

Husky: A C++ Functional Programming Library Tutorial

Contents

1.	Introduction.....	3
2.	Library overview.....	4
3.	Container Manipulation Functions.....	5
3.1	Concatenation	5
3.2	Container Properties Functions.....	6
3.3	Container Manipulation Functions.....	7
3.4	String manipulation functions.....	8
3.5	Zippping functions.....	9
4.	Higher-order Functions	10
4.1	Maps.....	10
4.2	Filters.....	10

1. Introduction

Imperative programming has been the staple of programming languages since its inception. It is a natural way of interacting with a computer, and it follows (unless concurrency comes into play) a defined and straight forward order. Functional programming, by contrast, appears almost artificial, with its mathematical inspiration and style. Why is it then that all the big imperative languages are adding more and more functional capabilities in their new versions?

Functional programming is becoming more and more popular because, for many kinds of tasks (recursive algorithms, for example) it results in much more readable, and, depending on the language, sometimes even more efficient, than the usual imperative implementation. As such, many of its staples are being absorbed by languages such as C++.

This tutorial intends to guide C++ users in the use of the Husky library, a functional programming library for C++ that focuses on providing several higher-order and container manipulation functions inspired by Haskell's Prelude, the language's main module. We assume that the reader has some familiarity with C++11, but otherwise no familiarity at all with Haskell. The design discussion for the library and the complete manuals for every function present in Husky are provided elsewhere.

We intend to explain and exemplify basic uses of Husky's functions at first, then combine them in true functional programming style. For the sake of uniformity, we will use `std::vector` as our standard container, even though most of the functions accept any STL container.

Finally, we now can demonstrate our own version of the traditional "Hello, world" program. It uses one of our functions, `filter`, to process a string and generate another with only the capsized letters of the original. To have access to the library, one only needs to include "*husky.h*" and add the library folder to the compiler options.

```
#include <iostream>
#include <string>
#include "husky.h"

using namespace husky;
using namespace std;

auto caps = [](char c) { return (c >= 65 && c <= 90); };

int main() {
    string str = "HelloUSweetKoalaYou";
    string s = filter(str, caps);
    cout << s << endl;
    return 0;
}
```

0%

C:\WINDOWS\syst

HUSKY

Press any key to continue . . .

2. Library overview

Husky is a function-oriented library. As we intended to extend the STL, and not replace it, we have provided support for the standard containers, and therefore focused our efforts in providing a big set of functions for our users. They are mainly divided in two groups: higher-order functions and container manipulation functions.

Higher-order functions take one or more functions as input and/or output a function itself. They are very useful, as they can be composed to generate more powerful functions. Our library implements all the main higher-order functions, with its many variants. The complete list is indicated in the table below:

maps	<i>map, concatMap</i>
filters	<i>filter, takeWhile, dropWhile, span_container, break_container</i>
folds	<i>foldl, foldl1, foldr, foldr1, any_fold, all_fold,</i>
scans	<i>scanl, scanl1, scanr, scanr1</i>
zips	<i>zipWith, zipWith3</i>
function composition	<i>compose</i>

Table 1 Higher-order functions implemented by Husky

Our other main group of functions are container manipulation ones. They can be used to extract information or manipulate those containers. As strings are considered to be lists of characters in Haskell, and they can be viewed as containers generically in C++, we have added string manipulation functions to our library as well. They are listed in the table below:

concatenation	<i>concat, concat1</i>
list properties	<i>head, last, tail, init, is_null, length, at, elem, notElem</i>
list manipulation	<i>reverse, take, drop, splitAt, cons</i>
special folds	<i>and_fold, or_fold, sum, product, maximum_of, minimum_of</i>
zips	<i>zip, zip3, unzip, unzip3</i>
string manipulation	<i>lines, words, unlines, unwords</i>

Table 2 Container manipulation functions implemented by Husky

As some of these grouped functions have similar applications and effects, we will discuss in this tutorial the most important ones, separately at first and then combining them to generate interesting functions.

3. Container Manipulation Functions

3.1 Concatenation

There are two container concatenation functions in our library, *concat* and *concat1*. The first one behaves as expectedly for a concatenation function, receiving two containers and returning a new container with elements from the first and then the second one, in sequence. The *concat1* function must only be applied to containers of containers, as it returns a new single container with all the elements in the interior containers. It only receives a single argument.

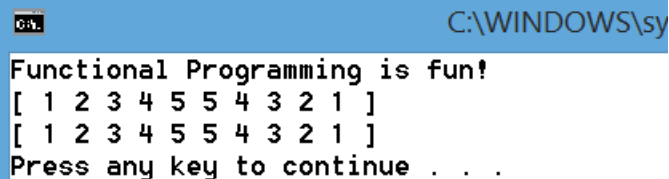
These functions do not change in any way their parameters. That will be a recurrent thing across our library, as we have endeavored to mimic Haskell's lack of side effects, i.e. refraining from changing variables, creating new ones instead. In a future implementation of our library, we intend to provide lazy evaluation to optimize this practice.

To use both *concat*s, we only need to call the function with the proper arguments. In the following examples, we use both functions to unite both lists and strings (as we have stated earlier, for the purpose of this library, strings work as containers of chars). That means that a vector of strings falls into the category of a container of containers, and we can call *concat1* with it to concatenate all the strings in it. The example also shows that using *concat* with two vectors of integers and using *concat1* in a vector of vector of integers that contains both of the previous vector results in the same final vector. The containers are being printed with our library's *printCont* method, which takes a container and prints it in a formatted way.

```
#include <iostream>
#include <string>
#include "husky.h"

using namespace husky;
using namespace std;

int main() {
    vector<string> v1 = { "Functional ", "Programming ", "is ", "fun!" };
    vector<int> v2 = { 1, 2, 3, 4, 5 };
    vector<int> v3 = { 5, 4, 3, 2, 1 };
    vector<vector<int>> vv;
    vv.push_back(v2);
    vv.push_back(v3);
    vector<int> v4 = concat(v2, v3);
    vector<int> v5 = concat1(vv);
    string str = concat1(v1);
    cout << str << endl;
    printCont(v4);
    printCont(v5);
    return 0;
}
```



C:\WINDOWS\sy

```
Functional Programming is fun!
[ 1 2 3 4 5 5 4 3 2 1 ]
[ 1 2 3 4 5 5 4 3 2 1 ]
Press any key to continue . . .
```

3.2 Container Properties Functions

This group of functions generally receive a container and return some property pertaining to that container. The exceptions are the membership functions, `elem` and `notElem`, that also receive an element that may or may not be in the container. Most of these functions are only interfaces over STL functions, and they were added to provide a uniform style over the library. More detailed information can be found in the manual pages.

The most interesting functions in this group are *head*, *tail* and *init*. They are similar to their Haskell siblings: *head* takes a container and returns its first element, *tail* takes a container and returns a new one, with all the elements but the head, and *init* is the opposite operation, returning all the elements but the last one.

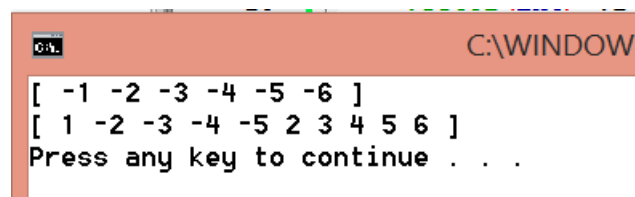
These functions should not be called with empty containers, as they will throw exceptions. They are useful for obtaining the important blocks of a container and using it to build others or pass them to some function.

In the example below we do some container “transplanting” to see these functions’ usage. We have also used as helper functions *concat*, mentioned before, and *cons*, which will be introduced later. For now, *cons* is just a function that takes an element and a container and returns a new container with the element at the head and the other container as tail. We create two new vectors from the first ones, messing with their elements.

```
#include <iostream>
#include <string>
#include "husky.h"

using namespace husky;
using namespace std;

int main() {
    vector<int> v1 = { -1, 2, 3, 4, 5, 6 };
    vector<int> v2 = { 1, -2, -3, -4, -5, -6 };
    vector<int> v3 = cons(head(v1), tail(v2));
    vector<int> v4 = concat(init(v2), tail(v1));
    printCont(v3);
    printCont(v4);
    return 0;
}
```



```
C:\WINDOWS
[ -1 -2 -3 -4 -5 -6 ]
[ 1 -2 -3 -4 -5 2 3 4 5 6 ]
Press any key to continue . . .
```

3.3 Container Manipulation Functions

These functions generally take a container and another parameter (except for *reverse*, which only takes the container itself) and generate a new container (or a pair of vectors, in the case of *splitAt*). The function *cons* mimics the Haskell list constructor. It takes an element and a container, and returns a new container that has the element at its head and the original container at its tail. The function *reverse* does exactly what the name entails, and returns a new container with elements in reverse order.

The *take* and *drop* functions behave similarly. They receive an integer and a container, and return either a prefix of the container of length *n* (in the case of *take*) or the complementary suffix to this prefix (in the case of *drop*). They are very useful for slicing data, a practice that Python programmers are very used to. The *splitAt* function works like a combined version of both, receiving a container and an integer and returning a pair of vectors in which the first would be the result of applying *take* and the second one the result of applying *drop* to the arguments.

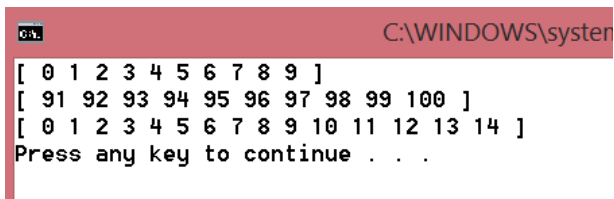
If per chance the integer parameter is bigger than the size of the container, the *take* function will just return the entire container, while the *drop* function will return an empty one. One should be careful then, when choosing the integer parameter.

In this example, we first construct a vector of ints from 1 to 100 using the STL function *iota*. We then construct another vector from the first one, using *cons* and adding a 0 at the front. We then used *take* and *drop* to slice at the vectors, obtaining small subsets from them. Finally we used *splitAt* to break the second vector at the fifteenth position, and printing the first resulting vector.

```
#include <iostream>
#include "husky.h"

using namespace husky;
using namespace std;

int main() {
    vector<int> v;
    v.resize(100);
    iota(v.begin(), v.end(), 1);    // Fill vector from 1 to 100
    vector<int> v2 = cons(0, v);
    vector<int> v3 = take(10, v2);
    vector<int> v4 = drop(90, v);
    auto p = splitAt(15, v2);
    printCont(v3);
    printCont(v4);
    printCont(get<0>(p));    // Print first vector in the tuple
    return 0;
}
```



```
C:\WINDOWS\system...
[ 0 1 2 3 4 5 6 7 8 9 ]
[ 91 92 93 94 95 96 97 98 99 100 ]
[ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 ]
Press any key to continue . . .
```

3.4 String manipulation functions

These specific string functions are related mainly to converting strings into vectors and back. They are *words*, *lines*, *unwords* and *unlines*. The first two functions both receive strings, and return a vector with separated strings; *words* breaks then by white space, and *lines* breaks then by the newline character. The *unwords* and *unlines* functions are their respective inverse operations, receiving vectors of strings and returning a single string, joined by white space and new line characters, respectively.

When combined with higher-order functions, these string manipulation ones become really useful. In Haskell, for example, one can read an entire file of numbers separated by new lines, for example, using only two lines of code and a one line function (this was actually used during the measurement testing in our project). The *readFile* function returns a single string. When we call *lines*, the data is passed onto a list of strings with only a number in each string. We then *map* this list to a function that simply reads a string and converts it to an int, applying it to all the elements in the vector.

We can actually reproduce this behavior creating a *readFile* function with similar effect, and then combine *map* and *lines* to generate a vector of ints in one line. As we learn more higher-order functions, we will be able to perform more complicated operations.

```
#include <vector>
#include <iostream>
#include <string>
#include <fstream>
#include "husky.h"

using namespace husky;
using namespace std;

string readFile(const char* filename) {
    ifstream input;
    input.open(filename);
    vector<string> v;
    string s;
    while (!input.eof()) {
        input >> s;
        v.push_back(s);
    }
    input.close();
    return unlines(v);
}

auto rInt = [](string s) { return stoi(s); };

int main() {
    string ints = readFile("ints.txt");
    vector<int> v1 = map(lines(ints), rInt);
    return 0;
}
```

```
rInt :: String -> Int
rInt = read

main = do
    ints <- readFile "../Input/ints.txt"
    let l1 = map rInt (lines ints)
    show $ l1
```


3.5 Zipping functions

Zippping containers together is an easy way to iterate over 2 or more of them at the same time. We have provided in our library several zippping functions, taking either 2 or 3 containers as arguments. There are also the higher-order version of these functions, which apply a binary or ternary function to the elements being accessed at the time. Our library implements the same zippping functions as Haskell prelude: *zip*, *zip3*, *unzip*, *unzip3*, *zipWith* and *zipWith3*.

In our implementation, we have used `std::pairs` for *zip* and `std::tuple` for *zip3*. These functions had all big limitations on implementation, because we needed to guarantee a specific structure for the data. We have then provided support for vectors only. The prototypes are all detailed in the corresponding manual.

When zippping vectors of different sizes, the function stops at the end of the smaller container. That means that when unzipping the result, one cannot retrieve the original bigger container. As such, it is indicated to check the containers' sizes if one wishes to access all their elements.

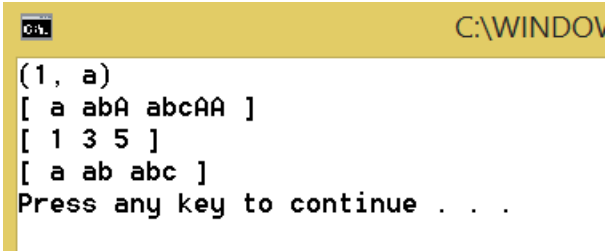
In the example below, we use all the pair versions of *zip* (the triple versions are similar, only with an extra container argument). We first *zip* a vector of ints with one of strings, and print the first resulting pair. Then we use a function that fills the strings with 'A's until the int parameter length is fulfilled, and call *zipWith* with the same initial vectors, printing the resulting one. Last we *unzip* the pair vector, recovering the smallest common subset of the original two.

```
#include <vector>
#include <iostream>
#include <string>
#include "husky.h"

using namespace husky;
using namespace std;

string fillWithAs(int x, const string& s) {
    string st{ s };
    while (st.length() < static_cast<unsigned>(x)) st.push_back('A');
    return st;
}

int main() {
    vector<int> v = { 1, 3, 5, 6 };
    vector<string> vs = { "a", "ab", "abc" };
    auto vp = zip(v, vs);
    std::cout << "(" << get<0>(vp[0]) << ", " << get<1>(vp[0]) << ")\n";
    auto vs2 = zipWith(fillWithAs, v, vs);
    printCont(vs2);
    auto p = unzip(vp);
    printCont(get<0>(p));
    printCont(get<1>(p));
    return 0;
}
```



```
C:\WINDOW
(1, a)
[ a abA abcAA ]
[ 1 3 5 ]
[ a ab abc ]
Press any key to continue . . .
```

4. Higher-order Functions

4.1 Mapping Functions

The mapping function basically applies a parameter function to the container received. The *concatMap* function is a composition of *concat1* and *map*, taking a function that generates containers, mapping it to the container and then concatenating the result. There is a version of *map* implemented in the STL, `std::transform`, but we could not call it in our own function, due to templates restrictions.

The *map* function is one of the most important higher-order ones, as it is widely used to manipulate containers through unary functions. In this tutorial we have already combined *map* with other functions to perform somewhat complex operations in just a few lines of code.

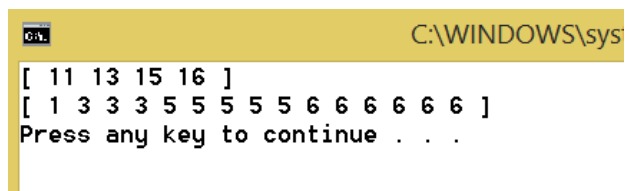
```
#include <vector>
#include <iostream>
#include <string>
#include <functional>
#include "husky.h"

using namespace husky;
using namespace std;

vector<int> fillUp(int x) {
    vector<int> v;
    while (v.size() < x) v.push_back(x);
    return v;
}

auto plus10 = [](int x) { return x + 10; };

int main() {
    vector<int> v = { 1, 3, 5, 6 };
    auto v2 = map(v, plus10);
    printCont(v2);
    auto v3 = concatMap(v, fillUp);
    printCont(v3);
    return 0;
}
```



```
C:\WINDOWS\system32\cmd.exe
[ 11 13 15 16 ]
[ 1 3 3 3 5 5 5 5 5 6 6 6 6 6 6 ]
Press any key to continue . . .
```

4.2 Filtering Functions

The filtering functions are similar to the mapping ones, but they receive predicates, i.e. functions that return a value that is bool-convertible. The classic *filter* function uses that predicate to generate a new container which only contains predicate-satisfying elements of the original container. The *takeWhile* and *dropWhile* functions use the predicate to determine when the prefix (or suffix) will end, acting afterwards like its take and drop versions. The remaining ones, *span_container* and *break_container* (so named due to reserved words in C++) are similar to *splitAt*, only that they use *takeWhile* and *dropWhile* instead, and *break_container* uses the opposite of the given predicate.

In the example below, we combine `takeWhile` and `filter` to obtain the vector containing all odd squares that are less than 1000. We also use `map` to apply the square function. We can do all this operations in one clear line of code using Husky.

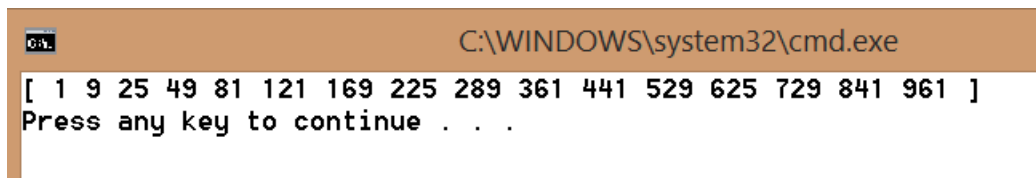
```
#include <vector>
#include <iostream>
#include "husky.h"

using namespace husky;
using namespace std;

vector<int> fillUp(int x) {
    vector<int> v;
    while (v.size() < x) v.push_back(x);
    return v;
}

auto lt1000 = [](int x) { return (x < 1000); };
auto square = [](int x) { return x*x; };
auto odd = [](int x) { return (x % 2); };

int main() {
    vector<int> v;
    v.resize(1000);
    iota(v.begin(), v.end(), 1);
    auto vodd = takeWhile(lt1000, filter(map(v, square), odd));
    printCont(vodd);
    return 0;
}
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd.exe'. The window contains the output of the program: a single line of numbers '[1 9 25 49 81 121 169 225 289 361 441 529 625 729 841 961]' followed by the prompt 'Press any key to continue . . .'. The numbers are the squares of odd integers from 1 to 31, which are less than 1000.

4.3 Folding functions

The folding functions usually take a function, a container and an optional accumulator, and then reduce the container to some single value, obtained from applying the function to the accumulator and the elements of the list. There are two main kinds of folds, the left ones and the right ones. The first kind runs from left to right while accessing the elements in the list, while the right version folds from right to left. They can generate different results for not symmetric functions, such as division, for example.

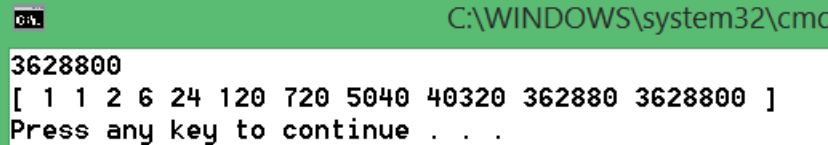
There are several special fold cases implemented in our library, such as `sum`, `product`, and `_fold`, etc. A detailed explanation of each one purpose is available in the manual pages. In this document, we will learn how to use `foldl` and `scanl` to obtain products. The main difference is that `scanl` (and its similar) returns a vector with the partial results of folding the container.

In the example, we use the `foldl` and `scanl` functions in a vector of ints, obtaining the factorial of the last element in the `foldl` application, and all the factorials of the previous elements using `scanl`.

```
#include <vector>
#include <iostream>
#include "husky.h"

using namespace husky;
using namespace std;

int main() {
    vector<int> v;
    v.resize(10);
    iota(v.begin(), v.end(), 1);
    auto fat10 = foldl(v, 1, multiplies<int>());
    cout << fat10 << endl;
    auto vfat = scanl(multiplies<int>(), 1, v);
    printCont(vfat);
    return 0;
}
```



```
C:\WINDOWS\system32\cmd
3628800
[ 1 1 2 6 24 120 720 5040 40320 362880 3628800 ]
Press any key to continue . . .
```

4.4 Function Composition

Function composition uses the mathematical similarly named concept to produce new functions. Our `compose` function receives two functions, `f` and `g` as parameters and returns a new function, `h`, such that $h(x) = f(g(x))$. The restriction imposed upon the functions is that the return type of `g` must be the same input type of `f`. This function also utilizes C++14 syntax, so it can only be run in appropriate compilers.

Function composition is particularly useful by making functions on the fly that can be passed to other functions, such as filters, maps or folds. It also results in clear and concise code. For example, we can take a vector of ints and return a vector with their numbers multiplied by 5 using function composition and `map`. To do this, we can model the old multiplication trick of multiplying by 10 and then dividing by 2, composing these two functions.

```
#include <vector>
#include <iostream>
#include "husky.h"

using namespace husky;
using namespace std;

auto mul10 = [](int x) { return x * 10; };
auto div2 = [](int x) { return x / 2; };

int main() {
    vector<int> v;
    v.resize(10);
    iota(v.begin(), v.end(), 1);
    auto v5 = map(v, compose(div2, mul10));
    printCont(v5);
    return 0;
}
```

```
Composetest.cpp Documentation Tests runTests.sh
DataCompilation README.md husky.h
Ayos-MacBook-Pro:Husky ayofapohunda$ clang++ -std=c++14 Composetest.cpp
Ayos-MacBook-Pro:Husky ayofapohunda$ ./a.out
[ 5 10 15 20 25 30 35 40 45 50 ]
Ayos-MacBook-Pro:Husky ayofapohunda$
```

5. Conclusion

With this tutorial, we intended to provide an overview of Husky's capabilities, and at the same time teach the user how to use the most important functions in our library. This document should also provide the reader with an idea of what is functional programming and how some of Haskell's most important functions work.

We have endeavored to follow as much as possible the STL paradigm for libraries, relying on already tried and tested auxiliary functions instead of trying to replace them with our own. Our intention is for Husky to be used alongside STL, as an easy and intuitive library that has satisfactory efficiency.

We have also provided as much documentation as possible, since one big difficulty that we had while working on this project was to find proper documentation. It has greatly slowed our testing with other libraries, as we often had to search through source code itself to find documentation.

We also intend to extend Husky in the future, and all the documentation will be updated to reflect it. The future versions of Husky should have support to other staple functional programming features, such as lazy implementation and currying.