Introducing functions

C is a procedural programming language, meaning that its primary synchronous control flow mechanism is the `procedure call`.

C names its procedures `functions` (in contrast, Java has a different mechanism - `methods`).

- In Mathematics, we `apply` a function, such as the trigonometric function `cos`, to one of more values.
  The function performs an evaluation, and `returns` a result.

- In many programming languages, including C, we `call` a function. We pass zero or more `arguments` to the function, the function's statements are executed (often involving the arguments), and a result is `returned`.

We've already seen the example of `main()` - the function that all C programs must have, which we might write in different ways:

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    // check the number of arguments
    if(argc != 2) {
        ....
        exit(EXIT_FAILURE);
    } else {
        ....
        exit(EXIT_SUCCESS);
    }
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    int result;
    // check the number of arguments
    if(argc != 2) {
        ....
        result = EXIT_FAILURE;
    } else {
        ....
        result = EXIT_SUCCESS;
    }
    return result;
}
```

The operating system calls `main()`, passing to it some (command-line) arguments, `main()` executes some statements, and `returns` to the operating system a result - usually `EXIT_SUCCESS` or `EXIT_FAILURE`. 
Why do we require functions?
The need for, and use of, main() should be clear. However, there's 3 other primary motivations for using functions:

1. Functions allow us to group together statements that have a strongly related purpose - statements, in combination, performing a single task.
   
   We prefer to keep such statements together, providing them with a name (as for variables), so that we may refer to the statements, and call them, collectively.
   
   This provides both convenience and readability.

2. We often have sequences of statements that appear several times throughout larger programs.
   
   The repeated sequences may be identical, or very similar (differing only in a very few statements). We group together these similar statement sequences, into a named function, so that we may call the function more than once and have it perform similarly for each call.
   
   Historically, we'd identify and group similar statements into functions to minimize the total memory required to hold the (repeated) statements.
   
   Today, we use functions not just to save memory, but to enhance the robustness and readability of our code (both good Software Engineering techniques).

3. Functions provide a convenient mechanism to package and distribute code. We can distribute code that may be called by other people's code, without providing them with a complete program.
   
   We frequently use libraries for this purpose.
Where do we find functions?

1. We'll initially write our own functions, in the same file as `main()`, to simplify our code and to make it easier to read.

2. Soon, we'll write our own functions in other, multiple files, and call them from our main file.

3. Collections of related functions are termed **libraries** of functions.

   The most prominent example, that we've already seen, is C's standard library - a collection of frequently required functions that must be provided by a standards' conforming C compiler.

   In our programming, so far, we've already called library functions such as:

   ```
   printf(), atoi(), and exit().
   ```

4. Similarly, there are many task-specific 3rd-party libraries. They are not required to come with your C compiler, but may be downloaded or purchased - from Lecture 1 -

<table>
<thead>
<tr>
<th>Function domain</th>
<th>3rd-party libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>operating system services (files, directories, processes, inter-process communication)</td>
<td>OS-specific libraries, e.g. glibc, Win32, Carbon</td>
</tr>
<tr>
<td>web-based programming</td>
<td>libcgi, libxml, libcurl</td>
</tr>
<tr>
<td>data structures and algorithms</td>
<td>the generic data structures library (GDSL)</td>
</tr>
<tr>
<td>GUI and graphics development</td>
<td>OpenGL, GTK, Qt, UIKit, Win32, Tcl/Tk</td>
</tr>
<tr>
<td>image processing (GIFs, JPGs, etc)</td>
<td>gd</td>
</tr>
<tr>
<td>networking</td>
<td>Berkeley sockets, AT&amp;T's TLI</td>
</tr>
<tr>
<td>security, cryptography</td>
<td>openssl, libmp</td>
</tr>
<tr>
<td>scientific computing</td>
<td>NAG, Blas3, GNU scientific library (gsl)</td>
</tr>
<tr>
<td>concurrency, parallel and GPU programming</td>
<td>pthreads, OpenMPI, openLinda, CUDA, OpenCL</td>
</tr>
</tbody>
</table>
The role of function main()

In general, even our small programs will have several functions.

We will no longer place all of our statements in the main() function.

main() should be constrained to:

- receive and check the program's command-line arguments,
- report errors detected with command-line arguments, and then call exit(EXIT_FAILURE),
- call functions from main(), typically passing information requested and provided by the command-line arguments, and
- finally call exit(EXIT_SUCCESS) if all went well.
The **datatype of a function**

There are two distinct categories of functions in C:

1. functions whose role is to just perform a task, and to then return control to the statement (pedantically, the expression) that called it.

   Such functions often have *side-effects*, such as performing some output, or modifying a global variable so that other statements may access that modified value.

   These functions don’t return a specific value to the caller, are termed *void functions*, and we casually say that they “return void”.

2. functions whose role is to calculate a value, and to return that value for use in the expressions that called them. The single value returned will have a *type*, such as `int`, `char`, `bool`, or `float`.

   These functions may also have side-effects.

```c
#include <stdio.h>
#include <stdlib.h>

void output(char ch, int n)
{
    for(int i=1; i<=n; i++)
    {
        printf("%c", ch);
    }
}

int main(int argc, char *argv[])
{
    output(’,’, 19);
    output(‘\n’, 1);
    return 0;
}
```

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

float square(float x)
{
    return x * x;
}

int main(int argc, char *argv[])
{
    if(argc > 2)
    {
        float a, b, sum;
        a = atof(argv[1]);
        b = atof(argv[2]);
        sum = square(a) + square(b);
        printf("hypotenuse = %f\n", sqrt(sum));
    }
    return 0;
}
```

Note that this 2nd example uses the `sqrt()` function from the standard maths *library*.

We should thus compile it with: `cc [EXTRAOPTIONS] -o program program.c -lm`
Passing parameters to functions
The examples we've already seen show how parameters are passed to functions:

- a sequence of expressions are separated by commas, as in:

  ```
  a = average3( 12 * 45, 238, x - 981 );
  ```

- each of these expressions has a datatype. In the above example, each of the expressions in `anint`.

- when the function is called, the expressions are evaluated, and the value of each expression is assigned to the parameters of the function:

  ```
  float average3( int x, int y, int z )
  { 
      return (x + y + z) / 3.0;
  }
  ```

- during the execution of the function, the parameters are local variables of the function.

  They have been initialized with the calling values (x = 12 * 45 ...), and the variables exist while the function is executing. They “disappear” when the function returns.

- Quite often, functions require no parameters to execute correctly. We declare such functions with:

  ```
  void backup_files( void )
  { 
      ..... 
  }
  ```

  and we just call the functions without any parameters: `backup_files();`
Very common mistakes with parameter passing

Some common misunderstandings about how parameters work often result in incorrect code (even a number of textbooks make these mistakes!):

- The **order of evaluation** of parameters is **not defined** in C. For example, in the code:

```c
int square(int a)
{
    printf("calculating the square of %i\n", a);
    return a * a;
}

void sum(int x, int y)
{
    printf("sum = %i\n", x + y);
}

....
sum(square(3), square(4));
```

are we hoping the output to be:

```
calculating the square of 3  // the output on PowerPC Macs
calculating the square of 4
sum = 25
```

or

```
calculating the square of 4  // the output on Intel Macs
calculating the square of 3
sum = 25
```

Do not assume that function parameters are evaluated left-to-right. The compiler will probably choose the **order of evaluation** which produces the most efficient code, and this will vary on different processor architectures.

(A common mistake is to place auto-incrementing of variables in parameters.)
Very common mistakes with parameter passing, continued

Another common mistake is to assume that function arguments and parameters must have the same names to work correctly.

Some novice programmers think that the matching of names is how the arguments are evaluated, and how arguments are bound to parameters. For example, consider the code:

```c
int sum3( int a, int b, int c )
{   return a + b + c;
}
....
int a, b, c;
a = 1;
b = 4;
c = 9;
printf("%d\n", sum3(c, a, b));
```

Here, the arguments are not "shuffled" until the names match.

It is not the case that arguments must have the same names as the parameters they are bound to. Similarly, the names of variables passed as arguments are not used to "match" arguments to parameters.

If you ever get confused by this, remember that arithmetic expressions, such as \(2 \times 3 + 1\), do not have names, and yet they are still valid arguments to functions.
Very common mistakes with parameter passing, continued

While not an example of an error, you will sometimes see code such as:

```c
return x;
```

and sometimes:

```c
return(x);
```

While both are correct, the parentheses in the 2nd example are unnecessary. `return` is not a function call, it is a statement, and so does not need parentheses around the returned value.

However - at any point when writing code, use extra parentheses if they enhance the readability of your code.
Functions receiving a variable number of arguments

To conclude our introduction to functions and parameter passing, we consider functions such as `printf()` which may receive a variable number of arguments!

We've carefully introduced the concepts that functions receive strongly typed parameters, that a fixed number of function arguments in the call are bound to the parameters, and that parameters are then considered as local variables.

But, consider the perfectly legal code:

```c
int i = 238;
float x = 1.6;
printf("i is %i, x is %f\n", i, x);
.....
printf("this function call only has a single argument\n");
.....
printf("x is %f, i is %i, and x is still %f\n", x, i, x);
```

In these cases, the first argument is always a string, but the number and datatype of the provided arguments keeps changing.

`printf()` is one of a small set of standard functions that permits this apparent inconsistency. It should be clear that the format specifiers of the first argument direct the expected type and number of the following arguments.

Fortunately, within the ISO-C99 specification, our `cc` compiler is permitted to check our format strings, and warn us (at compile time) if the specifiers and arguments don't "match".

Another good justification for our use of "strong" compilation options:

```
cc -std=c99 -Wall -Werror -pedantic ...
```

We'll see later in the unit how we can correctly write our own functions to accept a variable number of arguments.