CS 237 Meeting 18 — 10/22/12

Announcements

1. Midterm: New date: Oct 29th. In class open book/notes.

2. Duane Bailey has volunteered to do a chips and breadboards lab this week. Will any of you volunteer to do lab at 4?

Structures in C

1. Programming in an object-oriented language like Java depends heavily on the definition of classes that collect a group of variables and functions (method) that manipulate them.

2. C provides a lower-level facility that allows the programmer to describe a collection of related variables called a structure. Using structures, one can simulate much of the functionality of a class by defining appropriate functions.

3. In C, the notation

```
struct point {
    double x;
    double y;
}
```

   describes a new type of values composed of pairs of values of the double type. This new type is called a struct or structure. Unlike the Java class, a structure is just a collection of variables. There are no functions.

4. We can declare a variable `somePoint` as a structure by saying:

```
struct point {
    double x;
    double y;
} somePoint;
```

5. Once we have describe a struct we can use the keyword struct followed by the name we used to define other variables:

```
struct point someOtherPoint;
```

6. Give a variable of a struct type, we can access its components using periods and component names. For example, we could print the coordinates of a point using the statement

```
printf( "( %f, %f )\n" , somePoint.x, somePoint.y );
```

7. We can also pass struct values as parameter and return them as the results of a function. Here are three examples of functions (two of which mimic methods from our Java Point class) that take point parameters:

```
void move( struct point p, double xDiff, double yDiff ) {
  p.x = p.x + xDiff;
  p.y = p.y + yDiff;
}

double distanceTo( struct point p, struct point otherPoint ) {
  double xDiff = p.x - otherPoint.x;
  double yDiff = p.y - otherPoint.y;
  return sqrt( xDiff*xDiff + yDiff*yDiff );
}

void printPoint( struct point p ) {
  printf( "( %f, %f )\n" , p.x, p.y );
}
```

typedefs

1. Structure definitions are often used together with another C mechanism, typedefs, that allows the programmer to define new type names.

2. In general, a typedef looks like a variable declaration preceded by the word typedef. Its effect is to associate the name that would have been a variable with the type with which that variable would be associated if typedef had been omitted. For example,

```
typedef int numberList[10];
```

   define numberList to be a name for the type of arrays of 10 integers. After this declaration, saying

```
numberList scores;
```

would define scores as an array.

3. As mentioned above, it is common to use a typedef to associate a name with a structure that can be used without first saying struct. For example, if we said:

```
typedef
    struct point {
           double x;
           double y;
        } point;
```

we could then define variables as

```
point somePoint;

point someOtherPoint;
```

and also use the new name in parameter declarations as in:

```
void move( point p, double xDiff, double yDiff ) {
  p.x = p.x + xDiff;
  p.y = p.y + yDiff;
}

double distanceTo( point p, point otherPoint ) {
  double xDiff = p.x - otherPoint.x;
  double yDiff = p.y - otherPoint.y;
  return sqrt( xDiff*xDiff + yDiff*yDiff );
}
```

Structs and pointers

1. If we actually try using the move function defined above, we will discover it does not work. The problem is that when we pass a variable as a parameter, we pass the value of the variable rather than the variable itself. Therefore, if we say

```
move( somePoint, 4, 4 );
```

the move function modifies a copy of the value of somePoint, rather than somePoint itself.

2. This is the same problem we saw long ago in the first version of swap I presented. We learned that we could fix this problem by passing a pointer to a variable rather than the variable's value. The same trick works here.

   - First we redefine move:

```
void move( point *p, double xDiff, double yDiff ) {
  (*p).x = (*p).x + xDiff;
  (*p).y = (*p).y + yDiff;
}
```

   - Then we pass a pointer when we invoke move:

```
move( &somePoint, 4, 4);
```

   - Not only does this make move work, it makes using structs more efficient since copying the value of a large struct passed as a parameter is more time consuming than passing a 4 byte pointer.
   - It is, however, painful to have to type (*p). everyplace. Luckily, C provides the shorthand notation p-> as an abbreviation for (*p). so we can rewrite move as

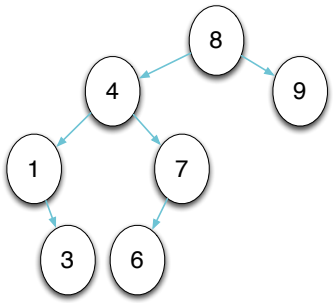
```
void move( point *p, double xDiff, double yDiff ) {
  p->x = p->x + xDiff;
  p->y = p->y + yDiff;
}
```

Structs, Pointers, and Recursive Structures

1. Together, structs and pointers make it possible to define recursive structures in C just as you can in Java.

2. As a motivating example, let's consider sorting (again)!

3. One approach we can use to sort a list of numbers is to organize the numbers into a binary tree so that all the numbers in the left subtree of a node are less than the number in the node and all the numbers in the right subtree of a node are greater than (or equal to) the number in the node.



Then, we can extract the numbers in order by completing a pre-order traversal of the tree.

4. If we wanted to write a Java program to perform such a sort, we might first define a recursive tree class:

```
public class IntTree {

    private int value;
    private IntTree left;
    private IntTree right;

    public IntTree( int value ) {
        this.value = value;
    }

    public void preOrder( ) {
        if( left != null ) {
            left.preOrder();
        }<
        System.out.print( value + " " );
        if ( right != null ) {
            right.preOrder();
        }
    }

    public void insert( int value ) {
        if ( value < this.value ) {
            if ( left == null ) {
                left = new IntTree( value );
            } else {
                left.insert( value );
            }
        } else {
            if ( right == null ) {
                right = new IntTree( value );
            } else {
                right.insert( value );
            }
        }
    }
}
```

along with a main method to drive the process:

```
import java.util.*;

public class Sort {

    public static void main( String argv[] ) {
        Scanner input = new Scanner( System.in );

        int value = input.nextInt();
        IntTree tree = new IntTree( value );
        while ( input.hasNextInt() ) {
            tree.insert( input.nextInt() );
        }
        tree.preOrder();
    }
}
```

5. This code illustrated two features of Java that are not quite matched in C.

   - IntTree is a directly recessive class. It has instance variables whose

type is just the type defined by the class.

- main and IntTree both use the new operation to allocate space for new objects.

6. If you try creating a directly recursive structure in C the compiler will not be happy. A declaration like:

```
typedef struct intTree {
  int value;
  struct intTree left;
  struct intTree right;
} intTree;
```

describes a structure containing two substructures each of which contains two structures each of which contains two substructures each of which contains two substructures each of which contains two substructures ...

7. The problem is that there is no way to terminate the recursion. In Java, any variable that refers to an object can have null as its value. In C, a structure has to have its components and if those components are themselves structures then they have to have their components too.

8. In C, however, pointer variables can be null. We have already seen that it is often useful to access structures through pointers. If we want to build a recursive structure type this become more than a convenience. It is essential.

9. For example, to define the intTree structure we would say:

```
typedef struct intTree {
  int value;
  struct intTree *left;
  struct intTree *right;
} *intTree;
```

That is, we will have each structure that represents a tree hold two (possibly null) pointers to its subtrees.

10. Note that we also define intTree as a pointer type. Most of the time, it will be the pointer type that we need to refer to rather than the structure type. When and if we do need to refer to the structure type we can say "struct intTree".

11. As a quick example of how we can use this type, consider the following definition of a preorder traversal function:

```
void preOrder( intTree root ) {
  if ( root != NULL ) {
    preOrder( root->left );
    printf("%d ", root->value );
    preOrder( root->right );
  }
}
```

Note that unlike the Java preorder method, this function can be called on a NULL tree and survive. This makes the function a bit more concise since we don't have to check whether left or right are null before making recursive calls.

- This is why we taught you not to use null to represent empty lists in 134.

Separate Compilation of C Programs

1. In the Java version of tree sort shown above, all the definitions of the tree class were collected in one file while the main method resided in its own class in a separate file. This is a good thing since the details of the implementation of a type like IntTree should be separated from the other details of the program that uses the type.

2. Let us see how we can arrange for the same separation in C.

- The trick is to somehow let the code in the .c file that contains main know enough about the tree type to use it even though the details of the tree type are specified in a separate file that will not be read by the compiler while it is processing the file containing main.

- The C solution is to place the details of the interface of a type like intTree in its own source file bearing a name ending in .h (for header). Then, both the file that includes main and the file that contains the details of the types implementation can access the description of the interface by using a `#include` to tell the compiler to read the file.
- That is, we will first create a file named `intTree.h` containing the code

```
typedef struct intTree {
  int value;
  struct intTree *left;
  struct intTree *right;
} * intTree;

// Insert a node for the number value in the
// appropriate place in a tree
void insert( intTree *rootPtr, int value );

// Visit and print the values of all nodes in root
void preOrder( intTree root );
```

- The typedef in this file describes the struct type used to represent tree nodes. The function headers provide enough information to use the functions associated with the type in the definition of main even though it might be in a different file from the function definitions themselves.
- In the file containing main, we include a `#include` for this .h file.

```
#include <stdio.h>
#include "intTree.h"

intTree root = NULL;

// Sort an input file full of integers
int main( int argc, char *argv[] ) {

  // Put all the numbers from the file into a tree
  int newValue;
  while( scanf( "%d", &newValue ) == 1 ) {
    insert( &root, newValue );
  }

  // Traverse the tree while printing its contents
  preOrder( root );

  // Free all the tree nodes to clean up nicely
  freeTree( root );
}
```

- Similarly, we include the .h file in the .c file that defines insert and preOrder:

```
#include "intTree.h"
#include <stdlib.h>
#include <stdio.h>


// Visit and print the values of all of the nodes in root
void preOrder( intTree root ) {
  if ( root != NULL ) {
    preOrder( root->left );
    printf("%d ", root->value );
    preOrder( root->right );
  }
}

// Make a new, one-node tree containing a give value
intTree newTree( int value ) {
  intTree result =
        (intTree) malloc( sizeof( struct intTree ) );

  result->value = value;
  result->left = result->right = NULL;
  return result;
```

```
}

// Insert a node for the number value in the appropriate place
// in a tree
void insert( intTree *rootPtr, int value ) {
  if ( *rootPtr == NULL ) {
    *rootPtr = newTree( value );
  } else if ( value <= (*rootPtr)->value ) {
    insert( &(*rootPtr)->left, value );
  } else {
    insert( &(*rootPtr)->right, value );
  }
}

// Free up the memory allocated for a tree
void freeTree( intTree root ) {
  if ( root != NULL ) {
    freeTree( root->right );
    freeTree( root->left );
    free( root );
  }
}
```

Passing Pointers by Reference

1. The first example of using pointers we really looked at was the swap function where pointers were use to pass an int variable by reference rather than by value.

2. The insert function in our tree sort code is an example of a C function that takes a pointer variable as a reference parameter.

   - If the method is passed a pointer variable that is NULL, it will update the variable to point to a newly created tree.

3. As in our swap example, we only need to do three things to pass a parameter by reference:

   - Add a star to the declaration of the parameter to indicate that it will hold a pointer to something of the parameter type rather than a value of the parameter type.
   - Throughout the body of the function, use "*x" instead of "x" to refer to the parameter (assuming its name is x).
   - When invoking the function, put an ampersand in front of the variable begin passed.

4. When pointers are passed by reference, however, things at least look a little more complicated syntactically.

   - If you were going to say "p->c" and try changing it to "*p->c" you will discover that the precedence rules are wrong (C tries to evaluate the arrow before the star) so you have to write "(*p)->c".