Functions

A function is a sequence of statements that have been grouped together and given a name.

Each function is essentially a small program, with its own declarations and statements.

Some advantages of functions:
- A program can be divided into small pieces that are easier to understand and modify.
- We can avoid duplicating code that’s used more than once.
- A function that was originally part of one program can be reused in other programs.
- The memory cost of the program can be reduced if the memory needed by each a function is released when the function terminates.

The Caesar Cipher example discussed earlier provides an illustration of the use of functions in the design and implementation of a small C program.
Function Definitions

General form of a function definition:

```
return-type function-name ( parameters )
{
  declarations
  statements
}
```

```c
double average(double a, double b) {
  return (a + b)/2.0;
}
```

Functions may not return arrays.

Specifying that the return type is `void` indicates that the function doesn’t return a value.

If the return type is omitted in C89, the function is presumed to return a value of type `int`.

In C99, omitting the return type is illegal.
Function Definitions

Variables declared in the body of a function can’t be examined or modified by other functions.

In C99, variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

Functions that do not return a value are declared with a return type of `void`.

The returned valued from a call to a non-`void` function may be ignored.

Many `void` functions can be improved by using a return type of `bool`. 
Function Declarations

C doesn’t require that the definition of a function precede its calls:

```c
#include <stdio.h>

int main(void) {
    double x, y;

    printf("Enter two numbers: ");
    scanf("%lf%lf", &x, &y);
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
    return 0;
}

double average(double a, double b) {
    return (a + b) / 2;
}
```

The C compiler assumes that the function returns an `int` value.
A function declaration provides the compiler with a brief glimpse at a function whose full definition will appear later.

The general form of a function declaration:

```
return-type function-name ( parameters ) ;
```

The declaration of a function must be consistent with the function’s definition.

A function declaration looks just like the first line of the function definition:

```
double average(double a, double b);
```

Note: the parameter names can be omitted, but not the parameter types.
Formal vs Actual Parameters

**Formal parameters** are the names used in the function definition.

**Actual parameters** are the names used in the function call.

```c
#include <stdio.h>

double average(double a, double b);

int main(void) {
    double x, y;

    printf("Enter three numbers: ");
    scanf("%lf%lf", &x, &y);
    printf("Average of %g and %g: %g\n", x, y, average(x, y));

    return 0;
}

double average(double a, double b) {
    return (a + b) / 2;
}
```
Formal parameters have automatic storage duration and block scope, just like local variables.

Formal parameters are automatically initialized with a copy of the value of the corresponding actual parameter.

There is no connection between the actual and formal parameters other than that they store the same value at the time of the function call.

```c
int Exp = 10;
int Base = 4;

int BaseToExp = Power(Base, Exp);
printf("%d ^ %d = %d\n", Base, // still 4
        Exp,       // still 10
        BaseToExp);
```

```c
int Power(int X, int N) {
    int Result = 1;
    while (N-- > 0) {
        Result = Result * X;
    }
    return Result;
}
```
```c
int main() {
    . . .
    int Exp = 10;
    int Base = 4;

    int BaseToExp = Power(Base, Exp);

    printf("%d ^ %d = %d\n",
           Base,              // still 4
           Exp,              // still 10
           BaseToExp);
    . . .
}

int Power(int X, int N) {
    int Result = 1;

    while (N-- > 0) {
        Result = Result * X;
    }

    return Result;
}
```

<table>
<thead>
<tr>
<th>Exp</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>4</td>
</tr>
<tr>
<td>BaseToExp</td>
<td>??</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>4</td>
</tr>
<tr>
<td>Result</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>0</td>
</tr>
<tr>
<td>Result</td>
<td>10000</td>
</tr>
</tbody>
</table>
The Stack

The execution of a C program is organized by use of a collection of *stack frames* (or *activation records*) stored in a stack structure.

Each time a function is called, a stack frame is created and pushed onto the stack.

The stack frame provides memory for storing:
- values of parameters passed into the function
- values of local variables declared within the function (unless they're *static*)
- the return address (of the instruction to be executed when the function terminates)

We will examine the details of the stack later.
For now, we'll restrict our attention to single-file programs. The typical organization is:

```
// include directives
...

// file-scoped declarations of functions
//    and constants
...

int main() {
    ...
}

// implementations of other functions
```