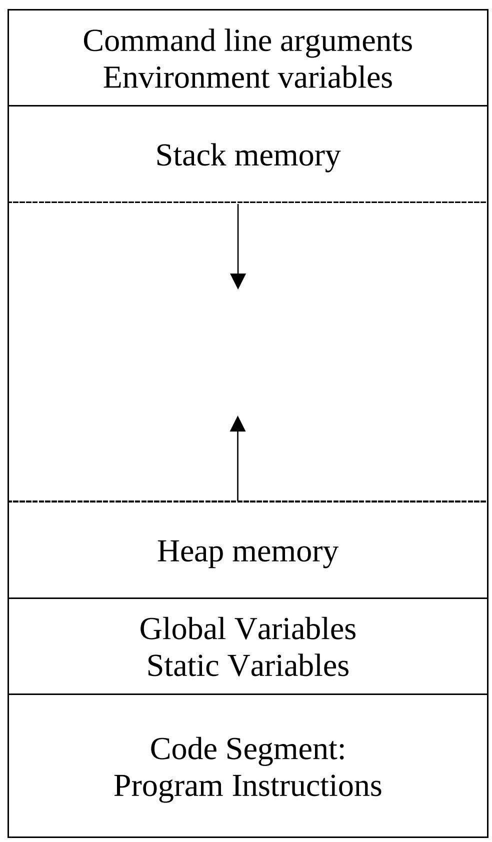
CSE 15
Introduction to Data Structures
Lab Assignment 4

The goal of this assignment is to learn the basics of function pointers, and to use them in a program that evaluates some arithmetic operations encoded in an input file. Begin by reading section 3.3 on Function Pointers in the online text

Understanding and Using C Pointers

(You'll need your CruzID Gold password to access this digital resource from the University Library.)

Memory Layout in C
We will begin with a more detailed picture of program memory for a running C program. We've discussed the heap and stack areas of program memory. We now include sections for other data used by a program.



The code segment contains the program source code, translated into binary machine readable instructions. In particular, it contains a block of instructions for each defined function. As the program runs, the flow of execution traces a pathway through this area of memory, starting at function main(). As it does so, the other areas of memory (like stack and heap) are read from and written to. Each function call causes a new stack frame be allocated, initialized and pushed onto the stack. These frames contain memory for local variables and formal parameters. They also contain the address of the instruction in the code segment to which execution will transfer when the function returns. Multiple calls to the same function generate multiple stack frames, but there is only one block of instructions in the code segment defining a given function. Just as the name of an array in C is a pointer to the first element in the array, the name of a function in C is a pointer to the first instruction in its block within the code segment. Just as array variables (which are pointers to either stack or heap memory) can be passed as arguments to functions, so also function pointers (which are pointers to blocks of memory in the code segment) can be passed as arguments to functions.

Function Pointers

A function pointer is a variable capable of storing the address of a block of instructions in the code segment corresponding to a function. The general form for declaration of a function pointer is

```
return_type (*function_pointer)(……parameter_list……);
```

This declares `function_pointer` to be a variable capable of pointing to any function having the given return type and list of formal parameters. For instance

```
double (*fp)(double, double);
```

declares `fp` to be a pointer variable capable of storing the address of any function that takes as input two doubles, and returns a double. Elsewhere in the program we define

```
double add(double x, double y){
   return x+y;
}
```

and

```
double sub(double x, double y){
   return x-y;
}
```

and

```
double mult(double x, double y){
   return x*y;
}
```

Functions `add()`, `sub()` and `mult()` can then be pointed to by `fp`, which is accomplished by an assignment statement.

```
fp = add;
printf("%f\n", fp(1.0, 2.0));  // prints 3.0
fp = sub;
printf("%f\n", fp(3.0, 4.0));  // prints -1.0
fp = mult;
printf("%f\n", fp(5.0, 6.0));  // prints 30.0
```

The declaration of `fp` above can itself be used as a formal parameter declaration in a function, as follows

```
double apply( double (*fp)(double, double), double x, double y){
   return fp(x, y);
}
```

allowing us to obtain the same results without the assignment statements.

```
printf("%f\n", apply(add,1.0,2.0));   // prints 3.0
printf("%f\n", apply(sub,3.0,4.0));   // prints -1.0
printf("%f\n", apply(mult,5.0,6.0));  // prints 30.0
```

The function pointer declaration `double (*fp)(double, double)` is somewhat awkward as a function parameter. We can create a new data type for such a pointer using a typedef statement. Observe that this is unlike most other uses of the typedef keyword. Normally the new type would be the last identifier in the statement. In this case the new type appears in the middle of the statement.

```
typedef double (*fptr_t)(double, double);
```

The type being created here is `fptr_t`, which follows the C convention of giving types the suffix "`_t`". With this new data type, the function pointer declaration

```
double (*fp)(double, double);
```

is equivalent to

```
fptr_t fp;
```

We can now declare variables of this new type, and assign them to any appropriate function.

```
fptr_t fp1, fp2, fp3;  // three variables of type fptr_t
fp1 = add;
fp2 = sub;
fp3 = mult;
printf("%f\n", fp1(fp2(1.0, 2.0), fp3(3.0, 4.0)) );  // prints 11.0
printf("%f\n", fp2(fp3(1.0, 2.0), fp1(3.0, 4.0)) );  // prints -5.0
printf("%f\n", fp3(fp1(1.0, 2.0), fp2(3.0, 4.0)) );  // prints -3.0
```

See the program FunctionPointerExamples.c posted on the webpage for some working code illustrating all of the above and more.

What to turn in

Write a C program called ArithmeticOperations.c containing the following definitions.

(1) A `typedef` statement creating a data type called `fptr_t` representing a pointer to a function taking two `int`s as input, and returning an `int` as output.

(2) Five functions called `sum()`, `diff()`, `prod()`, `quot()` and `rem()` of the type described in (1) that return the sum, difference, product, quotient and remainder of their two `int` arguments, respectively. For instance $\mathrm{quot}(x, y)$ will return the integer quotient `x/y`. Notice that x/y and y/x are not the same `int` quantity. For the non-commutative operations in the above list, return the quantity: x op y, where x is the first operand, y is the second operand, and op is the operator in question.

(3) A function with heading `int apply(fptr_t fp, int x, int y)` that returns the result of applying the function pointed to by `fp` to the arguments x and y (in that order).

(4) A function with heading `int compute(fptr_t fcn[5], int* A, int* idx, int n)` that returns the value of an expression (described below) built up from its array arguments `fcn[]`, `A[]` and `idx[]`. The integer n is the length of the array `idx[]`, which contains indices (in the range 0 to 4) of the function pointer array `fcn[]`. Each element of `idx[]` is therefore an op-code for an arithmetic operation in `fcn[]`. Array `A[]`, which is of length $n + 1$, contains the corresponding operands. If for instance, `fcn[]` is initialized to `{sum, diff, prod, quot, rem}`, array `idx[]` is initialized to

`{0, 2, 1, 4, 2, 2, 3, 1}`, array `A[]` is initialized to `{3, 2, 5, 4, 6, 7, 9, 2, 8}` and $n = 8$, then the expression to be evaluated is

$$\left(\left(\left(\left(\left(\left(\left((3+2)*5\right)-4\right)\%6\right)*7\right)*9\right)/2\right)-8.$$

One checks that the numerical value of this expression is 86. In general, the expression returned by `compute()` is:

$$\text{fcn}\big[\text{idx}[n-1]\big]\big(\ldots\text{fcn}\big[\text{idx}[2]\big]\big(\,\text{fcn}\big[\text{idx}[1]\big]\big(\,\text{fcn}\big[\text{idx}[0]\big](A[0], A[1]), A[2]\big), A[3]\big), \ldots, A[n]\big).$$

This can be evaluated either iteratively or recursively by calling `apply()` on appropriate arguments.

Once the above types and functions are defined, test them using ArithmeticTest.c posted on the webpage, which contains stubs for all required functions in this project. Your program ArithmeticOperations.c will read one command line argument, giving the name of an input file, and will write one integer to the standard output stream. An input file will contain the following 3 lines of text.

- A single integer n.
- A space separated list of n op-codes in the range 0 to 4, each representing an arithmetic operation.
- A space separated list of $n+1$ operands.

Function `main()` will initialize an array of 5 function pointers to be `{sum, diff, prod, quot, rem}`, as above. It will then read the input file, allocate space for the op-code and operand arrays from heap memory, initialize these arrays to the values on lines 2 and 3 respectively, call function `compute()` on these arrays, and then print the returned value to standard out. Submit the files

ArithmeticOperations.c
Makefile
README

to lab4 before the due date. A few input file examples will be posted on the class webpage under Examples, along with ArithmeticTest.c, FunctionPointerExamples.c and a Makefile which you may alter to your liking. As usual, please start and ask for help early.