<std::string, size_t>>`, because we can sort it by the number of occurrences (the `second` of its pairs).

Since we see that right from the beginning that defining the type representing the word count is not trivial, let's not settle for a definitive type. Instead let's give it a name, `WordCount`, and use an alias declaration of a `std::vector`. It will make it easier to change it later if necessary, and the advantage of using an alias over a fully-fledged type is that we benefit from all the interface of `std::vector` without writing any additional code:

```
using WordCount = std::vector<std::pair<std::string, size_t>>;
WordCount getWordCount(std::string const& code);
```

Now that we have an interface to begin with, let's move on to the implementation. The basic structure of the function will be:

- identifying all the symbols in the input piece of code,
- counting the occurrences of each one of them,
- sorting the results by decreasing order of occurrences.

## Identifying all the symbols in the input piece of code

Each programming language defines a set of characters that can be used in symbols. In C++, valid symbols are constituted of alphanumerical characters (a to z, A to Z and 0 to 9), as well as underscores (_). A symbols is a succession of such characters, and it stops at any character that is not in this set. For examples, symbols in C++ code are separated by all sorts of white space (space, new lines, tabs) operators (`.`, `+`, `->`, etc.) and brackets (`[]`, `{}`, `()`).

So identifying the symbols in a piece of code represented by a string consists in splitting the string, by using as a delimiter any character that is not a-z, A-Z, 0-9 or an underscore.

The easiest way to split a string in C++ is to use Boost.Split:

```
auto symbols = std::vector<std::string>{};
boost::split(symbols, code, isDelimiter);
```

This outputs in `symbols` the collection of words in the string `code`, delimited by characters satisfying the predicate `isDelimiter`. Let's now implement `isDelimiter`:

```
bool isDelimiter(char c)
{
    auto const isAllowedInName = isalnum(c) || c == '_';
    return !isAllowedInName;
}
```

A delimiter is a character that is not allowed in a name. And as said above, the characters allowed in names are the alphanumerical ones, identified by the standard function `isalnum`, and underscores.

We now have list of all symbols between delimiters. However this collection contains too many entries: when there are two consecutive delimiters, such as `->` or `||` or `).` for example, it generates an empty string corresponding to the (empty) word between those delimiters.

We therefore have to clear our results from empty strings. If you have wrapped the C++ erase-remove idiom in a function, you can write something like this:

```
erase(symbols, "");
```

Otherwise you have to write it out in full:

```
symbols.erase(std::remove(begin(symbols), end(symbols), ""),
end(symbols));
```

This extra step suggests that Boost.Split may not be the right tool here, and that we will have to write our own function to delimit words at some point. We will do it in a future post, but for the moment let's move on to have a working version, that we can start to use and unit test. We will come back to it afterwards.

## Counting the occurrences of each symbol

At this point of the function, we have a `std::vector<std::string>` that contains all the symbols in the function, and we need to count the occurrences of each one of them. Let's create a sub-function in charge of this operation:

```
std::map<std::string, size_t> countWords(std::vector<std::string>
const& words)
{
    auto wordCount = std::map<std::string, size_t>{};
    for (auto const& word : words)
    {
        ++wordCount[word];
    }
    return wordCount;
}
```

This function iterates over the collection of symbols, and increments the number of occurrences of each symbol that we store in a map. Note that the expression `wordCount[word]` creates an entry in the map with a key equal to the `word` if it doesn't exist in the map already.

## Sorting the results by decreasing order of occurrences

Now that we have a map that associates symbols with their number of occurrences, we need to turn it into a `WordCount` sorted by decreasing number of occurrences.

Since `WordCount` is a vector of `std::pairs`, and that a `std::map` is also a container of `std::pair`, we can leverage on the range constructor of `std::vector`. To differentiate the word count that we will sort, let's call it `sortedWordCount` (even though it is not sorted just yet):

```
auto const wordCount = countWords(words);
auto sortedWordCount = WordCount(begin(wordCount), end(wordCount));
```

We finish up the function by sorting the vector in decreasing order of the `.second` of its elements:

```
std::sort(begin(sortedWordCount), end(sortedWordCount),
[](auto const& p1, auto const& p2){ return p1.second >
p2.second; });
```

## Putting it all together

Here is all the code contributing to the function `getWordCount`:

```
bool isDelimiter(char c)
{
    auto const isAllowedInName = isalnum(c) || c == '_';
    return !isAllowedInName;
}

std::map<std::string, size_t> countWords(std::vector<std::string>
const& words)
{
    auto wordCount = std::map<std::string, size_t>{};
    for (auto const& word : words)
    {
        ++wordCount[word];
    }
    return wordCount;
}

WordCount getWordCount(std::string const& code)
{
    auto symbols = std::vector<std::string>{};
    boost::split(symbols, code, isDelimiter);
    symbols.erase(std::remove(begin(symbols), end(symbols), ""),
end(symbols));

    auto const wordCount = countWords(symbols);

    auto sortedWordCount = WordCount(begin(wordCount),
end(wordCount));
    std::sort(begin(sortedWordCount), end(sortedWordCount), [](auto
const& p1, auto const& p2){ return p1.second > p2.second; });

    return sortedWordCount;
}
```

# Constructing a raw string literal from code

If we have a piece of code to analyse with our word counter, how do we make it reach the `getWordCount` function? In later revisions of the program we will fetch code from a file, end even from multiple files, but for the moment let's go for the simplest solution possible: putting the input right into code.

This is not the cleanest and definitive solution, but it has the advantage of being immediate and doable on the go, if you are not at home and only have access to online compilers such as coliru.

But copy-pasting a piece of code into a `std::string` is challenging, because if the code has quotes (`"`) you need to escape them. Also, you have to deal with line returns if your code spreads over several lines (which it likely does).

Fortunately, C++11 raw string literals solve exactly that kind of problems. There are several ways to create a raw string literal, but the simplest one is to write an `R` before opening the quotes, and putting the string inside parentheses: `R"(this is my text with "quotes")"`.

Here is the raw string literal corresponding to the code we have written so far:

```
   static constexpr auto code = R"(

bool isDelimiter(char c)
{
    auto const isAllowedInName = isalnum(c) || c == '_';
    return !isAllowedInName;
}

std::map<std::string, size_t> countWords(std::vector<std::string>
const& words)
{
    auto wordCount = std::map<std::string, size_t>{};
    for (auto const& word : words)
    {
        ++wordCount[word];
    }
    return wordCount;
}

WordCount getWordCount(std::string const& code)
{
    auto symbols = std::vector<std::string>{};
    boost::split(symbols, code, isDelimiter);
```

```
    symbols.erase(std::remove(begin(symbols), end(symbols), ""),
end(symbols));

    auto const wordCount = countWords(symbols);

    auto sortedWordCount = WordCount(begin(wordCount),
end(wordCount));
    std::sort(begin(sortedWordCount), end(sortedWordCount), [](auto
const& p1, auto const& p2){ return p1.second > p2.second; });

    return sortedWordCount;
}
})";
```

# Printing a word count

To start exploiting the information provided by word counts, we will output them to the console. To do this, let's write a function that prints a word count as a table of two columns, with the symbols on one side and their numbers of occurrences on the other side.

To do this by using standard C++ (before C++20, that could adopt the popular {fmt} library), we will rely on stream operations, which you can read about in The Complete Guide to Building Strings In C++:

```
void print(WordCount const& entries)
{
    for (auto const& entry : entries)
    {
        std::cout << std::setw(30) << std::left << entry.first <<
'|' << std::setw(10) << std::right << entry.second << '\n';
    }
}
```

This function fixes the sizes of the two columns to 30 and 10 characters respectively. Let's improve it by adapting the size of the first column to the longest symbol size + 1. To do this we need to locate the longest symbol size. We use `std::max_element`, by giving it a predicate to compare the sizes of the `first`s in the pairs in the vector:

```
auto const longestWord = *std::max_element(begin(entries),
end(entries), [](auto const& p1, auto const& p2){ return
p1.first.size() < p2.first.size(); });
auto const longestWordSize = longestWord.first.size();
```

In an empty collection, `std::max_element` returns the `end` of the collection. Since we cannot deference that, we need to deal with this case, for example by using a guard:

```
void print(WordCount const& entries)
{
    if (entries.empty()) return;
    auto const longestWord = *std::max_element(begin(entries),
end(entries), [](auto const& p1, auto const& p2){ return
p1.first.size() < p2.first.size(); });
    auto const longestWordSize = longestWord.first.size();

    for (auto const& entry : entries)
    {
        std::cout << std::setw(longestWordSize + 1) << std::left <<
entry.first << '|' << std::setw(10) << std::right << entry.second <<
'\n';
    }
}
```

# Putting it all together

Here is a working example of word count, on the code of the word counter itself (also available in this coliru):

```
#include <boost/algorithm/string.hpp>
#include <cctype>
#include <iostream>
#include <iomanip>
#include <map>
#include <vector>

bool isDelimiter(char c)
{
    auto const isAllowedInName = isalnum(c) || c == '_';
    return !isAllowedInName;
}

std::map<std::string, size_t> countWords(std::vector<std::string>
const& words)
{
    auto wordCount = std::map<std::string, size_t>{};
    for (auto const& word : words)
    {
        ++wordCount[word];
    }
    return wordCount;
}
```

```
using WordCount = std::vector<std::pair<std::string, size_t>>;

WordCount getWordCount(std::string const& code)
{
    auto symbols = std::vector<std::string>{};
    boost::split(symbols, code, isDelimiter);
    symbols.erase(std::remove(begin(symbols), end(symbols), ""),
end(symbols));

    auto const wordCount = countWords(symbols);

    auto sortedWordCount = WordCount(begin(wordCount),
end(wordCount));
    std::sort(begin(sortedWordCount), end(sortedWordCount), [](auto
const& p1, auto const& p2){ return p1.second > p2.second; });

    return sortedWordCount;
}

void print(WordCount const& entries)
{
    if (entries.empty()) return;
    auto const longestWord = *std::max_element(begin(entries),
end(entries), [](auto const& p1, auto const& p2){ return
p1.first.size() < p2.first.size(); });
    auto const longestWordSize = longestWord.first.size();

    for (auto const& entry : entries)
    {
        std::cout << std::setw(longestWordSize + 1) << std::left <<
entry.first << '|' << std::setw(10) << std::right << entry.second <<
'\n';
    }
}

int main()
{
    static constexpr auto code = R"(

bool isDelimiter(char c)
{
    auto const isAllowedInName = isalnum(c) || c == '_';
    return !isAllowedInName;
}

std::map<std::string, size_t> countWords(std::vector<std::string>
const& words)
{
    auto wordCount = std::map<std::string, size_t>{};
    for (auto const& word : words)
    {
        ++wordCount[word];
    }
```

```
    return wordCount;
}

using WordCount = std::vector<std::pair<std::string, size_t>>;

WordCount getWordCount(std::string const& code)
{
    auto symbols = std::vector<std::string>{};
    boost::split(symbols, code, isDelimiter);
    symbols.erase(std::remove(begin(symbols), end(symbols), ""),
end(symbols));

    auto const wordCount = countWords(symbols);

    auto sortedWordCount = WordCount(begin(wordCount),
end(wordCount));
    std::sort(begin(sortedWordCount), end(sortedWordCount), [](auto
const& p1, auto const& p2){ return p1.second > p2.second; });

    return sortedWordCount;
}

void print(WordCount const& entries)
{
    if (entries.empty()) return;
    auto const longestWord = *std::max_element(begin(entries),
end(entries), [](auto const& p1, auto const& p2){ return
p1.first.size() < p2.first.size(); });
    auto const longestWordSize = longestWord.first.size();

    for (auto const& entry : entries)
    {
        std::cout << std::setw(longestWordSize + 1) << std::left <<
entry.first << '|' << std::setw(10) << std::right << entry.second <<
'\n';
    }
}

})";
    print(getWordCount(code));
}
```

Here is the word count output by this program:

```
std             |        20
auto            |        13
const           |        13
symbols         |         7
return          |         6
wordCount       |         6
string          |         6
entries         |         5
```

```
end              |         5
p2               |         4
p1               |         4
first            |         4
sortedWordCount  |         4
begin            |         4
WordCount        |         4
c                |         3
size_t           |         3
vector           |         3
entry            |         3
size             |         3
second           |         3
map              |         2
longestWord      |         2
longestWordSize  |         2
setw             |         2
word             |         2
words            |         2
isDelimiter      |         2
isAllowedInName  |         2
code             |         2
countWords       |         2
for              |         2
erase            |         1
10               |         1
_                |         1
bool             |         1
void             |         1
boost            |         1
using            |         1
char             |         1
split            |         1
cout             |         1
sort             |         1
empty            |         1
1                |         1
getWordCount     |         1
right            |         1
if               |         1
remove           |         1
print            |         1
pair             |         1
n                |         1
max_element      |         1
isalnum          |         1
left             |         1
```

The most frequent words is `std`, which reflects that we've used the standard library quite intensively. Among the frequent words not related to C++, we find `symbols` and `wordCount`, which is indeed what this code is about.

# Next steps

Now that we have a working (as far as I know!) word counter, we can make it evolve.

One interesting feature for counting words in code is to extract individual words from camelCaseSymbols. To do this we will implement our own function to extract words from code, and at the same time use an implementation more adapted than Boost.Split.

# Word Counting in C++: Extracting words from camelCase symbols

Counting words in code, what an exciting topic!

Ok, if you don't see what exactly is exciting in counting words in code, maybe a little context will help. Word counts can reveal useful information about a piece of code, and with the right tooling, it takes very little time to perform.

Reading code is one of our principal activities as software developers, and being able to quickly make sense of a unknown piece of code in an invaluable skill. I believe that word counts can help doing that. If you'd like to see what sort of things they can reveal about code, you can check out the introductory post about word counts.

And speaking about the right tooling, this post, along with a few others before and after it, is about programming a word counter in C++, which happens to be an interesting task in itself, as it shows practical usages of the STL.

Now are you excited about word counts?

## A word counter in camelCase

In the last episode we left off with a word counter that could make a list of the words in a piece of code, with their number of occurrences associated. We will take its implementation as a starting point. Now, we are going to extract the words inside of the symbols in camel case of the piece of code.

A word in camel case is a concatenation of several words all starting with a capital letter, except for the first one. For example, `thisIsAWordInCamelCase`. But we will also include the symbols that start with a capital letter, which is *strict sensu* called Pascal case. For example `ThisIsAWordInPascalCase`.

If the above two examples appeared in a piece of code, with our previous word counter they would have generated the following word count:

```
ThisIsAWordInCamelCase |         1
thisIsAWordInCamelCase |         1
```

With the word counter that we will implement now, they would generate the following word count:

```
A     |         2
Camel |         2
Case  |         2
In    |         2
Is    |         2
Word  |         2
This  |         1
this  |         1
```

# Extracting the words

Let's start by coding a function that takes a piece of code (represented by a `std::string`), and extracts all the individual words inside of all the camel (or Pascal) case symbols in it. We will use this function instead of the current code that extracts the words in code which, as a reminder, was this:

```
auto symbols = std::vector<std::string>{};
boost::split(symbols, code, isDelimiter);
```

To start experimenting with a working word counter, we had used Boost Split even if it forced us to remove the empty words afterwards. Now we will replace these three lines of code with a call to our function extracting words from code in camel case. Here is its interface:

```
std::vector<std::string> getCamelCaseWordsFromCode(std::string
const& code);
```

## The algorithm

To extract a given word inside a piece of code, we need to work out two things: where the word starts, and where it ends. And since we need to do this for each word, there will probably be some kind of loop involved.

So to break down the implementation of the algorithm in small steps, we'll proceed in two steps:

- Step 1: start by writing code to extract the first word,
- Step 2: adapt this code to loop over all the words.

Before that, let's create the return value to output:

```
std::vector<std::string> getCamelCaseWordsFromCode(std::string
const& code)
{
    auto words = std::vector<std::string>{};
```

Note that another option would have been to follow the conventions of the STL and use an output iterator. We keep this option in mind if we decide later to make our function more generic.

## Step 1: locating the first word

To locate the first word, we can use two iterators: `beginWord` that points to the first letter of the word, and `endWord` that points to the first letter *after* the word (or the end of `code`). This way, we will be able to manipulate the word like a range (a sub-range of `code`) and use all the interfaces that the STL offers.

The first letter of the first word is not necessarily the first word of the piece of code. Indeed, the code may start with blanks or other characters that are not part of a symbol. The first letter of the word is the first one that is not a delimiter. We can locate it by using the STL algorithm `std::find_if_not`:

```
auto const beginWord = std::find_if_not(begin(code), end(code),
isDelimiter);
```

We can use the `isDelimiter` function we had used in our previous implementation of a simple word counter:

```
bool isDelimiter(char c)
{
    auto const isAllowedInName = isalnum(c) || c == '_';
    return !isAllowedInName;
}
```

A delimiter is anything that is not in a name, and names in C++ are made of alphanumerical characters (a-z, A-Z, 0-9) and underscores (_).

Now we need to find the end of the first word. A word can end with two things:

- either a delimiter,
- or a capital letter, that marks the beginning of a new word inside of a symbol in camel case.

So we're looking for the first character after `beginWord` that is either one of these two things. We can use the convenient `std::next` function to start looking after the first letter of the word:

```
auto const endWord = std::find_if(std::next(beginWord), end(code),
[](char c){ return isDelimiter(c) || isupper(c); });
```

Indeed, if we had started searching for a capital letter from `beginWord`, and that `beginWord` happened to point to a capital letter itself, the search would not have gone past the first letter, which may not be the end of the word.

Also note that if we call `std::next` on the `end` of a container, using the returned value leads to undefined behaviour. We therefore have to check that we are not at the end of the piece of code before executing the above line of code.

## Combining functions

I don't know what you think, but I find the expression `[](char c){ return isDelimiter(c) || isupper(c); }` rather annoying to write and read, because it contains a lot of noise. It would have been nicer to write something like this:

```
auto const endWord = std::find_if(std::next(beginWord), end(code), isDelimiter || isupper);
```

But this is not legal C++. Boost Phoenix would have allowed to write something like this, after some declarations involving macros:

```
auto const endWord = std::find_if(std::next(beginWord), end(code), isDelimiter(arg1) || isupper(arg1));
```

There may be other ways to write this, but we'd risk straying from our exciting topic of word counting if we go further. We will explore the combinations of functions in another post. You're welcome to share your suggestions about this topic in the comments section below.

## Extracting the word

Now that we have located the word with `beginWord` and `endWord`, we need to send it to the output collection, `words`. To do this, we could use the constructor of `std::string` that takes two iterators to construct a `std::string`, and add it to the `std::vector` by using `push_back`.

But a more direct way is to use the `emplace_back` method of `std::vector`, that accepts constructors arguments to directly construct the new object in the memory space of the vector (using a placement `new`), thus avoiding a copy:

```
words.emplace_back(beginWord, endWord);
```

The compiler may have been able to optimize away the copy, but `emplace_back` leads to more direct code anyway. `emplace_back` has been added to the standard in C++11.

## Step 2: looping over the words

After a series of trials and errors, I could come up with the following loop: find `beginWord` before the loop, then repeat the finding of `endWord` and the `beginWord` for the next word:

```
auto beginWord = std::find_if_not(begin(code), end(code),
isDelimiter);
while (beginWord != end(code))
{
    auto endWord = std::find_if(std::next(beginWord), end(code),
[](char c){ return isDelimiter(c) || isupper(c); });
    words.emplace_back(beginWord, endWord);
    beginWord = std::find_if_not(endWord, end(code), isDelimiter);
}
```

I don't claim that it is the optimal solution, in particular because it duplicates the code performing the search of the beginning of a word, and I'd be glad to hear your suggestions to improve it, and potentially to simplify it by using STL algorithms.

We can now integrate this function with our previous word counter. This is done in this coliru, which you can use to play around and count the words in your code using camel and pascal case.

# Next up: parametrization

We now have a word counter that counts the words inside of camel case symbols, but that no longer counts the whole words! This also was a valid way to count words.

The next step will be to allow our word counter to perform both types of counts. This will make us reflect on:

- how to mutualize code,
- how to design an expressive interface that allows to choose between types of treatments.

# Word Counting in C++: Parametrizing the Type of Case

In our first step implementing a word counter in C++, we wrote code that could extract the words inside of a piece of code. In the second step, we changed that code so that it extracted individual words inside of camelCaseSymbols (and also of PascalCaseSymbols), losing the previous feature of counting entire words.

Today, we are going to make the code able to do either type of extraction, entire words or words inside of camel case symbols. This will make us practice two aspects of writing expressive code:

- avoiding code duplication,
- designing a clear API to choose between various treatments (here, between entire words and camel case).

The reason why we're building a word counter in the first place is that counting words can reveal useful information about a piece of code, and also because implementing it is an instructive project to improve our coding skills in C++.

## Summary of the previous episodes

In the first version of the word counter, we went for the quickest solution to have a working prototype. For this, we used Boost Split to extract entire words, even though it wasn't the most adapted tool for our purpose, as it needed a second pass to remove the empty words:

```
auto symbols = std::vector<std::string>{};
boost::split(symbols, code, isDelimiter);
symbols.erase(std::remove(begin(symbols), end(symbols), ""),
end(symbols));
```

Where `isDelimiter` is a function that determines whether a given character is a delimiter, meaning that it cannot be part of a C++ name:

```
bool isDelimiter(char c)
{
    auto const isAllowedInName = isalnum(c) || c == '_';
    return !isAllowedInName;
}
```

C++ names are made of alphanumerical characters and underscores. Other characters (brackets, ., -, >, +, spaces, etc.) are delimiters. And in code with several delimiters in a row (like with ->), that leads to empty words (between - and >)

This solution, although quick to put in place, did not have the flexibility to extract words from symbols in camel or pascal case. So we had to implement our own extraction:

```
std::vector<std::string> getCamelCaseWordsFromCode(std::string
const& code)
{
    auto words = std::vector<std::string>{};
    auto beginWord = std::find_if_not(begin(code), end(code),
isDelimiter);
    while (beginWord != end(code))
    {
        auto const endWord = std::find_if(std::next(beginWord),
end(code), [](char c){ return isDelimiter(c) || isupper(c); });
        words.emplace_back(beginWord, endWord);
        beginWord = std::find_if_not(endWord, end(code),
isDelimiter);
    }
    return words;
}
```

If you'd like to have more details about how we came up with this function, you can check out the dedicated post. This function essentially locates the beginning and and of each word, and places them inside of the output vector.

# From words in camel case to entire words

What is the difference between locating an entire word and locating a word inside of a symbol in camel case?

Both start by a character that is not a delimiter. Where they differ is with their end: words inside of a camel case symbol end when we encounter a capital letter (which is the beginning of the next word) or a delimiter (end of the whole camel case symbol). Entire words can only end with a delimiter.

There is one place in the above function where we check for the end of a word:

```
std::vector<std::string> getCamelCaseWordsFromCode(std::string
const& code)
{
    auto words = std::vector<std::string>{};
    auto beginWord = std::find_if_not(begin(code), end(code),
isDelimiter);
    while (beginWord != end(code))
    {
        auto const endWord = std::find_if(std::next(beginWord),
end(code), [](char c){ return isDelimiter(c) || isupper(c); });
        words.emplace_back(beginWord, endWord);
        beginWord = std::find_if_not(endWord, end(code),
isDelimiter);
    }
    return words;
}
```

To split on entire words, we therefore only need to change that predicate:

```
std::vector<std::string> getEntireWordsFromCode(std::string const&
code)
{
    auto words = std::vector<std::string>{};
    auto beginWord = std::find_if_not(begin(code), end(code),
isDelimiter);
    while (beginWord != end(code))
    {
        auto const endWord = std::find_if(std::next(beginWord),
end(code), isDelimiter);
        words.emplace_back(beginWord, endWord);
        beginWord = std::find_if_not(endWord, end(code),
isDelimiter);
    }
    return words;
}
```

From that point it becomes natural to make only one function that takes the predicate identifying the end of a word:

```
template<typename EndOfWordPredicate>
```

```
std::vector<std::string> getWordsFromCode(std::string const& code,
EndOfWordPredicate isEndOfWord)
{
    auto words = std::vector<std::string>{};
    auto beginWord = std::find_if_not(begin(code), end(code),
isDelimiter);
    while (beginWord != end(code))
    {
        auto const endWord = std::find_if(std::next(beginWord),
end(code), isEndOfWord);
        words.emplace_back(beginWord, endWord);
        beginWord = std::find_if_not(endWord, end(code),
isDelimiter);
    }
    return words;
}
```

# The client interface

We want the user of our word counter to choose between entire words and words inside camel case. The interface as it is is too low in terms of levels of abstraction: we want the user to express his choice by writing something like `EntireWords` or `WordsInCamelCase`, and not by passing a predicate. Therefore we need an additional indirection to raise the level of abstraction.

This higher level of abstraction can consist in a function where the user passes its `code`, as well as an indication about `EntireWords` or `WordsInCamelCase`. The question now is, how to express that latter indication?

The purpose of our function is to take a piece of code and extract the words in it. Its sole natural input is the piece of code. The *way* we want it to perform that extract is a different form of input. It is more something that *parametrize* the function than a regular input. As if two types of extraction would really be two different functions.

To express this, I think we should pass the type of extraction to the function via a different channel than its normal input. We have at least two channels for this: template parameters, and currying.

## Template parameters

Template parameters incur a constraint: they have to be specified at compile time.

Our template parameter should be able to take two values, one for entire words and one for words in camel case. To represent this, we can use an `enum`:

```
enum class HowToDelimitWords
{
    EntireWords,
    WordsInCamelCase
};
```

Then we use it as a template parameter, in the header file:

```
template<HowToDelimitWords howToDelimitWords>
std::vector<std::string> getWordsFromCode(std::string const& code);
```

Note that since we don't use the template parameter inside of the declaration, we can omit its name, which was redundant:

```
template<HowToDelimitWords>
std::vector<std::string> getWordsFromCode(std::string const& code);
```

Also note that if we provide the implementations for both values of the enum class, we don't have to write them in the header file. We can use a .cpp file and the linker will find them there:

```
template<HowToDelimitWords>
std::vector<std::string> getWordsFromCode(std::string const& code);

template<>
std::vector<std::string>
getWordsFromCode<HowToDelimitWords::EntireWords>(std::string const&
code)
{
    return getWordsFromCode(code, isDelimiter);
}

template<>
std::vector<std::string>
getWordsFromCode<HowToDelimitWords::WordsInCamelCase>(std::string
const& code)
{
```

```
    return getWordsFromCode(code, [](char c){ return isDelimiter(c)
|| isupper(c); });
}
```

You can find all the code put together in this coliru.

## Currying

Currying means partial application of a function. Here we will use currying to choose the type of extraction at runtime.

To do this, we start by passing the type of extraction as a regular function parameter, then we will partially apply the function to fix the type of extraction.

If we pass the enum as a regular function parameter, our function becomes:

```
std::vector<std::string> getWordsFromCode(std::string const& code,
HowToDelimitWords howToDelimitWords)
{
    if (howToDelimitWords == HowToDelimitWords::EntireWords)
    {
        return getWordsFromCode(code, isDelimiter);
    }
    else
    {
        return getWordsFromCode(code, [](char c){ return
isDelimiter(c) || isupper(c); });
    }
}
```

And its declaration in the header file becomes:

```
std::vector<std::string> getWordsFromCode(std::string const& code,
HowToDelimitWords howToDelimitWords);
```

Since we would like to make the function take only the `code` as parameter, we can resort to partially apply it with lambdas. Note that we can write the lambdas *in the header file*, with only the function declaration available:

```
std::vector<std::string> getWordsFromCode(std::string const& code,
HowToDelimitWords howToDelimitWords);

auto const getEntireWordsFromCode = [](std::string const& code){
return getWordsFromCode(code, HowToDelimitWords::EntireWords); };
```

```
auto const getWordsInCamelCaseFromCode = [](std::string const&
code){ return getWordsFromCode(code,
HowToDelimitWords::WordsInCamelCase); };
```

We now have two functions, `getEntireWordsFromCode` and `getWordsInCamelCaseFromCode`, that both take only one parameter, `code`. And we avoided code duplication.

You can find all the code using currying put together in that coliru.

The option using lambda is perhaps less scalable than the one using templates, if we add other parameters. At this stage though, we don't know if we will ever need extra parameters. And if we do, we will always be able to adapt the code, or use wrappers in the worst case.

Which option do you prefer?

# Next steps

We have now allowed a user of our word counter to choose between counting entire words and counting individual words in camel case.

The next features we will implement include performing case-insensitive word counts as well as word counts on multiple files at the same time. This will let us practice other aspects of code design.

# Word Counting in C++: Computing the Span of a Word

Here is a new episode in the series of word counting! Today we will focus on computing the **span** words in code.

As a reminder, word counting consists in counting the occurrences of every term in a piece of code (for example, in a function), and sorting the results by most frequent words. This can reveal at a glance useful information about that piece of code.

Over the past few posts, we've been building a word counter in C++. We're investing time in this project for several reasons:

- it is an opportunity to practice with the STL,
- it is an opportunity to practice with interface design,
- we have a more an more complete word counter to use on our code.

## The span of words

Today we add a new feature to our word counter: computing the span of words! The span of a term in a piece of code is the number of lines over which it spreads. For example, consider the following piece of code:

```
int i = 42;
f(i);
f(i+1)
std::cout << "hello";
++i;
```

The span of `f` is 2, the span of `i` is 5 and the span of`cout` is 1.

The span of a word is an interesting measure because it indicates how spread out the word is in a piece of code: are all its usage located in the same area? Is it used throughout the function? Such are the questions that can be answered by measuring the span of that word.

Combined with the count of occurrences of a word (a feature that our word counter already has), the span can measure the **density** of a term. If a word has a high number of occurrences and a low span, it means that its usages are all crammed in a part of a function:

Knowing such a piece of information brings at least two things:

- quickly knowing what a part of the code is about,
- suggesting a refactoring task (taking away that part of the code in a separate function).

# Computing the span of a word

Let's pick up the word counter where we left it off.

The basic design of our word counter was to extract the successive words in the piece of code, and then to count the number of occurrences of each of those words:


As you can see, in that first implementation we used standard types, such as `string` for the extracted words and `size_t` for their number of occurrences.

To implement the span, we will need to extract and process more information (about line numbers in particular), so this implementation won't hold. We need to make it more robust, by replacing the raw standard types by dedicated classes:


The data extracted from the code is now called `WordData`, and the aggregates computed from this data for each word is now `WordStats`. At this stage, `WordData` and `WordStats` are simple encapsulations of their standard types equivalents:

```
class WordData
{
public:
    explicit WordData(std::string word);
    std::string const& word() const;
private:
    std::string word_;
};

class WordStats
{
public:
    WordStats();
    size_t nbOccurrences() const;
    void addOneOccurrence();
private:
    size_t nbOccurrences_;
};
```

If we didn't want to go further than this, we could have considered using strong types instead of defining our own classes. But the point here is to add new features to the class, so we'll stick with regular classes.

## Extracting line numbers

Our current code for extracting words from code is this:

```
template<typename EndOfWordPredicate>
std::vector<WordData> getWordDataFromCode(std::string const& code,
EndOfWordPredicate isEndOfWord)
{
    auto words = std::vector<WordData>{};
    auto beginWord = std::find_if_not(begin(code), end(code),
isDelimiter);
    while (beginWord != end(code))
    {
        auto const endWord = std::find_if(std::next(beginWord),
end(code), isEndOfWord);
        words.emplace_back(std::string(beginWord, endWord));
        beginWord = std::find_if_not(endWord, end(code),
isDelimiter);
    }
    return words;
}
```

The `isEndOfWord` predicate checks for the end of word that can be either a capital letter for words inside of camel case symbols, or a delimiter in all cases.

And `isDelimiter` indicates if a character is not part of a word:

```
bool isDelimiter(char c)
{
    auto const isAllowedInName = isalnum(c) || c == '_';
    return !isAllowedInName;
}
```

This code extracts the words of the piece of code. We would now like to also make it extract the line numbers of those words. We will then be able to compute the span, as being the distance between the first line and the last one.

A simple way to work out the line number of a given word is to compute the number of line returns from the beginning of the piece of code an until that word. But doing this for each

word makes for a quadratic number of reads of the characters of the piece of code. Can we do better than quadratic?

We can if we count the number of line returns since the end of the previous word, and add this to the line number of the previous word. This has a linear complexity, which is much better than quadratic complexity.

We could consider going further by checking every character only once, and find the beginning of the next word AND the number of line returns until then, all in one single pass. But that would lead to more complex code. So we will suffice with the above linear algorithm, even it is makes several reads of the same characters. We keep the code simple until there is a compelling reason not to do so (for example, a poor performance which profiling indicates that we should go for a more elaborate algorithm).

Here is the code updated in that sense:

```
template<typename EndOfWordPredicate>
std::vector<WordData> getWordDataFromCode(std::string const& code,
EndOfWordPredicate isEndOfWord)
{
    auto words = std::vector<WordData>{};
    auto endWord = begin(code);
    auto beginWord = std::find_if_not(begin(code), end(code),
isDelimiter);
    size_t line = 0;

    while (beginWord != end(code))
    {
        auto const linesBetweenWords = std::count(endWord,
beginWord, '\n');
        line += linesBetweenWords;
        endWord = std::find_if(std::next(beginWord), end(code),
isEndOfWord);
        words.emplace_back(std::string(beginWord, endWord), line);
        beginWord = std::find_if_not(endWord, end(code),
isDelimiter);
    }
    return words;
}
```

## Computing the span

We now have a collection of `WordData`, that each contains a word an a line number. We now feed this collection to a `std::map<std::string, WordStats>`. The code before taking the span into account looked like this:

```
std::map<std::string, WordStats> wordStats(std::vector<WordData>
const& wordData)
{
    auto wordStats = std::map<std::string, WordStats>{};
    for (auto const& oneWordData : wordData)
    {
        wordStats[oneWordData.word()].addOneOccurrence();
    }
    return wordStats;
}
```

One way to pass line numbers of the words so that `WordStats` can process them is to pass it as an argument to the method `addOneOccurrence`:

```
std::map<std::string, WordStats> wordStats(std::vector<WordData>
const& wordData)
{
    auto wordStats = std::map<std::string, WordStats>{};
    for (auto const& oneWordData : wordData)
    {

wordStats[oneWordData.word()].addOneOccurrence(oneWordData.lineNumbe
r());
    }
    return wordStats;
}
```

`WordStats` should be able to provide a span in the end, so it needs to remember the smallest and highest line numbers where the word appears. To achieve that, we can keep the smallest (resp. highest) line number encountered so far in the `WordStats` and replace it with the incoming line number in `addOneOccurrence` if it is smaller (resp. higher).

But what **initial value** should we give to the smallest and highest line numbers encountered so far? Before giving any line number, those two bounds are "not set". To implement this in C++, we can use `optional` (`std::optional` in C++17, `boost::optional` before):

```
class WordStats : public Comparable<WordStats>
```

```
{
public:
    WordStats();
    size_t nbOccurrences() const;
    void addOneOccurrence(size_t lineNumber);
    size_t span() const;
private:
    size_t nbOccurrences_;
    std::optional<size_t> lowestOccurringLine_;
    std::optional<size_t> highestOccurringLine_;
};
```

With this, the implementation of `addOneOccurrence` can be:

```
void WordStats::addOneOccurrence(size_t lineNumber)
{
    ++nbOccurrences_;
    if (!lowestOccurringLine_) // means that it is the first line
number coming in
    {
        lowestOccurringLine_ = lineNumber;
    }
    else
    {
        lowestOccurringLine_ = std::min(*lowestOccurringLine_,
lineNumber); // the "min" that we were talking about
    }

    // then same thing for the highest line
    if (!highestOccurringLine_)
    {
        highestOccurringLine_ = lineNumber;
    }
    else
    {
        highestOccurringLine_ = std::max(*highestOccurringLine_,
lineNumber);
    }
}
```

Then `span` comes naturally:

```
size_t WordStats::span() const
{
    if (!lowestOccurringLine_ || !lowestOccurringLine_)
    {
        return 0;
    }
    else
    {
        return *highestOccurringLine_ - *lowestOccurringLine_ + 1;
```

```
        }
}
```

# The feature of span

We have highlighted the main part of the design. If you'd like to have a look at the code in its entirety, and play around with the word counter, you will find all the above in this coliru.

The code produces the span of the words, but I certainly don't claim that it's the optimal implementation. Did you see things that you would like to correct in the design, or the implementation?

More generally, do you think that measuring the span of words, as well as their density, is a relevant measure for your code?