Lab 12: Iterators

In previous labs, we introduced "Object-Oriented C" and how to use exceptions in C. In this lab, we'll introduce the concept of an "iterator". A iterator is a form of function that can return values in a sequence, one at a time. In principle, the sequence can even be infinite (e.g. all prime numbers or all rational numbers).

The concept of iterators comes from *functional programming* (FP), a way to program that offers many advantages over procedural programming, which is what we've been doing in class so far. One such feature of FP is that FP code is intrinsically thread-safe.

This page will be available as

> http://www.tricity.wsu.edu/~bobl/cpts360/lab12_iterators/writeup.html

Calling it up in a browser will allow you to cut-and-paste code in the following and save typing. You may also download whole files from the lab link on the course web page.

1. The `factorial` Iterator

A properly-implemented iterator is an example of "lazy" evaluation. All elements are not computed at once. Instead, each element is computed "on-the-fly" when requested. This is especially useful when the sequence is infinite, such as the set of prime numbers, or expensive to compute.

The best way to understand how iterators work is to build one. We'll do that from scratch in the second part of this lab, but in the first part we'll study a simple one: an iterator for factorials.

Recall that `n!` ("n factorial") is the product of all positive integers less than or equal to `n`, so the factorial sequence starts off with:

```
1, 2, 6, 24, 120, 720, ...
```

Let's examine a factorial iterator. Even though factorials are often implemented using recursion, we're not doing that here.

The `FactorialState` Struct

The most important thing in designing an iterator is to start with a structure that contains the iterator's "state": all of the data needed when one iteration completes to compute the next one. Iterators never use static or global storage.

Here's the header file `factorial.h`, which implements the iterator state as a C `struct`:

```
struct FactorialState {
// contains all state of the Factorial iterator.
    unsigned int n;     // previous value of n
    unsigned int nFact; // current value of n!
};

extern void factorial_init(struct FactorialState *factorialState);
extern int factorial_next(struct FactorialState *factorialState,
                          unsigned int *nFact);
```

This header also includes `extern` declarations for two functions. (You can think of them as "methods" in an object-oriented sense.) In keeping with object-oriented conventions, they always take a (pointer to the) state `struct` as a first argument.

The first function, `fibonacci_init()`, is passed a `FibonacciState struct` (pointer) to initialize. It may also take any parameters needed to initialize that `struct`, but in this case there are none.

Once the `struct` is initialized, you pass it to the second ("next") function `fibonacci_next()` to get every successive member of the sequence. When the sequence is not complete, the "next" function returns 1. When the sequence completes (if it completes), it returns 0. (See the test program below for how this works.)

The `Factorial` Implementation

The implementations of these functions are in `factorial.c`:

```
#include <limits.h>
#include <stdio.h>

#include "factorial.h"


void factorial_init(struct FactorialState *factorialState)
// initialize factorial state
{
    factorialState->n = 1;

    // this will be the first value returned by the iterator
    factorialState->nFact = 1;
}


int factorial_next(struct FactorialState *factorialState, unsigned int *result)
// the "next" method for factorial iterator
{
    if (factorialState->nFact == 0) // can't compute the next result
        return 0;
    (*result) = factorialState->nFact;

    // compute the next step (or prepare to terminate)
    factorialState->n++;

    //
    // In principle, (n+1)! = n! * (n+1), but for this to work with
    // finite precision (32- or 64-bit) ints, we must have
    //
    //     (n+1)! = n! * (n+1) <= UINT_MAX
    //
    // which implies
    //
    //     n! <= UINT_MAX / (n+1)
    //
```

```
    // This predicts, but avoids, overflow. If n! can't be
    // represented, the iterator exits.
    //
    if (factorialState->nFact > UINT_MAX / factorialState->n)
        factorialState->nFact = 0; // iterator will stop on the next call
    else
        factorialState->nFact *= factorialState->n;

    return 1;
}
```

`factorial_init()` is easy enough. There's a bit of a complication in `factorial_next()`: Even though the factorial sequence is infinite, the fact that the results have to be represented in 32-bit unsigned integers limits the sequence. (See the comments for details.)

Test Program

Here is a test program `factorial_t.c` for the factorial iterator:

```
#include <stdio.h>

#include "factorial.h"


int main(int argc, char *argv[])
{
    struct FactorialState factorialState;
    unsigned int nFact;

    factorial_init(&factorialState);
    while (factorial_next(&factorialState, &nFact)) {
        printf("%u\n", nFact);
    }
}
```

Notice how simple this makes using the iterator: Call the "init" function once and the iterator (the "next" function) as many times as needed. If the iterator returns zero, there are no more elements in the sequence. It's even possible to create iterators for infinite sequences that never return 0, such as one for *n*-sided die rolls where *n* would be an argument to the "init" function.

Download a copy of the test program from the lab link on the course web page. You will not need to modify this file.

Assignment

1. Download and modify `factorial.c` and `factorial.h` to work with (64-bit) unsigned `long` s. This will allow it to generate more values. (This should not take long. The idea is to get you used to the idea of working with iterators.)

2. Create a makefile that compiles `factorial.c` into `factorial.o` and then links `factorial.o` and `factorial_t.o` into `factorial`.

2. The `string_range` Iterator

In this part of the lab, you will create an iterator for a "string range": a string that specifies one or more integer ranges separated by commas. Each integer range consists of two positive integers separated by a dash.

You might, for instance, use this to indicate (on a command line, perhaps) a range of pages to be printed from a large document. If you wanted to print pages 4 through 7, you would specify a string range "4-7" with `stringRange_init()`. Successive calls to `stringRange_next()` would return 4, 5, 6, and 7 via a pointer argument and 1 as a `return` value. Any additional calls would return a 0 as a `return` value (and not modify the state).

It's also possible to have an arbitrary number of additional ranges, separated by commas, so passing "12-14,20-22" to `stringRange_init()` would make `stringRange_next()` return 12, 13, 14, 20, 21, and 22.

Here's a partially filled-in, downloadable `string_range.h`:

```
struct StringRangeState {
    // ASSIGNMENT: insert state data here
};

extern void stringRange_init(struct StringRangeState *stringRangeState,
                             char *ranges);
extern int stringRange_next(struct StringRangeState *stringRangeState,
                            int *result);
```

Assignment

1. Download the above `string_range.h`.
2. In the header file, design and code the `StringRangeState` struct.
3. Implement the `StringRange` iterator in a file named `string_range.c`. You may want to use `factorial.c` as a template.
4. Write your own test program (in a separate file) to test the iterator.

Submission

Put the `factorial.c`, `factorial.h`, `string_range.h`, `string_range.c`, a makefile that compiles `factorial.o` and `string_range.o`, and a `README.txt` (if you choose to have one) in a submission tarball and submit it via Canvas.

The instructor will use your implementations of `factorial.c`, `factorial.h`, `string_range.h`, and `string_range.c`, but will use a separate test file.