# Learning FC++: The C++ functional programming library

## Use FC++ to implement basic functional programming

Arpan Sen

August 10, 2010

C++ is usually synonymous with object-oriented programming (OOP), and further replenished in no small measure by popular technical literature. This article tries something different —functional programming with C++ using the open source FC++ library from Yannis Smaragdakis and Brian McNamara. Learn how you can use FC++ to implement basic functional programming.

## Why try functional programming, specifically with FC++?

Some of the advantages that functional programming has over other programming paradigms, such as OOP, are:

- Conciseness of code
- Programming that's free of side effects (no global/static variables manipulated by endless set/get routines)
- Fast prototyping
- FC++ provides a wealth of syntax and library functions that help make the transition smooth for Haskell programmers.

You get around the fact that C++ does not have any functional programming constructs by using libraries. FC++ is the best available open source implementation of a C++ based functional programming library that you can plug in with legacy C++ code. FC++ has been used in projects such as BSFC++, which is a library for functional bulk synchronous parallel programming in C++.

## Download and installation

FC++ is available for download from SourceForge (see Resources). Unpacking the installed compressed (.zip) file reveals a collection of header files. Including the `prelude.h` header in the user application sources is all you need to do to get started. Listing 1 shows you how to compile sources that use FC++ code. Note that this is a header-dependent installation, and no other libraries are involved.

### Listing 1. Compiling sources that use FC++ code

```
g++ user_source1.cpp #I<path to FC++ installation>
```

Note: All code in this article was tested using FC++ 1.5 with g++ 3.4.4.

## Understanding CFunType

The functional programming paradigm lets functions accept other functions as arguments. Clearly, the base versions of C/C++ don't allow for such syntax. To circumvent this problem, FC++ functions are expressed as instances of classes that follow certain coding conventions, and this is where `CFunType` comes into play. C++ function objects are characterized by the presence of the `operator ( )` in the class definition. Listing 2 below is an example.

### Listing 2. Typical use of C++ function objects

```
struct square {
   int operator( ) (int x) { return x * x; }
};

square sqr1;
int result = sqr1(5);
```

The problem with the implementation in Listing 2 is that, in mathematical terms, the function type for `sqr1` is `int –> int`, but the C++ type for `sqr1` is `struct square`. FC++ introduces the template `CFunType`, which is used for encoding the type signature information. The last argument in `CFunType` is the return type of the function, and the rest are input type information in the same order they appear in the function prototype. Listing 3 shows how square looks using `CFunType`.

### Listing 3. Using `CFunType` to encode function signature for square operation

```
#include #prelude.h#

struct square : public CFunType<int, int> {
   int operator( ) (int x) { return x * x; }
};

square sqr1;
int result = sqr1(5);
```

Listing 4 is another example that inserts an integer into a list and returns the updated list.

### Listing 4. Using `CFunType` to encode function signature for list manipulation

```
#include #prelude.h#

struct Insert : public CFunType<int,List<int>,List<int> > {
   List<int> operator()( int x, const List<int>& l ) const {
    // code for determining where to insert the data goes here
}
```

Note: The `List` data type in Listing 4 is a predefined FC++ type described later in this article.

## Transforming functions into objects

For functions to accept functions as input arguments, the functions must be somehow transformed into objects. FC++ defines the `Fun`*`N`* category of classes that are built on top of `CFunType` and the `ptr_to_fun` routine, which actually carries out the transformation. Take a look at Listing 5.

### Listing 5. Using `ptr_to_fun` to convert function to FC++ function object

```
int multiply(int m, int n) { return m*n; }

Fun2<int, int, int> mult1 = ptr_to_fun (&multiply);
int result = mult1(8, 9);

// result equals 72
```

As in `CFunType`, signature for `Fun2` implies that this object represents a function that accepts two integer inputs and returns an integer. Likewise, you may have `Fun3<int, double, double, string>`, which represents a function that accepts one integer, two doubles and returns a string.

## List and laziness basics

List manipulation is at the heart of functional programming. FC++ defines its own list data type, which is different from Standard Template Library (STL) list. FC++ lists are *lazy*. You can create lists with infinite elements in FC++, but they are only evaluated on a need basis. Listing 6 demonstrates what this means.

### Listing 6. Defining and using lazy lists

```
List<int> numbers = enumFrom (33);
List<int> even_and_greater_than_33 = filter (even, numbers);
assert (take(4, even_and_greater_than_33)) = list_with (34, 36, 38, 40);
```

The `enumFrom`, `filter`, `even`, `take`, and `list_with` elements are part of predefined functionality in FC++. In Listing 6 above, `enumFrom` returns an infinite list of numbers starting from 33. The `filter` routine returns another infinite list with numbers that are even and greater than 33. Finally, the `take` routine actually extracts the first four elements from this list. Clearly, none of the lists store an infinite list of numbers—the evaluation is strictly on a need basis.

Table 1 describes some of the typical functions used with lists in FC++.

### Table 1. Functions that are used in conjunction with the FC++

| Function | Description |
|---|---|
| `head(<list>)` | Returns the first element of the list |
| `tail(<list>)` | Returns a list with the same elements as `<list>` other than the first |
| `cons(<element >, <list>)` | Returns a list with `<element>` added to the head of the list |
| `NIL` | Signifies an empty list |
| `list_with(<element1, element2>, , <element`*`N`*`>)` | Creates a list with *N* elements |

| | |
|---|---|
| `enumFrom(<element1>)` | Creates an infinite list starting with `element1` |
| `compose(<func1>, <func2>)` | Compose (f, g) = f(g(x)), where f(x) and g(x) are two functions |
| `filter(<func1>, <list>)` | Returns a list of elements from `<list>` filtered using the `<func1>` function |
| `take(<N>, <list>)` | Returns a list with the first *N* elements from `<list>` |
| `map(<function>, <list>)` | Applies the first `<function>` function to each element of first `<list>` |

Listing 7 is another example that shows how to create and display the contents of a list.

### Listing 7. Creating a list, checking its contents, and displaying data

```
#include <iostream>
#include #prelude.h#

int main( )
{
int x=1, y=2, z=3;
List<int> li = cons(x,cons(y,cons(z,NIL)));

// head also removes the 1st element from the list
assert( head(li) == 1 );

// tail returns whatever is left of in the list, and list_with is
// used to define small sized list
assert( tail(li) == list_with(2,3) );

 while( li ) {
      std::cout << li.head() << " ";
      li = li.tail();
 }
return 0;
}
```

Note: In the creation of the `li` list, the `cons` routine adds elements to the front of a list; z, y and x are added in that order to create the final list.

## Faster list implementation

FC++ 1.5 provides an additional variant of the `List` data structure called `OddList`, which is defined in list.h. `OddList`s have exactly the same interface as `List`s, but they are faster. All FC++ routines that operate on `List` operate on `OddList`, too. The efficiency in `OddList` is gained by caching the next node in the list. Listing 8 sums up some of the subtler aspects of using `OddList`.

### Listing 8. Subtler aspects of using `OddList`

```
OddList<int> odd1 = enumFrom (1);
List<int> list1 = odd1.tail ( ); // always returns List<int>!!

OddList<int> odd2 = enumFrom (1);
List<int> list2 = odd2.delay ( ); // create a List<int> with same data as odd2

List<int> list3 = enumFrom (1);
OddList<int> odd3 = list3.force ( ); // creates an OddList<int> with same data as list3
```

`OddList`s don't have support for STL style iterators that exist for `Lists`. See Resources for further detail on `OddList` implementation.

## Creating your own filters

If you want to create your own filter in Listing 6 (for instance all numbers that are divisible by 100 and greater than 33), all you need to do is define your own filter function and then call `ptr_to_fun` to convert it into a function object. Listing 9 shows you how.

### Listing 9. Using `CFunType` to encode function signature for list manipulation

```
bool div_by_100 (int n) {
  return n % 100 ? false : true;
}

List<int> num = enumFrom(34);
List<int> my_nums = filter( ptr_to_fun(&div_by_100), num);
```

Note that FC++ `List`s and `filter`s are completely generic in nature and can accommodate any data type.

Next, look into two fundamental functional techniques: currying and composition.

## Currying

*Currying* is a functional programming technique that binds a subset of some function's arguments to fixed values, thus creating new functions. Listing 10 is an example that curries the `f` function.

### Listing 10. Using currying to create new functions

```
int multiply(int m, int n) { return m * n; }
Fun2<int, int, int> f2 = ptr_to_fun (&multiply);
Fun1<int, int> f1 = curry2 (f2, 9);

std::cout << f1(4) << std::endl; // equivalent to multiply(9, 4)

Fun1<int, int> f1_implicit = f2(9);
std::cout << f1_implicit(4) << std::endl; // same as f1(4)
```

The predefined `curry2` routine binds the first argument of `f2` to `9`. FC++ 1.5 provides `curry1`, `curry2`, and `curry3` operators that fix the first *`N`* arguments to specific values. Additionally, FC++ also defines the bind routines to create new functions that prefix values to specific arguments of existing functions. For example, `bind2and3of3 (f, 8, 9)` is equivalent to `f(x, 8, 9)` where f(x, y, z) is a 3-input function. Yet another interesting way of specializing arguments is to use an underscore (`_`). For instance, `greater (_, 10)` is the same as f(x) = (x > 10). Note that greater is predefined in FC++. Listing 11 provides some more examples of currying.

### Listing 11. More currying examples

```
List<int> integers = enumFrom (1);
List<int> int_gt_100 = filter(greater(_, 100), integers);

// This list will add 3 to all elements of integers.
List<int> plus_3 = map (plus(3), integers);
```

Listing 12 shows a code snippet that displays all the factors of a number, including the number itself.

### Listing 12. Displaying all the factors of a number

```
#include "prelude.h"
using namespace fcpp;

#include <iostream>
using namespace std;

bool divisible( int x, int y ) { return x%y==0; }

struct Factors : public CFunType<int,OddList<int> > {
    OddList<int> operator()( int x ) const {
        return filter( curry2(ptr_to_fun(&divisible),x), enumFromTo(1,x) );
    }
} factors;

int main()
{
  OddList<int> odd = factors(20);
  while (odd) {
      cout << head(odd) << endl;
      odd = tail(odd);
  }
  return 0;
}
```

The key to understanding Listing 12 lies in this snippet: `return filter( curry2(divisible,x), enumFromTo(1,x) );`. You are creating a filter for the list returned by `enumFrom(1, 20)` such that all numbers that perfectly divide 20 form a part of the final list. The `curry2` routine binds 20 to the first argument of the `divisible` function. Note that `ptr_to_fun` makes `divisible` a function object that can be passed as an argument to `curry2`.

## Composition

Functional programming produces new functionality by combining existing code. The `compose ( )` operator composes two unary functions, `f(x)` and `g(x)`, to yield a new function, `h(x)`, such that h(x) = f(g(x)). For example, `compose (head, tail)` on a list returns the second element in the list. This is functional coding in its proper sense; `g(x)` serves as the argument to `f(x)`. Listing 13, obtained from "Functional Programming with the FC++ Library" (see Resources), is an example that uses composition.

### Listing 13. Using `compose` and `tail` to obtain the second element of a list

```
std::string s=#foo#, t=#bar#, u=#qux#;
List<std::string> ls = cons(s, cons(t, cons(u, NIL)));

ls = compose(tail, tail) (ls); // tail(tail(ls));
assert (head(ls) == #qux#); // s, t are removed
```

Listing 14 is another example that increments all elements of a list by two.

### Listing 14. Using `compose` to increment list elements

```
List<int> integers = enumFrom (1);
map (compose(inc, inc), integers);
// this modifies integers to an infinite list [3, 4, 5 ...]
```

## Lambda functions

Any discussion about functional programming is incomplete without a mention of lambda functions. Lambda abstraction is used for defining anonymous functions. This is useful when you don't want to define separate functions for small pieces of code. To use lambda functionality in code, you need to define the `FCPP_ENABLE_LAMBDA` macro. Listing 15 succinctly defines new mathematical and logical functions from existing code. Notice how `factorial` is defined.

### Listing 15. Defining lambda functions

```
// a new function where f(x) = 3*x+1
lambda(X)[ plus[multiplies[3,X],1] ]

// a new function where f(x) = x! (factorial x)
lambda(X)[ l_if[equal[X,0],1,multiplies[X,SELF[minus[X,1]]]] ]
```

The code in Listing 15 is self-explanatory. Routines `plus`, `multiplies`, and so on are defined as part of the FC++ library, and you use the `lambda` operator to create new functionality from existing code.

## Conclusion

FC++ provides:

- Objects of type `CFunType`, which you can easily extend to serve functional programming needs
- Implementation of lazy lists that can potentially hold infinite sequences
- Several functional programming operators such as `head`, `tail`, `map`, `filter`, `ptr_to_fun`, and so on
- The ability to create new functions from existing functions using currying operators, `lambda`, or `compose`

Probably the singular drawback of FC++ is the lack of standardized documentation that describes the functions defined in its headers. This article introduced the most useful ones: `compose`, `curry`, `bind`, `take`, `map`, `ptr_to_fun`, and `filter`.