Calling Prewritten Functions in C

We've already called two prewritten functions that are found in the C library stdio.h:

printf, scanf.

The function specifications for these two are complicated (they allow a variable number of parameters), so when first learning functions, we don't typically look at these specifications.

The easiest prewritten function specifications included in C libraries to understand are related to mathematics functions and functions used to generate pseudorandom numbers. The former are in the library math.h and the latter are in stdlib.h. Here is an abbreviated listing of some of these functions we will use:

Library math.h

```
double cos(double x); // in radians
double acos(double x);
double cosh(double x);
double exp(double x);
double ceil(double x);
double floor(double x);
double fabs(double x);
int abs(int x);
double pow(double x, double y);
double sqrt(double x);
```

Library stdlib.h(To use random numbers)

```
int rand(void);
void srand(unsigned seed);
```

Reading a Function Specification

Here is the written description along with the function specification for the pow function in the math library:

```
// Returns x raised to the power of y.
double pow(double x, double y)
```

The first item listed, double, represents the return type of the function. Some functions return things and others don't. If a function doesn't return something then void must be written as its return type.

The second item listed is the name of the function. This is always followed by parentheses in the function specification (and also when you call the function).

Inside the parentheses, we have listed the formal parameters. There can be 0 or more of these. The pow function has two. For each formal parameter, we list its type and its variable name. The first is a double named x while the second is a double named y.

The job of the function is to take whatever you give it for x and whatever you give it for y, calculate x^y and return that value to you.

When you *call* a function, you do <u>NOT</u> list any type information. Instead, you just use the name of the function and in place of each formal parameter, you put the actual parameter you want. An actual parameter <u>*can be any expression*</u> that matches the type of the formal parameter given. Consider the example of

calculating compounded interest. The mathematical formula to calculate the end value of the account, given the initial investment, P, the yearly rate, r, and the number of years, t, of investment is:

$$f(P, r, t) = P(1 + \frac{r}{12})^{12t}$$

(Note that the rate is a decimal, typically in between 0 and 1, which is NOT in percentage form.) Imagine that in code we have variables, principle, rate and time, respectively, representing the variables in the formula above. Assume that these three variables have been given values by the user and we wanted to store the value of the end investment in a variable called money. Here is how we could utilize a call to the pow function to accomplish this task:

```
double money = principle*pow(1+rate/12, 12*time)
```

Notice that when we *call* the pow function, it gets called as part of a bigger line of code. In no place in the function call do you see any type information. Notice that neither of the actual parameters we passed to the function, 1+rate/12 and 12*time, are variables and neither have an x or y in them. The power of the pow function (sorry for the pun), is that the user can *call it with any set of actual parameters she sees fit, to make ANY exponentiation calculation that needs to be made.*

The way this works is that the computer sees that we want to calculate pow of something. It figures out the current value of both actual parameters (say rate is .12 and time is 2, then the two actual parameters would be 1.01 and 24, respectively), then it calculates the value of the first value raised to the power of the second one (for this example, it would return $1.01^{24} \sim 1.27$). From here, the computer would replace the section of the line of

code with the function call with this value and then proceed with what the line of code says to do. In this case, the line of code would take 1.27 and multiply it by whatever was stored in the variable P. Then it would take that value and assign it to the variable money.

The key is this: functions that return values should not be called on a line by themselves. They should be called as part of a bigger line. When calling them, replace the formal parameters with actual parameters that are any expression of your choosing. These actual parameters need not have any connection to the formal parameters, in terms of their name. Rather, their values at the time should indicate whatever calculation you wish for the function to do. Note that the *order* you pass in the parameters mattered, had we done `pow(12*time, 1+rate/12)`, the function would have instead returned $24^{1.01} \sim 24.77$ for the example we discussed. I am sure the bank would never make this error!!!

Example of a Program Calling the pow Function

Utilizing the example from above, here is a complete program that calculates the value of an account that uses monthly compounded interest.

```
#include <stdio.h>
#include <math.h>

int main() {
    double principle, rate;
    int time;

    printf("How much are you investing (in dollars)?\n");
    scanf("%lf", &principle);
    printf("What is the yearly rate of return,years of investment?\n");
    scanf("%lf%d", &rate, &time);

    double money = principle*pow(1+rate/12, 12*time);
    printf("In %d years, you will have $%.2lf.\n", time, money);
    return 0;
}
```

Example of a Program Calling the abs Function

Consider the problem of finding the distance between two intersections on a city grid. On a city grid, one must walk in one of the four cardinal directions down a street or avenue. We can give Cartesian coordinates to each city intersection. For example, the distance between (2, 3) and (7, 1), when one can only walk parallel to the x or y axis (same as walking in one of the four cardinal directions) is (7 - 2) + (3 - 1) = 7. This distance calculation is called Manhattan distance, since Manhattan has streets and avenues arranged in a grid like fashion.

In the following program, we utilize the abs function (since we don't know in advance which x coordinate is bigger or which y coordinate is bigger), to calculate the Manhattan distance between two intersections. Mathematically, the Manhattan distance between (x_1, y_1) and (x_2, y_2) is $|x_1 - x_2| + |y_1 - y_2|$. (Of course, one could switch the ordering of the x's or y's within either absolute value symbol…)

```
#include <stdio.h>
#include <math.h>

int main() {

    int x1, y1, x2, y2;
    printf("Please enter the starting x and y coordinates.\n");
    scanf("%d%d", &x1, &y1);
    printf("Please enter the ending x and y coordinates.\n");
    scanf("%d%d", &x2, &y2);

    int dx = abs(x2-x1);
    int dy = abs(y2-y1);

    int dist = dx + dy;
    printf("You have to walk %d blocks.\n", dist);

    return 0;
}
```

Example calling sqrt, cos Functions

Imagine that we want to write a program to solve the following problem:

Prompt the user for two sides lengths of a triangle and the included angle. The program should read these values in, calculate and output the length of the third side.

For those of you who have not taken trigonometry, there is a formula to calculate this third side length from the given information. Let a and b be the given side lengths and angle C be the included angle, then the third side length c can be determined as follows:

c = $\sqrt{a^2 + b^2 - 2ab\cos C}$

Assume that the user will enter the measure of the included angle in degrees and not radians.

Here is a general plan for our program:

1) Ask the user for a, b, and angle C.
2) Calculate the square of a, the square of b and add these.
3) Convert angle C into radians so that we can use the prewritten cos function in the math library.
4) Subtract 2abcosC from the value in 2.
5) Take the square root of the value from step 4.
6) Output the value calculated in step 5.

// Calculates the length of the third side of a triangle given the // other two sides and their included angle..

```
#include <stdio.h>
#include <math.h>

#define     PI    3.14159265358979

int main(void) {

     double a, b, angleC, Crad;
     double sumsq;

     // Get user input
     printf("Enter the lengths of sides a and b.\n");
     scanf("%lf %lf", &a, &b);

     printf("Enter the measure of angle C, in degrees.\n");
     scanf("%lf", &angleC);

     sumsq = pow(a,2) + pow(b,2); // Calculate sum of squares
     Crad = angleC*PI/180;  // Convert C to radians.
     sumsq = sumsq - 2*a*b*cos(Crad); // Subtract appropriate term

     // Calculate and output the square root of the value above.
     printf("The length of side c is %lf\n", sqrt(sumsq));

     return 0;
}
```

Example of Calling a Void Function

Programs are more fun with random numbers. Let's consider a simple program that we'll build on later. Imagine a program that gives a student multiplication practice. If it always gave the same problems to the student, they might just memorize those answers but not know how to multiply all possible pairs of numbers (within some range).

Thus, we want to write a program that generates two random numbers and asks the user for the product of those numbers. Then, our program will tell the user how close they were to the correct answer. (Once we get to the if statement in the next lecture, we can print out different messages depending on whether the student gets the question right or wrong.)

For any program that uses random numbers, we should seed the random number generator. This seeding should occur only once, even though the program itself may generate many random numbers. The function to seed the random number generator is srand and here is its specification:

```
// Seeds the random number generator with
// the integer seed.
void srand(int seed);
```

In reality, a computer can't generate truly random numbers. Instead, it uses a complicated mathematical formula to generate a sequence of numbers that appear to be random. To start the generation, a seed is needed. If you don't seed the random number generator, C always seeds it with 0. What this will do is generate the same stream of random numbers every time you run your program. You don't want to do this, so it's best to seed the random number generator with a different integer each

time. The easiest way to do this is to seed it with something based on the system time. In the time.h library, there is a function time. If you pass this function the parameter 0, it just returns the current time (in seconds after January 1, 1970). What it returns precisely isn't that important. All that matters is that each time you run the program, it returns something different.

Thus, the line of code we want near the beginning of main for any program where we want to generate random numbers is:

```
srand(time(0));
```

Void functions get called on their own line. Since nothing is returned, a return value doesn't have to be stored anywhere. The function just does its task and completes, and then we can move onto the next line of code.

Example Program using Random Numbers

Now, we can write the program described above, after examining the rand function. Here is the specification:

```
// Returns a random integer in between 0 and
// 32767
int rand();
```

Thus, we call the function rand without any parameters and it will return to us a random integer in the range designated above. In most applications, we don't desire random integers in this range. The easiest way to constrain a random number is to use mod. In general, the expression `rand()%n` will always evaluate to an integer in between 0 and n-1, inclusive, since mod returns a remainder which is guaranteed to be in between 0 and 1 less than the number we are dividing by. In general, if we want a

random integer in the range [a, b], where a and b are both positive and smaller than 32767, we can use the following expression: `a + rand()%(b-a+1)`. The latter portion of the expression will always be an integer in the range 0 to b-a. When we add a to this range, we get the range a to b, as desired.

Now, let's look at our multiplication program, where we give the user a random problem where both operands are in between 1 and 12, inclusive. Notice the use of the abs function to print out how close to the answer the user was. Notice that it can be "inside of" a printf and that the actual parameter can be a complicated expression.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define MAX 12

int main() {

    // Seed the random number generator (just one time)
    srand(time(0));

    int x, y, answer;

    // Generate our two operands and ask the user for the answer.
    x = 1 + rand()%MAX;
    y = 1 + rand()%MAX;
    printf("What is %d x %d?\n", x, y);

    // Read the answer in.
    scanf("%d", &answer);

    // Print out how close the user was.
    printf("You were %d away from the answer.\n", abs(answer-x*y));

    return 0;
}
```