Labs: how do you feel?

- Too easy
- Ok
- Way too difficult
- Really not sure about that is going on
Labs: Learning & Learning from Examples

• Understanding and being able to ask questions is key point!

• We learn the most by building things ourselves!

• ..but we can also learn from examples (at least if they are not too advanced)

• Look at all the code that is given to you in the labs and think about how to re-use it
  – Today we will talk about functions, they are a good way to re-use code!
Textbook

- The material covered in this lecture is in chapter 6
- Please go through chapters 6 and 7 before the lab
Credits + feedback

- The quizzes in these slides are from http://www.cprogramming.com/

- The rest of the material is original, so feedback is welcome!
Functions

- Functions are used to make your programs modular
- Modular code is good!
- It is easier to understand (and therefore to debug)
- It can be re-used
- Frequently used functions are worth collecting into libraries so that they can easily be used in many programs (e.g. printf, round...)
Defining functions

• This is the anatomy of a function:

<return type> <function name>(<arg. 1>, <arg. 2>, ..)
{
    <statements>;
    return <return value>;
}

• For example:
int mult(int x, int y)
{
    int result = x * y;
    return result;
}
Defining functions (cont.)

- or
  
  ```
  int mult(int x, int y)
  {
    return x * y;
  }
  ```

- Arguments are very similar to variable declarations:
  type + name

- Argument values are *copied* into the function variables (more later)
Defining functions (cont.)

- If we have no arguments we need to use the keyword `void`:
  ```c
  int say_hello(void)
  {
    printf("hello\n");
    return 0;
  }
  ```

- The `void` keyword can also be used to say "no return value"
  ```c
  void say_hello(void)
  {
    printf("hello\n");
    return;
  }
  ```
Calling functions

- This should already be **very** familiar

- You can pass information either through variables or “values” (constants) -- e.g.:

  - `printf("%d\n", 33);
    or..
    int v = 33;
    printf("%d\n", v);
  
  - `mult(2, 3);
    or
    int v = 2;
    mult(v, 3);`
Recursive function

- Functions can call other functions
- They can also call themselves
- `void countdown(int counter)`
  ```c
  { printf("%d\n", counter);
    if (counter > 0) {
      countdown(counter-1);
    }
  return;
  }
  ```
Declaring functions

• You need to tell the compiler about your functions before calling them

• Sometimes defining functions before calling them is not convenient (e.g. recursive functions)

• If you want to call a function before it’s defined you use a prototype before the call:

  <return type> <function name>(<arg. 1>, <arg. 2>, ..);

• That’s the same as the initial part of the function definition PLUS a semicolon!
#include <stdio.h>

int mult(int a, int b)
{
    return a * b;
}

int main(void)
{
    int m;
    m = mult(3, 2);
    printf("%d\n", m);
    return m;
}
#include <stdio.h>

int main(void)
{
    int m;
    m = mult(3, 2);
    printf("%d\n", m);
    return m;
}

int mult(int a, int b)
{
    return a * b;
}
Declaring functions: example (OK)

```
#include <stdio.h>

int mult(int a, int b);

int main(void)
{
    int m;
    m = mult(3, 2);
    printf("%d\n", m);
    return m;
}

int mult(int a, int b)
{
    return a * b;
}
```
Quiz to check your understanding

Which is **not** a proper prototype?

- A. `int funct(char x, char y);`
- B. `double funct(char x)`
- C. `void funct();`
- D. `char x();`
Quiz to check your understanding

What is the return type of the function with prototype: 
"int func(char x, float v, double t);"

- A. char
- B. int
- C. float
- D. double
Quiz to check your understanding

Which of the following is a valid function call (assuming the function exists)?

- A. funct;
- B. funct x, y;
- C. funct();
- D. int funct();
A function to print arrays

- When we work with arrays we often need to print their content
- You should know that you can do it this way:

```c
for (i=0; i<size; i++)
{
    printf("%d,", arr[i]);
}
printf("\n");
```
- Where `arr` is an `int` array with `size` elements
  ..what about `i`?
A function to print arrays (cont.)

• We can put this code into a function

• What does the function need to know in order to print the array content?
A function to print arrays (cont.)

- We can put this code into a function
- What does the function need to know in order to print the array content?
  - The array itself
  - The array size
A function to print arrays (cont.)

• We can put this code into a function

• What does the function need to know in order to print the array content?
  – The array itself
  – The array size

These are the arguments we need then!
A function to print arrays (cont.)

- We can put this code into a function
- What does the function need to know in order to print the array content?
  - The array itself
  - The array size
- Do we need to return any value?
A function to print arrays (cont.)

• We can put this code into a function

• What does the function need to know in order to print the array content?
  – The array itself
  – The array size

• Do we need to return any value?
  – No (or not necessarily)
A function to print arrays (cont.)

• We can put this code into a function

• What does the function need to know in order to print the array content?
  – The array itself
  – The array size

• Do we need to return any value?
  – No (or not necessarily)

• `void print_array(const int size, int a[]);`
A function to print arrays (cont.)

```c
void print_array(const int size, int a[])
{
    int i;
    for (i=0; i<size; i++)
    {
        printf("%d,", a[i]);
    }
    printf("\n");
    return;
}
```
A function to print arrays (cont.)

```c
void print_array(const int size, int a[])
{
    int i;
    for (i=0; i<size; i++)
    {
        printf("%d,", a[i]);
    }
    printf("\n");
    return;
}
```

**Important!**
This name is independent from the name of the array in the calling function!
Argument names

• For example this call:

```c
int array[5] = {1,2,3,4,5};
print_array(5, array);
```

• ..is good, even though the array name here is "array" and in the function it is "a"
Variables scope

• When you declare a variable inside a function it can be accessed only inside that function
  – Did you notice main is a function too?
• A variable declared in a function is named a **local variable**
• That’s very useful to avoid confusion, e.g.:

```c
int sum(int a, int b)
{
    int result = a + b;
    return result;
}

int mult(int a, int b)
{
    int result = a * b;
    return result;
}
```
Variables scope (cont.)

- If you declare a variable outside any function it is accessible by all functions, and it is named *global*

- Using global variables is important in some cases, but it can lead to mistakes: use only when strictly necessary!
  
  - As much as you can, pass and return values instead of using global
Automatic vs. Static Variables

• By default local variables are automatic (keyword `auto`)

• Variables can be explicitly defined as static (that’s a keyword):
  
  – `static int v = 10;`

• Static variables keep their state across multiple function calls
Automatic vs. Static Variables (cont.)

• Example with static:
  void func(void){
    static int v = 10;
    printf("%d\n", v);
    v = v * v;
    return;
  }
  prints 10, then 100, then 10000, ...
Automatic vs. Static Variables (cont.)

- Example without static:
  
  ```c
  void func(void){
    int v = 10;
    printf("%d\n", v);
    v = v * v;
    return;
  }
  ```

  prints 10, then 10, then 10, ...
Arguments and scope

- Arguments define local variables
- When the function is called this local variables are initialized with the values passed to the function

```c
int mult(int a, int b)
{
    int result = a * b;
    return result;
}
```

```c
int v = 2;
mult(v, 3);
```

```
int a = 2;
int b = 3;
```
Arguments and scope (cont.)

- Changing the arguments does not affect the callee:

```c
int mult(int a, int b)
{
    int result = a * b;
    a = 100;
    return result;
}
```

...  

```c
int v = 2;
mult(v, 3);
/* here v is still 2 */
```

- There is a way around this..
Quiz to check your understanding

- This code will print..
  - A. 10
  - B. 3
  - C. 1
  - D. None of these

```c
int sum(int a, int b) {
    int result = a % b;
    a = 100;
    return result;
}

... 
int a = 10;
int b = sum(a, 3);
printf("%d", b);
```
Quiz to check your understanding

• This code will print..
  
  – A. 100
  – B. 10
  – C. 1
  – D. None of these

```c
int sum(int a, int b) {
    int result = a % b;
    a = 100;
    return result;
}

... 
int a = 10;
int b = sum(a, 3);
printf("%d", a);
```
Quiz to check your understanding

• This code will print..
  – A. 1
  – B. 100
  – C. 3
  – D. None of these

```c
int sum(int a, int b)
{
    int result = a % b;
    a = 100;
    return result;
}
...
int a = 10;
int b = sum(a, 3);
printf("%d", result);
```
Summary

• Functions can and should be used to make code modular
  – Modular code is reusable and more clear

• Functions can take arguments (zero or more)

• Functions can return one value (or no value)

• Normally arguments are copied into the functions: modifying the argument values inside the function does not affect the caller