

Exercises for 01-function

functions

1.

Write a program that invokes the function `clock`. Compile the program. Try to interpret the message from the compiler. Did the compiler produce a binary/program for you?

Note: what the function `clock` does is left as an exercise for the reader. . . . gahh, don't we all hate it when a book says so?

2.

Add the following “flag” or “option” to the compile statement: `-Wall`. The command should now look something like:

```
gcc -Wall invoke1.c - assuming your file is called invoke1.c
```

Did the code compile properly?

3.

Add the following “flag” or “option” to the compile statement: `-Werror`. The command should now look something like:

```
gcc -Werror invoke1.c - assuming your file is called invoke1.c
```

Did the code compile properly?

4.

Add the following code at the top of the file you wrote in (1).

```
#include <time.h>
```

Compile without any flags/options. , typically like this:

```
gcc invoke1.c - assuming your file is called invoke1.c
```

Did the code compile properly?

5.

Together with these exercises you can find a file called `invoke.c`. Open up that file in an editor and add a `main` function at the end - you'll find a note in the file where to add the `main` function.

Make sure that the `main` method invokes the `hello` function.

6.

Change the code in (5) to invoke the `hello` function 10 times.

7.

Imagine you had to change the code in (5) to invoke the `hello` function 100 times.

Note: Wouldn't it be cool if there was some kind of construct that made this easy... and there is. More on this later on in the course :)

8.

Imagine you have a friend that is really great at shouting out names really loud. Assume you were to ask your friend to shout out a name. You ask your friend to do it... and you find your friend staring back at you thinking you're a fool. Why? Because you haven't told your friend what name to shout out.

Let's imagine you now ask your friend to shout out the name "Kalle" and your friend does that. Excellent, it worked! This time you supplied some extra information (e.g. the name "Kalle") to your friend. This extra information is called arguments.

Now, finally, it is time to hack (write clever code). Write a program that prints the name "Kalle" on the screen. That is invoke `printf` with some extra information ("Kalle") or put another way, invoke `printf` with "Kalle" as argument.

Note: actually we want you to 'print' on something called stdout, but that's another story

9.

Together with these exercises you can find a file called `invoke-return.c`. Open up that file in an editor and add a main function at the end - you'll find a note in the file where to add the main function.

Make sure that the main method invokes the `gimme_three` function.

Note: nothing cool should happen. Sorry!

10.

The function `gimme_three` is returning something. What is the type of the return value? How did you tell?

11.

Since the `gimme_three` function returns something, we can store the returned value, typically in a variable. Do that and print out the value of the variable.

What is a good type for the variable?

12.

Together with these exercises you can find a file called `invoke-parameters.c`. Open up that file in an editor and add a main function at the end - you'll find a note in the file where to add the main function.

The main method should:

- invoke the `add` function with two integer values as arguments
- store the returned value in a variable with an appropriate type
- print the resulting value using `printf`

13.

Analyse the function `add` in the previous exercise. What does the function do?

14.

Add to the file in (12) a function, `mult`, that takes two parameters and returns the product (think multiplication) of the parameters.

Note: the function `mult` shall, and need not, print something at all

15.

The main method, from (12), should:

- invoke the `mult` function with two integer values as arguments
- store the returned value in a variable with an appropriate type
- print the resulting value using `printf`

16. (optional)

What do you think happens if you add the following code, in the file used in (15).

```
int result;  
  
result = add(add(12, 34), add(34,56));
```

Note: try it out and reflect over the result

17.

Together with these exercises you can find a file called `invoke-proto.c`.

Compile the file. Gcc warns about `implicit declaration of function 'mult'`.

This means gcc that gcc doesn't know how `mult` works. We could solve this in several ways. We give two examples here:

- move the function `mult` before the `mult_and_print` function. This is a rather bad solution since it does not scale well.
- write a prototype of the function after the include statement. You simply add the following `int mult(int a, int b);`

Out solution, the latter of the two above, is telling the compiler (gcc) that there is a function `int mult(int a, int b);` which is defined somewhere else. Gcc is totally happy with this - as long as there actually is such a function somewhere.

18.

Rename the `mult` function to `tumult`. Compile again. You'll get an error from gcc (actually it is the linker - a part of the compiler) who refuses to produce a program since the call to `mult` cannot be done since there is no such function.