Eric Roberts
CS 106B

Handout #55
March 13, 2013

Functions as Data

Functions as Data

Eric Roberts
CS 106B
March 13, 2013

Iteration Strategies

- Chapter 20 of the text covers two strategies for iterating over collections: ***iterators*** and ***mapping functions***.
- Dawson showed you how to use iterators in his lecture last Friday. My goal for today is to discuss mapping functions and, more generally, the entire idea of using functions as data.

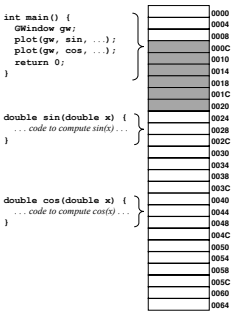
The von Neumann Architecture

- One of the foundational ideas of modern computing—traditionally attributed to John von Neumann although others can make valid claims to the idea—is that code is stored in the same memory as data. This concept is called the ***stored programming model***.
- If you go on to take CS 107, you will learn more about how code is represented inside the computer. For now, the important idea is that the code for every C++ function is stored somewhere in memory and therefore has an address.

John von Neumann and J. Robert Oppenheimer

Callback Functions

- The ability to determine the address of a function makes it possible to pass functions as parameters to other functions.
- In this example, the function **main** makes two calls to **plot**.
- The first call passes the address of **sin**, which is **0028**.
- The second passes the address of **cos**, which is **0044**.
- The **plot** function can call this function supplied by the caller. Such functions are known as ***callback functions***.



Function Pointers in C++

- One of the hardest aspects of function pointers in C++ is writing the type for the function used in its declaration.
- The syntax for declaring function pointers is consistent with the syntax for other pointer declarations, although it takes some getting used to. Consider the following declarations:

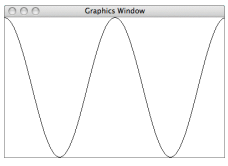
`double x;`	*Declares* **x** *as a* **double**.
`double *px;`	*Declares* **px** *as a pointer to a* **double**.
`double f();`	*Declares* **f** *as a function returning a* **double**.
`double *g();`	*Declares* **g** *as a function returning a pointer to a* **double**.
`double (*proc)();`	*Declares* **proc** *as a pointer to a procedure returning a* **double**.
`double (*fn)(double);`	*Declares* **fn** *as a pointer to a function taking and returning a* **double**.

Plotting a Function

- Section 20.2 defines a **plot** function that draws the graph of a function on the graphics window.
- The arguments to **plot** are the graphics window, the function, and the limits in the *x* and *y* directions. For example, calling

```
plot(gw, cos, -2 * PI, 2 * PI, -1.0, 1.0);
```

produces the following output:



Exercise: Defining the `plot` Function

1. What is the prototype for the `plot` function?

2. How would you convert values `x` and `y` in the mathematical domain into screen points `sx` and `sy`?

   Hint: In the time that `x` moves from `minX` to `maxX`, `sx` must move from 0 to `gw.getWidth()`; `y` must move in the opposite direction from `gw.getHeight()` to 0.

Exercise: Dispatch Strategies

- In your implementation of BASIC, you presumably have a function that looks something like this:

```
Statement *parseStatement(TokenScanner & scanner) {
   string token = toUpperCase(scanner.nextToken());
   if (token == "REM") return new RemStmt(scanner);
   if (token == "LET") return new LetStmt(scanner);
   if (token == "PRINT") return new PrintStmt(scanner);
   if (token == "INPUT") return new InputStmt(scanner);
   if (token == "GOTO") return new GotoStmt(scanner);
   if (token == "IF") return new IfStmt(scanner);
   if (token == "END") return new EndStmt(scanner);
   error("Unrecognized statement: " + token);
}
```

- This strategy would become problematic if there were more statement types or if those types could expand dynamically. How might you restructure this implementation so that the size of the code did not depend on the number of statements?

Mapping Functions

- The ability to work with pointers to functions offers one solution to the problem of iterating through the elements of a collection. To use this approach, the collection must export a ***mapping function*** that applies a client-specified function to every element of the collection.
- Most collections in the Stanford libraries export the method

```
template <typename ValueType>
void mapAll(void (*fn)(ValueType));
```

  that calls `fn` on every element of the collection.
- As an example, you can print the elements of a `Set<int> s`, by calling `s.mapAll(printInt)` where `printInt` is

```
void printInt(int n) {
    cout << n << endl;
}
```

Exercise: Implement `mapAll`

Implement the function

```
void mapAll(void (*fn)(string));
```

as part of the `StringMap` class, for which the private section looks like this:

```
/* Type definition for cells in the bucket chain */

   struct Cell {
      std::string key;
      std::string value;
      Cell *link;
   };

/* Constant definitions */

   static const int INITIAL_BUCKET_COUNT = 13;

/* Instance variables */

   Cell **buckets;        /* Dynamic array of pointers to cells */
   int nBuckets;          /* The number of buckets in the array */
```

Passing Data to Mapping Functions

- The biggest problem with using mapping functions is that it is difficult to pass client information from the client back to the callback function. The C++ packages that support callback functions typically support two different strategies for achieving this goal:

  1. Passing an additional argument to the mapping function, which is then included in the set of arguments to the callback function.
  2. Passing a function object to the mapping function. A ***function object*** is simply any object that overloads the function-call operator, which is designated in C++ as `operator()`.

- For the last bit of today's class, I'll go through each of these strategies in the context of a program that writes out every word in the English lexicon that has a particular length.