CSC161 2010F *Imperative Problem Solving*

Assignment 4: Fun with Bits and Beyond

Assigned: Friday, 17 September 2010
Due: 11:00 p.m., Wednesday, 22 September 2010

Summary: In this assignment, you will explore the underlying representation of some types and the consequences of C's type systems. You will also build a simple calculator.

Purposes: To give you more experience with these key concepts in C.

Expected Time: Two to three hours.

Collaboration: I encourage you to work in groups of two or three students. However, you may work on your own or in groups of up to size four. You may discuss the assignment with anyone you wish, provided you clearly document such discussions.

Submitting: Email me your answers. See further details below.

Warning: So that this assignment is a learning experience for everyone, I may spend class time publicly critiquing your work.

Assignment

Part 1: Examining Bits

This part of the assignment was done in class!

Write a function, `void show_int_bits (int i)`, the prints out the bits in integer `i`. For example, given 5 as input, your program should output something like

```
00000000000000000000000000000101
```

or

```
00000000 00000000 00000000 00000101
```

or

```
0000 0000 0000 0000 0000 0000 0000 0101
```

Part 2: Reverse Polish Notation

As you may know, Reverse Polish Notation (RPN) is a standard notation for calculation that is completely unambiguous. RPN is a postfix notation. That is, in RPN, the operator *follows* the operands. For example, to add 2 and 3 and multiply the result by 7, we would write

```
2 3 + 7 *
```

The standard implementation of RPN is fairly simple and uses a stack. When the next input is a number, it gets pussed on the stack. When the next input is an operation, the parameters get popped from the top of the stack, the operation gets applied, and the result gets pushed back on the top of the stack.

Write a program, `rpn`, that does RPN calculation, taking the input from the command line. For example,

```
% rpn 5
5
% rpn 5 2 +
7
% rpn 5 2 + 8 '*'
56
% rpn 8 5 2 + '*'
56
```

You should support at least the binary operations `+`, `-`, `*`, and `/`, and the unary operation `negate`.

You may assume that your stack need hold no more than sixteen values.

Some Hints

As you may know from your previous programming experience, one simple implementation of a stack is an array with an index that keeps track of the "top" of the stack. We'll call that index `top`. When you put something on the stack, you store it at `stack[top]` and then increment `top`. When you pop something from the stack, you grab the value at `stack[top-1]` and decrement `top`.

As you've learned recently, it's fairly easy to determine if a string has a particular value. The `strcmp` procedure returns 0 when given identical strings. For example, we might check whether `argv[i]` is the plus sign with

```
  if (strcmp (argv[i], "+") == 0)
    ...
```

Extra Credit

a. Turn your Reverse Polish Notation program into a procedure, `int rpn (int count, char *vec[])`, that implements a reverse polish notation calculator.

You can assume that any string in the array that is not one of the basic operations is an encoded integer that you can decode with `atoi`.

b. Write a series of examples that let you explore the behavior of your calculator. For example,

```
void
dump (int count, char *vec[])
{
  int i;
  if (count == 0)
    return;
```

```
  printf ("%s", vec[0]);
  for (i = 1; i < count; i++)
    printf (" %s", vec[i]);
} // dump

int
main ()
{
   char *exp0[] = { "5" };
   char *exp1[] = { "5", "3", "+" };

   dump (1, exp0);
   printf (" => %d\n", rpn(1, exp0));
   dump (3, exp1);
   printf (" => %d\n", rpn(3, exp0));

   return EXIT_SUCCESS;
} // main
```

c. Rewrite your tests to check each result and only report an error if the result is not what you would expect. For example,

```
  if (rpn (1, exp0) != 5)
    {
      dump (1, exp0);
      printf (": expected (5), got (%d)\n", rpn 1, exp0);
      ++errors
    }
  ...
  if (errors > 0)
    {
      printf ("\n%d errors encountered.\n", errors);
    }
```

Submitting Your Homework

Using `script`, build logs of sample runs of your programs.

Put those logs, your source code, and any other files you deem appropriate in a directory called `username.hw4`.

Make a tarball of that directory.

Submit that tarball as an attachment to an email message entitled should be titled CSC161 Assignment 4.
