

Functions, or how to think about fewer things at a time

Comments and questions to [John Rowe](#).

The basic concept of a function is extremely simple: a function is a self-contained piece of code, which can have its own local variables, and which either performs a specified task or calculates and returns a value. It's almost like a separate "mini-program" that we can write separately from the main program, without having to think about the rest of the program when we are writing it.

Examples of the "perform a task" type of function include `printf()` whilst the "calculate and return a value" type include the `math.h` functions `sin()`, `cos()`, `sqrt()`, etc.

We've already written one function, `main()`. Before we see how to write other functions we should ask "Why bother? Why not just put everything inside `main()`?"

Once we understand this paragraph that's almost all we need to know!

When and why are functions useful?

The usefulness of functions comes from a combination of two factors.

1. Code that gets used more than once

Recall the code snippet from the previous lecture that calculated a power for a positive integer exponent:

```
result = 1.0;
while ( exponent > 0 ) {
    result = result * value;
    exponent = exponent - 1;
}
```

When faced with wanting to use this piece of code twice, we may be tempted to just copy-and-paste it to a new place, thus having two copies of the above snippet in our code. A **much** better approach is to make it into a function and to then call that function twice.

Of course, if we know we are likely to use some code several times we should write it as a function at the very beginning. But if we do write some code and then find we need to use it again, we mustn't be afraid to convert it into its own separate function. This is usually the better option.

Why use a function, not copy and paste?

1. **It's shorter.** This is surprisingly important as we will spend quite a lot of time looking through our code trying to find mistakes (bugs). More code means more to look through.
2. **It's easier to test.**
3. **It's easier to fix and extend.**

As an example of the second two points, consider first that the code above doesn't handle negative values of `exponent`. We can fix this easily enough, although the code is now complicated enough to warrant a brief comment:

```
/* Calculate "value" to the power of "exponential" */
result = 1.0;
while ( exponent > 0 ) {
    result = result * value;
    exponent = exponent - 1;
}
while ( exponent < 0 ) {
    result = result / value;
    exponent = exponent + 1;
}
```

Often, such problems only get noticed after the code has been used for a while so we would have to go through the code making sure we had found every copy and fixing them all.

Problem: what if `value == 0`?

if we look at the new code above we see a problem: it divides `result` by `value` which will clearly give us problems if `value` equals zero. It's not clear how we should deal with this: do we use an `if ()` to test for it and if `value` is zero what do we do? Leaving this issue scattered throughout the code should make us feel uneasy.

I would suggest we should at least print a warning message.

2. Hiding complexity: out of sight, out of mind

The power code above is getting a bit complicated and if we were in the middle of something that was already quite complicated we could get quite distracted by it. We saw a similar problem in our preface lecture where we could all easily multiply two one-digit numbers together but had problems with two-digit numbers (over twelve!).

You may have already experienced this with your course work: a piece of code may be quite easy to understand but adding just a little bit more complexity leads to a disproportionate increase in the difficulty of understanding it. (If you have found this, don't worry: it's not just you! If you haven't found it yet: you will.)

If it is simple to describe what a piece of code does but quite complicated to actually write it, it is much easier to write it as a separate "mini-program" or **function** that lets us think about the task separately from the main program. Conversely, when we return to the main program we can use our function without having to think about how it does its job, just as we do with `sin()` or `sqrt()`.

However, **this requires that we have thoroughly tested our function**, as otherwise we cannot rely on it.

Criteria for a good function

It follows that for a chunk of code to be a good candidate to be made into a separate function, even if we are using it only once, it must simplify our mental model ("coding") of the problem by allowing us to think, not of the individual steps involved but of a simple description of the task.

A good function:

- Does a **single, specific task**.
- Is considerably more complicated to implement than to describe.
- Can be specified **unambiguously**.
- If there are variables that are needed to perform the task but are not needed afterwards that could well be a sign that it should be a function.

The above criteria operate in parallel with the "used several times" criterion above: a piece of code that we might think was fine to have in the main code with a comment if we were only using it once could become a function if we needed it twice.

As an example, if we only wanted to use it once then our "power" code above could well be either be written "inline" inside the main code with a comment or be its own separate function. If we wanted to use it twice it would definitely be a function.

Whenever we are tempted to copy and paste more than a couple of lines of code it should almost certainly be a function.

Use cut and paste, not copy and paste.

The markers will be extremely tough on this

Writing our own functions

The following shows how we can package our previous "power" code into its own function (with a fairly crude way of handling the "value is zero, exponent is negative" problem):

```
/*
 * Return "value" to the power of "exponent".
 * If "value" == zero and "exponent" < zero,
 * print a warning and return zero
 */
double mypow(double value, int exponent) {
    double result = 1.0;

    if (value == 0.0 && exponent < 0 ) {
        printf("WARNING: mypow called with zero value and negative exponent\n");
        return 0.0;
    }

    while (exponent > 0) {
        result = result * value;
    }
}
```

```

    exponent = exponent - 1;
}

while (exponent < 0) {
    result = result / value;
    exponent = exponent + 1;
}

return result;
}

```

Notice that the comment before the function is both short and specific.

We have slightly dodged the question of how to handle zero to negative powers by just printing a warning and returning the value of zero. With marginal cases like this, what we do is sometimes less important than the fact that it is clearly documented.

Apart from the fact that `mypow()` is declared as `double` not `int`, the main difference between `mypow()` and `main()` is that the parentheses following the function name now contain:

```
double value, int exponent
```

This declares two **local** variables called **arguments** which can then be used within the body of the function just like any variables declared within that function's `{ ... }` block. The only thing that distinguishes arguments from "ordinary" variables declared inside the `{ ... }` block is that the arguments have their values initialised to the values of the corresponding expressions passed as parameters when the function is called.

For example, if `mypow()` is called from within `main()`, or any other function, as:

```

double a = 3.0, b = 1.0, x, y;
int j = 6;

x = mypow(2.0, 3);
y = mypow(a - b, j/2);

```

Then for each call to `mypow()` the arguments (local variables) `value` and `exponent` will be initialised to 2.0 and 3 respectively.

`mypow()` then calculates the required answer, which we have imaginatively called "`result`" and the statement:

```
return result;
```

causes the function to finish and return the value of "`result`".

Thus inside the calling function `x` and `y` will have the value 8.0. Notice how in the second example **we have used arithmetic expressions in the parameters**. This is absolutely fine.

Local variables: a vitally important principle

Two things should be noted from the body of the function:

1. `mypow()` declares a local `double` variable, called "`result`".
2. One of the arguments, `exponent`, has its value changed as the function executes, ending up with the value zero (except for the "early return" case).

Local variables are precisely that: local

Inside the body of `mypow()`, we saw the local variable called "`result`". This raises the obvious question: "if we have a variable called `result` in another function, are they in any way related?". If we change `result` inside `mypow()`, will the value of `result` in the other function also change?

A little thought will soon show that it would be almost impossible to write a computer program if that were the case. For example, although we have never seen the source code for the `printf()` function, we can be sure it contains some local variables. It would be disastrous if we were to accidentally give one of our variables the same name as one inside `printf()`: we would call `printf()` and the value of our variable would suddenly change!

Example

The following function calls `mypow()` in the doomed hope that its own local variable "`result`" will be set to the value of the (different) local variable "`result`" inside `mypow()`.

```

void badfun() {
    double result; /* The same name as inside mypow() */

    mypow(7.2, 4);
}

```

Remember

An *expression* may be a constant, the value of a variable or an arithmetic expression involving "+ - /* %", etc.

I would be pretty upset if somebody else committed a murder and they sent me to jail on the grounds that we had the same first name

The word "void" means that the function doesn't return a value. In this case the "return" statement

```
printf("7.2 to the power 4 is %d\n", result); /* Wrong! */
}
```

is optional.

Arguments are temporary, local copies of the calling parameters

Less obviously, the same applies to the arguments. Consider the following three ways of calling `mypow()`

```
int main() {
    double testval = 2.0, value = 2.0;
    int testexp = 3, exponent = 3;
    double pow1, pow2, pow3;

    pow1 = mypow(2.0, 3);
    pow2 = mypow(testval, testexp);
    pow3 = mypow(value, exponent);

    printf("power is %f, exponent is %d\n", pow3, exponent);
    return 0;
}
```

In each case, the argument `exponent` inside `mypow()` is initialised to three and during the course of the function it is assigned the value of zero. Whilst it's fairly clear that `mypow()` won't be able to change the value of the mathematical constant "3" to zero(!), it's not quite so clear in the second and third case that `testexp` and/or `exponent` won't have their value changed to zero.

In the third case we've gone as far as using the same variable names as the actual function arguments: the variable `exponent` "happens" to have the same name as used within the body of `mypow()`. Might the compiler take this as some sort of hint?

C is completely consistent and in all cases the argument within the function is a **brand new local variable** whose only connection to the calling parameter is that the value of the parameter is copied to become the initial value of the local argument.

Some programming adopt the opposite approach and **do** change the value of the calling parameter. In some of them, if we then call the function with a constant as a parameter the program will crash.

A few other points about functions

Functions don't have to return a value

We saw an example of this above with `badfun()`. In this case the function is declared as type `void` and `return` statement within the function is optional: it will just return when it reaches the end.

Here is an example that itself calls `mypow()`:

```
/*
 * Print out x to the power j, warning if x and j are invalid
 */
void printexp(double x, int j) {
    if ( x == 0.0 && j < 0 ) {
        printf("Invalid values to printexp\n");
        return;
    }
    printf("%f to the power %d is %f\n", x, j, mypow(x, j));
}
```

Functions are always declared outside of other functions

Trying to declare a function inside of another function will produce an error.

Checking the number of arguments, etc.

It's quite easy to miss out an argument to a function. In addition, if we use a function inside another arithmetic expression the compiler will need to know what type of result (`int`, `double`, etc) it returns. An obvious example is:

```
y = mypow(x, 4) / 8;
```

where it needs to know whether to use integer or floating-point division.

There are two ways of dealing with this:

1. Have the function code before the place where it is called

This does work but has the disadvantage that our program ends up being in the reverse order with `main()` at the end.

We may want to consider returning an integer value to tell the calling function whether `printexp` succeeded or failed.

making it hard to understand.

2. Explicitly tell the compiler the type of the function and its arguments

This is called a **prototype**, it appears **outside of any function**, before either the function itself or any other function that calls it. It looks like:

```
double mypow(double value, int exponent);
```

The names of the arguments are optional in function prototypes.

The easiest way to do this is to copy and paste the start of our function declaration and to replace the brace "{" with a semicolon.

A **prototype doesn't declare a new function**, it just says that a function is declared somewhere else and tells the compiler its type and the types of its arguments.

If the compiler then meets either the function declaration, or a call to the function, and either does not match the prototype it will complain.

The conventional place to put our prototypes is just before the first function, as in this complete example:

```
/*
 * Demonstrate function prototypes
 */
#include <stdio.h>

double mypow(double value, int exponent);
void printexp(double x, int j);

int main() {
    printexp(5.3, 2);
    return 0;
}

/*
 * Print out x to the power j, warning if x and j are invalid
 */
void printexp(double x, int j) {
    if ( x == 0.0 && j < 0 ) {
        printf("Invalid values to printexp\n");
        return;
    }
    printf("%f to the power %d is %f\n", x, j, mypow(x, j));
}

/*
 * Return "value" to the power of "exponent".
 * If "value" == zero and "exponent" < zero, return zero
 */
double mypow(double value, int exponent) {
    double result = 1.0;

    if (value == 0.0 && exponent < 0 )
        return 0.0;

    while (exponent > 0) {
        result = result * value;
        exponent = exponent - 1;
    }

    while (exponent < 0) {
        result = result / value;
        exponent = exponent + 1;
    }

    return result;
}
```

Summary

When to use a function

- When we have a self-contained task that is considerably harder to write than to describe what it does, having it as a function allows us to write and test it without thinking about the rest of the program.
- We must be able to say **unambiguously** what the function does.
- Repeated code: if we are tempted to copy and paste some code we should almost certainly cut and paste into a function instead.

When we use a function we must be sure to remember

- Local variables, including arguments, are precisely that. They are unrelated to variables of the same name in other functions or to variables used as parameters, except that the (local) arguments are initialised to the values of their arguments. After that it's all local!
- `return expression` returns that value of that expression to the calling function.

It's just like the functions we have already used

When we think of the following statement:

```
y = sin(x);
```

we know exactly what it will do: it will set `y` to the value of `sin(x)`. We don't expect it to change the value of `x` and we don't expect the value of another variable in our function to change because it happened to have the same name as a variable inside the `sin()` function. **Our own functions behave in exactly the same way.**

And finally: we don't use copy and paste for more than one or two lines

Instead use **cut** and paste and create a function (or possibly, as we shall see in a later lecture, a loop)

After the lecture

1. Review the key points.
2. Look at the complete example, seeing how these apply.