Maciej Sobieraj

Lecture 8

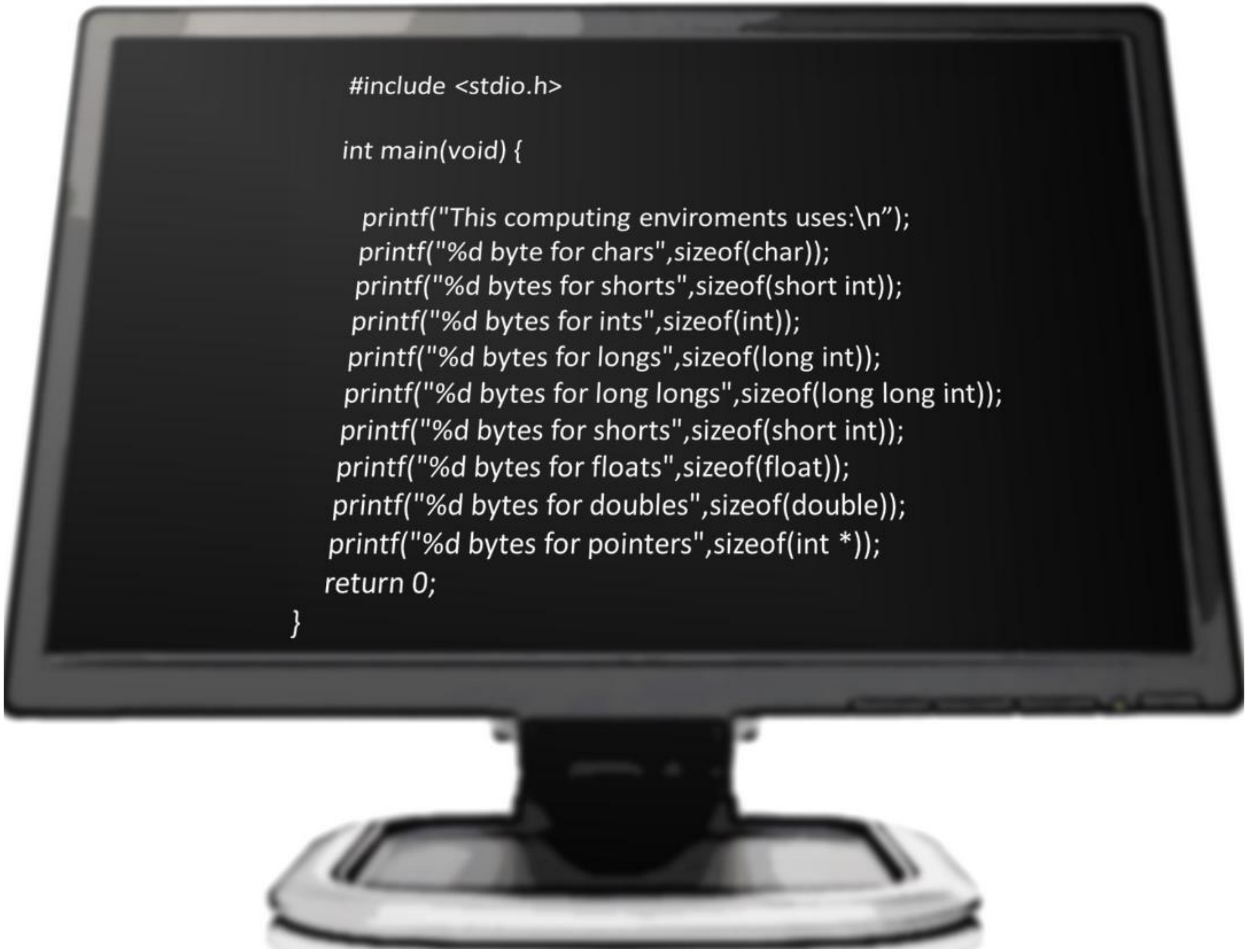
Outline

1. Functions
   1. **Functions: why do we need them?**
   2. Our first function
   3. Variables, parameters and results
   4. Scalars as function parameters
2. Quiz

Why would we want to write functions?

- **Reason #1**
  - It often happens that a particular piece of code is **repeated** many times in your program. It's repeated either literally or only with some minor modifications consisting of the use of other variables in the same algorithm.

Why would we want to write functions?



Why would we want to write functions?

- **Reason #2**
  - It may happen that the algorithm you're going to implement is so complex that the *main* function begins to grow in an uncontrolled manner, and suddenly you notice that you're having problems simply navigating through it.

Why would we want to write functions?

- **Reason #3**
  - It often happens that the problem is so large and complex that it cannot be assigned to a single developer, and a team of developers have to work on it. The problem has to be split between several developers in a way that ensures their efficient and seamless cooperation.

Outline

1. Functions
   1. Functions: why do we need them?
   2. **Our first function**
   3. Variables, parameters and results
   4. Scalars as function parameters
2. Quiz

What does the compiler need

- If the compiler is analyzing your program and encounters something that looks like a function invocation, it'll try to make sure that:
  - the function you want to call is **available**;
  - the parameters you've specified (or haven't specified at all) are **consistent** with what is expected for the function;
  - the return type of the function is **compatible** with the type of targeting l-value

```
NumberOfSheep = CountSheep();
```

What does the compiler need

- The compiler must have the following information for each function you’re going to use:
  - what is the name of the function?
  - how many parameters does the function expect and of which types?
  - what is the function's return type?
- The compiler can derive information about the functions from two sources:
  - the **declaration** of the function
  - the **definition** of the function.

Declaration vs. definition

- The **declaration** of a function is the part of the code containing all three key pieces of information (**name, parameters, type**), but doesn't contain the **body** of the function.
- A **definition** of a function is a part of the code containing its **full implementation** (including the body).

```
int CountSheep(void);           /* declaration */

int CountSheep(void) {          /* definition */
   return ++SheepCounter;
}
```

Our first function

```
void hello(void) {
    printf ("You've invoked me – what fun!\n");
    return;
}
```

- The declaration of this function would be as follows:
  - **void** hello(**void**);
- We can invoke our new function in the following way:

```
hello();
```

How do we not invoke our function?

- We mustn't invoke our function in the following way:

```
int i = hello();
```

How do we not invoke our function?

- Invoking it like this is prohibited, too:

~~`hello(2);`~~

Function eventually invoked

- Here's a complete program, ready to compile and run, including both the function definition and its invocation

```
#include <stdio.h>

void hello(void) {
     printf ("You've invoked me – what fun!\n");
     return;
}

int main(void) {
    printf("We are about to invoke hello()!\n");
    hello();
    printf("We returned from hello()!\n");
    return 0;
}
```

Function eventually invoked

- We had to change our code – it now looks like this:

```
#include <stdio.h>
int main(void) {
        printf("We are about to invoke hello()!\n");
        hello();
        printf("We returned from hello()!\n");
        return 0;
}
void hello(void) {
         printf ("You've invoked me – what fun!\n");
         return;
}
```

Function eventually invoked

- The compiler is forced to guess all the traits of the *hello* function before the compiler even reads its declaration or definition. You should expect the compiler to generate a warning message and the implicit declaration will perform its deduction.
- The deduction is very simple – it assumes that all **entities of unknown types are *ints***. This means that the compiler is convinced that the actual *hello* declaration looks as follows:
  - **int** hello(**void**);

Function eventually invoked

- We should warn the compiler that the function will be used and provide complete information about it.

```c
#include <stdio.h>

void hello(void);

int main(void) {
   printf("We are about to invoke hello()!\n");
   hello();
   printf("We returned from hello()!\n");
   return 0;
}

void hello(void) {
    printf ("You've invoked me – what fun!\n");
    return;
}
```

return statement

- The *return* statement executed inside any function **causes immediate function termination** and a return to the invoker.
- If the function is defined as void, then:
  - the acceptable return statement looks like

```
return;
```

  - if the body doesn't contain a return statement, it will be implicitly added after the last instruction of the function's block.

return statement

- This means that you can write the *hello* function in the following way too:

```
void hello(void) {
        printf ("You've invoked me – what fun!\n");
}
```

- Note that more than one return statement may exist in the function body.

return statement

- If the function type isn't specified as *void*, the only acceptable form of return statement is as follows

```
return expression;
```

  - where the expression **must provide** the value of the type matching the type of function; in this case using the *return* statement is mandatory and you cannot omit it in the function body.

Outline

1. Functions
   1. Functions: why do we need them?
   2. Our first function
   3. **Variables, parameters and results**
   4. Scalars as function parameters
2. Quiz

Functions and their local variables

- Function blocks and blocks in general can contain variable declarations – as many as you need.
- If we declare a variable inside a block (e.g. a function's block) the variable will be known and recognized **only inside that block** and, consequently, will not be known in any other part of the program.
- The name will not interfere with other variabl with identical names defined inside other blo

Functions and their local variables

```
#include <stdio.h>

void hello(void) {
     int i;

     for(i = 0; i < 2; i++)
         printf ("You've invoked me – what fun!\n");
     return;
}
int main(void) {
    int i;
    printf("We are about to invoke hello()!\n");
    for(i = 0; i < 3; i++)
        hello();
    printf("We returned from hello()!\n");
    return 0;
}
```

Global variables

- If the variable is declared outside of all the blocks, it becomes a **global** variable.
- A global variable is accessible to **all functions in a source file**.

Global variables

```
#include <stdio.h>

int global;

void fun(void) {
   int local;

   local = 2;
   global++;
   printf("fun: local=%d global=%d\n", local, global);
   global++;
}

int main(void) {
   int local;

   local = 1;
   global = 1;
   printf("main: local=%d global=%d\n", local, global);
   fun();
   printf("main: local=%d global=%d\n", local, global);
   return 0;
}
```

Global variables

- We can expect the following text to be sent to the screen:

```
main: local=1 global=1
fun: local=2 global=2
main: local=1 global=3
```

Function parameters

- The **function parameter** is a special kind of local variable. It behaves like a local variable – its name isn't known outside the function.
- It differs from the local variable in two important features:
  - first, the parameter is **not declared within the function**, but must be declared inside a pair of parentheses after the function name (which means that the parameter declaration is a part of the function declaration);

```
void hello2(int times);
```

Function parameters

```
void hello2(int times);
```

- The *times* variable may be used inside the function in exactly the same way as if it were a local variable; this is called a ***formal parameter***.
  - second, a prototype of the function containing formal parameters forces us to invoke that function with a **list of expressions**, and the number of expressions must be **equal** to the number of formal parameters in the prototype;

Function parameters

```
void hello2(int times);
```

- the types of these expressions must be **compatible** with the types of the corresponding formal parameters; each of these expressions is called an ***actual parameter;***
- at the beginning of the invocation every formal parameter is **assigned the value** of the corresponding actual parameter.

Function parameters

- It clearly shows that these three invocations are valid (they all deliver a value of type *int* to the formal parameter)

```
int notmany = 5;
hello2(100);             /* actual parameter is a literal */
hello2(notmany);         /* actual parameter is a variable */
hello2(2 * notmany); /* actual parameter is an expression */
```

Function parameters

- We must not call the function *hello2* in any of the following ways

```
hello2();         /* too few actual parameters */
hello2(1,2);      /* too many actual parameters */
hello2("Hey");    /* incompatible actual parameter */
```

Function parameters

- Let’s assume that the *hello3* function has the following declaration:
  - **void** hello3(**int** i, **float** f);
- and has been invoked as follows:
  - hello3(100, 3.14);
- The following assignments will be performed implicitly and beyond our control:
  - i = 100;

Function parameters

- The first format parameter is assigned with the current value of the first actual parameter;
  - f = 3.14;
- The second formal parameter is assigned with the current value of the second actual parameter.
- The parameterized function may modify its own behavior according to the parameter's value.

Function parameters

- The updated *hello2* function body goes here:

- If you invoke this function as follows:
  - hello2(100);
- the following assignment will take place automatically:
  - times = 100;
- This causes the function to manifest its joy 100 times.

Function parameters

```
void hello2(int times) {
    int i;
    for(i = 0; i < times; i++)
        printf ("You've invoked me – what fun!\n");
    return;
}
```

Function results

- If the function has been declared with a type before its name, it must perform the **return** statement equipped with an expression.

Function results

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

int main(void) {
    float a, b, a_sqr, b_sqr, c;

    printf("A?\n");
    scanf("%f", &a);
    a_sqr = a * a;
    printf("B?\n");
    scanf("%f", &b);
    b_sqr = b * b;
    c = sqrt(a_sqr +  b_sqr);
    printf("The length of the hypotenuse is: %f\n", c);
    return 0;
}
```

Function results

- The function won’t be particularly advanced – we expect it to:
  - accept one parameter of type *float*;
  - square the value of the parameter and return it as the result.
  - the result type is **float** (can you explain why?)
  - we’ll name our function ***square*** – it's good practice to name functions using verbs

Function results

```
float square(float param) {
    float x_sqr;

    x_sqr = param * param;
    return x_sqr;
}
```

Function results

- Of course, the function could be slightly simplified:

```
float square(float param) {
    return param * param;
}
```

Function results

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

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

int main(void) {
   float a, b, a_sqr, b_sqr, c;

   printf("A?\n");
   scanf("%f", &a);
   a_sqr = square(a);
   printf("B?\n");
   scanf("%f", &b);
   b_sqr = square(b);
   c = sqrt(a_sqr +  b_sqr);
   printf("The length of the hypotenuse is: %f\n", c);
   return 0;
}
```

Function results

- Note that *a_sqr*, *b_sqr* and *c* are used as **temporary containers** only. We can remove them.

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

float square(float param) {
   return = param * param;
}

int main(void) {
   float a, b;

   printf("A?\n");
   scanf("%f", &a);
   printf("B?\n");
   scanf("%f", &b);
   printf("The length of the hypotenuse is: %f\n",sqrt(square(a) + square(b)));
   return 0;
}
```

Outline

1. Functions
   1. Functions: why do we need them?
   2. Our first function
   3. Variables, parameters and results
   4. **Scalars as function parameters**
2. Quiz

Function parameters that are scalars

```
#include <stdio.h>

void function(int param) {
    printf("I've received value %d\n", param);
}

int main(void) {
    int variable = 111;

    function(1000);
    function(variable);
    function(variable + 1000);
    return 0;
}
```

Function parameters that are scalars

- We already know that actual parameter values are **assigned to formal parameters** during function invocation, which means that executing the program will send the following text to the screen:

  I've received value 1000
  I've received value 111
  I've received value 1111

Function parameters that are scalars

- What happens if the function changes the value of the formal parameter? It's always possible.

```
#include <stdio.h>

void function(int param) {
   printf("I've received value %d\n", param);
   param++;
}

int main(void) {
   int variable = 111;

   function(variable);
   printf("variable %d\m", variable);
   return 0;
}
```

Function parameters that are scalars

- The program will emit the following text:

```
I've received value 111
variable=111
```

Function parameters that are scalars

- The actual parameter's value is **copied** into the formal parameter and the relationship between them **definitely ends** at this point.
- Any change of the formal parameter **does not affect** the state of the actual parameter.
- This kind of cooperation between parameters is known as a **by-value parameter passing.**

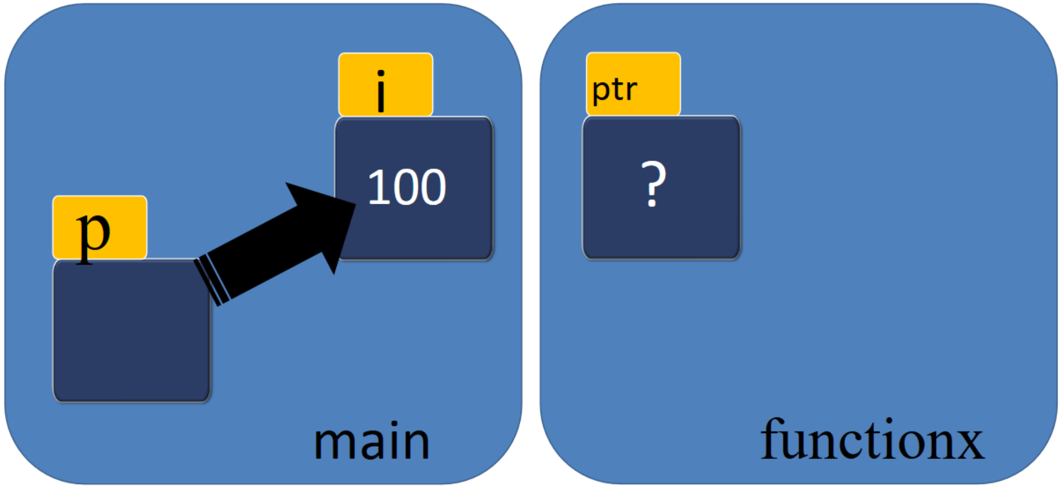
Function parameters that are scalars

- We can deal with this limitation by using **pointers**. This is how the *scanf* function works when it has to get the value from the user and assign it to a variable.

```
#include <stdio.h>

void functionx(int *ptr) {
   *ptr = *ptr + 100;
}

int main(void) {
   int i = 100;
   int *p = &i;

   printf("i = %d\n", i);
   functionx(p);
   printf("i = %d\n", i);
   return 0;
}
```

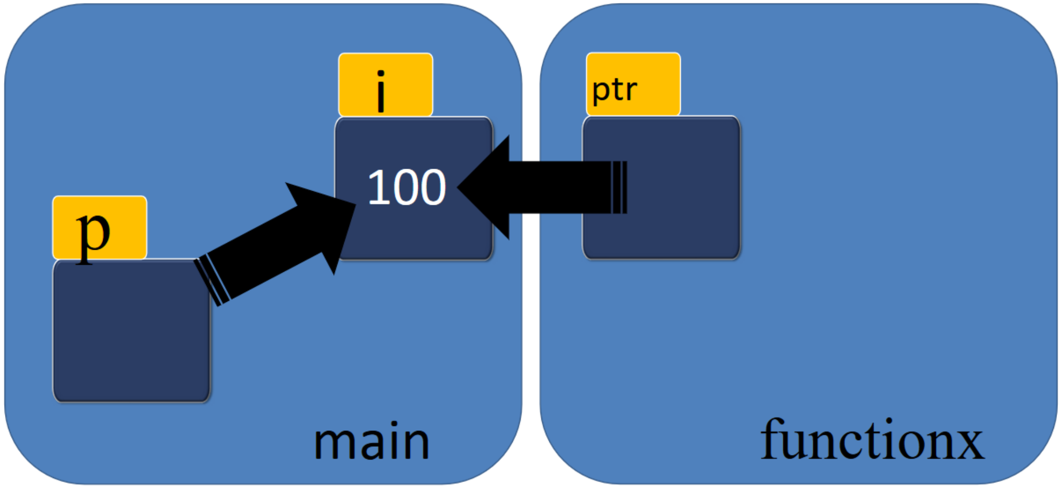
Function parameters that are scalars

- Let's navigate through our program.
  - we declare the *i* variable and initialize it with a value of 100
  - we declare the *p* variable and initialize it with a pointer to the *i* variable



Function parameters that are scalars

- Let's navigate through our program.
  - we invoke *functionx* and pass the value of *p* to it;
  - the value is copied to the *ptr* parameter



Function parameters that are scalars

- Let's navigate through our program.
  - *functionx* exits
  - the *p* variable hasn't changed its value
  - the *i* variable has changed its value and it’s now 200
  - the output screen looks as follows

i = 100

i = 200

Function parameters that are scalars

- We can simplify the main function in the following way.
- The *p* pointer is completely unnecessary and we can remove it. A pointer to *i* can be retrieved using the & operator

```
int main(void) {
    int i = 100;

    printf("i = %d\n", i);
    functionx(&i);
    printf("i = %d\n", i);
    return 0;
}
```

Function parameters that are scalars

- We're going to write a function which increases the value pointed to by its parameter.

```
void incr(int *value) {
    (*value)++;
}
```

- Now we're ready to make use of our function:

```
int main(void) {
        int var = 100;
        incr(&var);
        printf("var = %d\n", var);
        return 0;
}
```

Outline

1. Functions
   1. Functions: why do we need them?
   2. Our first function
   3. Variables, parameters and results
   4. Scalars as function parameters
2. **Quiz**

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int add(int par) {
   par += par;
   return par;
}
int add2(int p1, int p2) {
   return p1 + p2;
}
int main(void) {
   int var = 0;
   var = add2(2,4);
   var = add(var);
   var = add2(var,var);
   printf("%d",var);
   return 0;
}
```

- the program outputs 24
- the program outputs 48
- the program outputs 72

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int add(int par) {
    par += par;
    return par;
}
int add2(int p1, int p2) {
    return p1 + p2;
}
int main(void) {
    int var = 0;
    var = add2(add(2),add(4));
    var = add2(var,var);
    printf("%d",var);
    return 0;
}
```

- the program outputs 12
- the program outputs 24
- the program outputs 48

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int f1(int v) {
   v *= v;
   return v;
}
int f2(int p1, int p2) {
   return p1 / p2;
}
int main(void) {
   int v = 0;
   f1(f1(f2(2,4)));
   printf("%d",v);
   return 0;
}
```

- the program outputs 6
- the program outputs 0
- the program outputs 8

Quiz

What happens if you try to compile and run this program?

```
#include <stdio.h>
int fun(int n) {
   return n - 1;
}
int main(void) {
   printf("%d",fun(fun(fun(fun(fun(3))))));
   return 0;
}
```

- the program outputs -2
- the program outputs 0
- the program outputs -1