Functions

There are a few redundant things observed with the final solution of the string comparison problem in the previous section. Checking whether the string input is composed of letters only was done twice, once for st1, and another for st1. The same goes for storing st1 in tmp1, and storing st2 in tmp2. These two tasks do exactly the same thing. Copy. And lastly, we did two sets of case conversion. One for tmp1, and another for tmp2. Again, these two do the exact same thing. Convert. Is there a way to somehow just write one source for sub-tasks and then just re-use them when the need arises?

Let's look at a smaller problem first. Computing for x^{y^z}. This problem requires us to solve y^z first (so we store the result in res), and then compute for x^{res}.

Let's take a look at solution (down) below.

```c
#include <stdio.h>

//problem: (x^(y^z))

int main(){
    int x, y, z, i, temp, ans;

    do{
        printf("Enter a positive number for x: ")
        scanf("%i",&x);
    }while(x<1);

    do{
        printf("Enter a positive number for y: ")
        scanf("%i",&y);
    }while(y<1);

    do{
        printf("Enter a positive number for z: ")
        scanf("%i",&z);
    }while(z<1);

    //computes for y^z and stores result in temp
    for(i=0,temp=1; i<z; i++)
        temp = temp * y;

    //computes for x^temp, where temp is y^z
    for(i=0,ans=1; i<temp; i++)
        ans = ans * x;

    printf("%i^(%i^%i) = %i\n",x,y,z,ans);
}
```

A similar observation as the string comparison solution is the computation for the powers above. Two for loops, both alike in power computation (in fair main, where we lay our scene ...).

Kidding aside, the two for loops do exactly the same thing. They perform a number of multiplications. The compute for some base raised to some exponent!

We ask the same question here. Is there a way to code the power computation once, and just re-use it when we need it again? And the answer to the question is a resounding yes! Functions! You have seen and used functions already. `printf`, `scanf`, `strlen`, and even the `main`, they are all functions. How do we construct or implement them?

If a program needs to model a solution, most of the time the tasks involved are numerous and large (think string comparison and magnify that 10- or even a hundred fold!). So it is always best to decompose the solution into smaller pieces of code. In other words, these numerous and large tasks are delegated to these smaller pieces of code called modules.

Since these modules are smaller, they are more manageable resulting in better program maintainance. Errors are now easier to fix because they will occur locally in the different modules of the program. There is no need to look at a really long program written in the main function only. These modules are implemented in c using functions.

Once modules are implemented in functions, all the program has to do is to use them. And when the program needs a module many times, there is no need to rewrite code as many times. The program can use the module as many times as it needs.

We just described the benefits of functions. The first one is modularity, and the second one is code re-use.

Declare, Define, Call

Before a function can be used, it has to be declared and defined first.

Declaring functions follows the template below:

```
returnType functionName(parameterList);
```

Function declarations are usually placed before the `main` function.

Some samples are:

- `int func1(int, int)`
  - name of function: func1
  - parameters or arguments: two parameters as arguments both are int
  - return type: an int
- `int func2()`
  - name of function: func2
  - parameters or arguments: none
  - return type: an int
- `void func3()`
  - name of function: func3
  - parameters or arguments: none
  - return type: void or none

In naming your functions, the same rule applies as in naming your variables/identifiers. It should start with a letter or the underscore character. It can be followed by a series of alphanumeric characters including the underscore as well. It cannot contain special characters and operators and the space character. There is no limit on the length of the name of the function. This is exactly the same rule for namin a variable or identifier. And the only thing

that distinguishes a function from a variable is the pair of parenthesis that follows after its name regardless whether it accepts arguments/parameters or not.

Name your functions as descriptive of what they do as possible. If your function is to contain more than one word, write the first word in lowercase then capitalize the first letters only of the succeeding words. For example:

- `void functionOne()`
- `void anotherFunctionOne()`

This is just a convention. You will not encounter any syntax errors if you are not going to follow this convention.

The name of the function is used for calling or for invoking it.

The parameter list can contain zero or more elements. There is no limit to this number. If the function is not going to contain any parameter, leave it blank or use void. The parentheses are required even if the function will not contain any parameter. The parameter list or the argument list is used to pass information or data to the function. It is a way of communicating to the module. Take `strlen`, the parameter there is a string.

The return type is any of the available data types in c like int, double, float, char, an array (pointer) of the available data types in c, a user-defined data type (enums and structures - discussed at the last section). If the parameter list is used to pass information or let other functions, including the main, to communicate to the function, the return (type) is used by the function to send back message (data or information) to the caller of the function. Take `strlen`, again. The return type is an int. And the function returns the length of the string in the argument list. Using `strlen` somehow provided for a communication between the main function and the strlen function.

If a function is not to return anything, we use void as its return type. This type of function is sometimes referred to as a procedure.

Defining the function means giving it behavior. Declaration is not enough. The function is of no use if it does not have any definition. This is the part where you code the function's behavior. How does this module function? Write the definition after the main function.

```
int func1(int, int);    //function declaration

int main(){
    return 0;
}

int func1(int a, int b){ //function definition
    /*insert code here*/
}
```

Copy the function template exactly as how it was declared. But for the function definition, the arguments must have names this time. Because that's the only way we can retrieve information, via the identifier.

Let's put some definition into the function.

```
int sum(int, int);  //function declaration

int main(){
    return 0;
}

int sum(int a, int b){ //function definition
    int ans = a + b;
    return ans;
}
```

Since the return type of `sum` is an int, then a value of the same type must be *returned* by the function through the use of the keyword `return` followed by the identifier with the same type as the return type. In the case above, it is valid to return *ans* because it is of type int as well. Whenever the `return` is executed, the control of execution exits from the function.

Note: We only do this when the return type is NOT void.

Compile the source code above and run it. Notice that nothing happens. This should be obvious since there is nothing in the main function except for the `return 0;`. What we are trying to say here is that just because we have declared and defined the function doesn't mean it gets executed. Control of the flow of execution must be explicitly transferred to it. And we do this by invoking or calling the function.

Calling the function is done by using its name and passing the required arguments or parameters if there are any. A function can be called in the main function or in some other function or even inside itself (we call this a recursive call).

```c
#include <stdio.h>
void display();

int main(){
    printf("Before calling display().\n");
    display(); //calling or invoking display
    printf("After calling display().\n");
    return 0;
}

void display(){
    printf("You are inside display()");
}
```

In the case above, no argument was passed since function display does not accept any parameter. When the program is run, "Before calling display()." is displayed. Then `display()` is called. When this happens, the control of the execution is transferred to it. This means that the printf statement after the call to display() will be executed only when display() terminates and control is given back to the main() function and proceeds the execution of the program by executing printf statement displaying "After calling display()."

```c
#include <stdlib.h>
void display(int);

int main(){
    display(3); //calling or invoking display
    return 0;
}

void display(int a){
    printf("You are inside display()");
    printf("You passed the value %i.",a);
}
```

In this example, a value 3 was passed as an argument since display requires a parameter (an int). This may not always be a literal value. It may be a variable as well as long as its type is int.

```c
#include <stdlib.h>
int display(int);

int main(){
    int temp = display(3); //calling or invoking display
    printf("temp: %d\n",temp);
}

int  display(int a){
    printf("You are inside display()\n");
    printf("You passed the value %d.\n",a);
    return a * a;
}
```

Since display in this example returns an int, a variable of the same type as that of function's return type must receive the value. In this case, temp.

```c
#include <stdlib.h>
int display(int);

int main(){
    display(3); //calling or invoking display
    //printf("temp: %d\n",temp);
}

int  display(int a){
    printf("You are inside display()\n");
    printf("You passed the value %d.\n",a);
    return a * a;
}
```

If the value returned by the function is not received by a variable, no syntax error will be encountered. But this defeats the purpose of letting functions return values. And this would render the printf statement not usable.

We may also directly call the display() inside the printf since it returns an int.

```c
#include <stdlib.h>
int display(int);

int main(){
    //calling or invoking display from inside the printf
    printf(value returned by display(): %d\n",display(3));
}

int display(int a){
    printf("You are inside display()\n");
    printf("You passed the value %d.\n",a);
    return a * a;
}
```

We have to make sure that the format spcifier in the printf matches the return type of the function. This can only be done when the function is non-void.

The above example demonstrates how parameter passing and returning values can be used to let the functions communicate with each other in an efficient manner.

We are now ready to do x^{y^z}. We need to implement a power function that accepts 2 integers, the base and exponent, and have it return the result.

```c
#include <stdio.h>

int power(int,int); //function prototype

int main(){
    int x, y, z, ans;

    do{
        printf("Enter a positive value for x: ");
        scanf("%i",&x);
    }while(x<=0);

    do{
        printf("Enter a positive value for y: ");
        scanf("%i",&y);
    }while(y<=0);

    do{
        printf("Enter a positive value for z: ");
        scanf("%i",&z);
    }while(z<=0);

    ans = power(y,z);
    ans = power(x,ans);

    printf("%i^(%i^%i) = %i\n",x,y,z,ans);
    return 0;
}

int power(int b, int e){
    int i, ans = b;
    for(i=1; i<e; i++)
        ans = ans * b;

    return ans;
}
```

We only implemented one power function but it can be called many times over. As shown above, power is called twice. Once for the computing x^y, and another for computing x^{ans}.