**Functions**

So, we've only written our programs in the function called main. However, most computer programs have more than one function. You can think of a function as a mini-program that solves some sort of task. In fact, we've *called* several functions already. When we write printf, scanf, sqrt, and rand, those aren't truly single expressions or statements. Rather, for each of those expressions/statements, a set of several commands gets invoked.

What's nice about functions is that once a set of commands is written (maybe it takes 10 steps to find a square root), we can reuse those steps very easily just by calling the function. Furthermore, functions can solve general tasks. Thus, once a square root function is written, I can use it to find the square root of 17, the square root of 113 and the square root of any non-negative number, for that matter.

In this lecture, we'll learn about functions that take in input parameters and return an output value. This is very similar to the definition of a function you learned in math class.

Here is a program that uses a function:

#include <stdio.h>

int fact(int n);

int main() {

 int n;

 printf("Enter a value of n.\n");

 scanf("%d", &n);

 int answer = fact(n);

 printf("%d! = %d\n", n, answer);

 return 0;

}

int fact(int n) {

 int i, product = 1;

 for (i=1; i<=n; i++) {

 product = product\*i;

 }

 return product;

}

Now, let's examine all the pieces of this, one by one.

In any program with functions, we'll first have our pound includes followed by our function prototypes. The function prototype in this program is

int fact(int n)

This says that there is a function name fact (indicated by the fact, sorry for the pun, that there are parentheses right after the name fact) that takes in as input an integer, which inside the function will be called n, and returns an integer.

In particular, this function calculates the factorial of its input value n.

This is followed by our main function. In main, we've called the fact function. In particular, when we wanted to know the factorial of some value, instead of writing all the statements from scratch to calculate it, we just called the fact function:

int answer = fact(n);

Since the function returns something, we need to call it as part of a greater statement. (Think about all of the times you called the sqrt or pow functions.) What's happening here is that whatever value you put in the parentheses, the fact function will take the factorial of it and return it to you. In this case, that value will then get stored in the variable answer.

One important thing to note about the function call which has been true all along is that you don't put any type information in the function call. You just use the name of the function, and you just directly give it whichever input values you want. Then, you place the function call in a statement so that its return value is appropriately used. In this case, we store the return value of the fact function into the variable answer.

Finally, in the last (meaningful) line of main, we print out the value the function just returned.

After main completes, we can then put our other functions, one after another.

This part is new. In defining a function, you first start with the prototype information. Then, after the open brace, you write the code for the function.

Two important things to keep in mind:

(1) The value of the input parameter, named, n, in this case, is given to you.

(2) The ONLY variables you can use are the ones you declare inside of the function and the variable given to store the input value.

When we write this function, we **ASSUME that n already has a value.** Thus, there are no printfs or scanfs to help establish the value of n. Our goal at the very beginning of the function is to calculate n!

We do this by declaring two variables to aid us in our cause:

int i, product = 1;

From there, we assume that we're the only function that exists, and we try to solve our task at hand. Once we solve it, we return the answer that we've calculated via a return statement. The syntax of a return statement is just the word return followed by a space, followed by whatever expression you want to return. (The expression must resolve to a value of the same type as the return type, which is listed first in the function prototype.)

**Note: The best way to really learn what's going on is to look at pictures illustrating what is happening in memory when functions get called. All of these pictures will be shown in class.**

Using Multiple Functions

Once you've written a function, you can use it in writing new functions. Consider the mathematical function of a combination, which is defined as follows:

$$C\left(n, k\right)=\frac{n!}{k!\left(n-k\right)!}$$

You'll notice that this definition uses the factorial function three times! Thus, we can write a function to compute a combination in C as follows:

int combo(int n, int k) {

 return fact(n)/fact(k)/fact(n-k);

}

Because we had previously defined the fact function, instead of having to write 10 lines, we completed this function in one line!!!

Now, let's look at an entire program that makes use of the combo function:

#include <stdio.h>

int fact(int n);

int combo(int n, int k);

int main() {

 int numCandy, numChoose;

 printf("How many different candy bars are there?\n");

 scanf("%d", &numCandy);

 printf("How many different ones can you choose?\n");

 scanf("%d", &numChoose);

 int mycombo = combo(numCandy, numChoose);

 printf("You can choose %d different combos.\n", mycombo);

 return 0;

}

int fact(int n) {

 int i, product = 1;

 for (i=1; i<=n; i++) {

 product = product\*i;

 }

 return product;

}

int combo(int n, int k) {

 return fact(n)/fact(k)/fact(n-k);

}

One thing that confuses students is that it appears that the program is "jumping" to different parts in the file. Though each function has its statements listed from top to bottom in sequential order, the order of the functions doesn't indicate anything about what sequence they will be called, or even if they'll be called at all. The functions are just definitions. If the functions are called, THEN, they execute their steps in order. Afterwards, they "go back to sleep". Also, each instance of a function being called is completely different. This is why the square root function can find the square root of 17 for you one time and the square root of 113 another.

Now, let's consider the task of writing a power function that works for non-negative exponents. It will be similar to the factorial function, but instead of repeatedly multiplying different numbers, we repeatedly multiply the same number. Here is the end product:

int power(int base, int exp) {

 int answer = 1, i;

 for(i=0; i<exp; i++) {

 answer = answer\*base;

 }

 return answer;

}

Now, imagine the following task:

We want a function that can calculate any of the following sums:

12 + 22 + 32 + ... n2

13 + 23 + 33 + ... n3

14 + 24 + 34 + ... n4

etc.

Our function must take in n, so that it knows how many terms to add. Our function must also take in the exponent of each number (in the examples, 2, 3 and 4 are shown.)

Here is our function:

int powersum(int n, int exp) {

 int sum = 0, i;

 for (i=1; i<=n; i++) {

 sum = sum + power(i, exp);

 }

 return sum;

}

Note very carefully where n was used in the code, where exp was used in the code, where sum was used, and where power was called.

Using these two new functions, we can write a program that aids the programmer in finding various sums. The program is included on the following page.

#include <stdio.h>

int power(int base, int exp);

int powersum(int n, int exp);

int main() {

 int choice = 1;

 while (choice == 1) {

 int num\_terms, exp;

 printf("How many terms in your sum?\n");

 scanf("%d", &num\_terms);

 printf("What power should each be raised to?\n");

 scanf("%d", &exp);

 printf("Your sum is %d.\n", powersum(num\_terms, exp));

 printf("Another sum(yes=1, no=0)?\n");

 scanf("%d", &choice);

 }

}

int power(int base, int exp) {

 int answer = 1, i;

 for(i=0; i<exp; i++) {

 answer = answer\*base;

 }

 return answer;

}

int powersum(int n, int exp) {

 int sum = 0, i;

 for (i=1; i<=n; i++) {

 sum = sum + power(i, exp);

 }

 return sum;

}