C programming

Lecture 7 : functions

Based on slides designed by Shay Artsi, Aythan Avior, Gitit Rockstein and Saher Ismir for the department of Computer Science, Technion, Israel Institute of Technology
Translated and updated by Anne Weill-Zrahia
The large code problem

- If a program performs many operations, it will soon become **hard to manage** in the following aspects:
  - Writing and debugging
  - Understanding code
  - Maintaining and updating the code

- There will be many **redundancies** in the code: blocks of code which perform similar tasks will be repeated in the code
Solution : functions

- Code will be divided into several parts (in C : functions)
- The first function which is performed is `main()`
- Every code has at least one function
- To activate a function is defined as a *function call*
- A function can call to every function in the program
- Dividing the code into functions permits:
  - To avoid code duplication
  - To improve readability
  - To share code parts and software package
A taste of functions ... so to speak

- Preparing spaghetti Bolognese

```c
void spaghetti_bolognese() {
    prepare_spaghetti();
    prepare_sauce();
    stir;
    serve;
}
```

```c
void prepare_spaghetti() {
    boil_water;
    add_spaghetti;
    while (not_ready) {
        stir;
        wait(2);
    }
}
```
Less tasty : find the GCD of 3 numbers

```c
#include <stdio.h>

int gcd(int n, int m) {
    int t;
    while ( m != 0 ) {
        t = m;
        m = n % m;
        n = t;
    }
    return n;
}

int main() {
    int a, b, c, gcd_ab, gcd_abc;
    scanf("%d%d%d", &a, &b, &c);
    gcd_ab = gcd(a, b);
    gcd_abc = gcd(gcd_ab, c);
    printf("gcd: %d",gcd_abc);
    return 0;
}
```
Definition of a function

\[
\text{return-type } \text{ function-name(} \text{parameters-list}) \text{ }
\{
\begin{align*}
\text{variable definitions} \\
\text{statements} \\
\text{return statement}
\end{align*}
\}
\]

- **void** is a keyword whose meaning can be:
  - Function which does not return a value
  - Declaration of function which has no parameters

```c
void print_result(int)

int getchar(void);
```
Function call

The following actions are performed when a function is called:

1. Memory cells are allocated in order to contain the formal parameters which are contained in the function definition.

2. Computation of expression in the argument’s list (Actual parameters) (beware: the order of computation of arguments’ values is not defined)

3. The value of each expression is assigned to a variable.

4. Allocation of space for internal variables (defined in the function’s body) is performed

5. Execution of code
1. A value can be returned to the calling environment through a `return`.

2. Two kinds of `return`:
   - `return value;`  
     ```c
     int func()
     ```
   - `return;`  
     ```c
     void func()
     ```

3. When control gets to the `return` or to the end of the function the memory which was allocated for the function variables (parameters and local variable) is deallocated. and control returns to the statement following the function call.

4. There may be an automatic casting of the return value.
Example – function call and return—

```c
int gcd(int n, int m) {
    ...  
    return n;
}

int foo() {
    ...  
    d = gcd(2, 2);  
    return gcd(1, d);  
}
```

```plaintext
n
1

m

0

d

2
```
Declaration of function

Before calling a function the file should contain a declaration and a definition of the function.

A declaration informs the compiler on number and type of parameters, return type, and permits to do the necessary cast if needed.

The `#include` statement allows to import declarations of library function from a .h file.

We bring declarations of the functions `scanf` and `printf` from the file `stdio.h`.

double sqrt(double);
long power(int, int);
long power(int base, int exponent);
Declaration of functions

- We could define functions before the main, but this would contradict top-down-design.

We prefer the code inside the main function to appear first.

- We will have to use a declaration when a function is in the chain of recursive calls:
  If the function calls itself or another function which calls it back
Example – absolute value

double abs(double x) {
    if    ( x < 0 )
    return -x;

    return x;
}

Don’t we need else
Example 1 – syntax in a whole program

```c
#include <stdio.h>

double abs(double x);

int main()
{
    double y;

    printf("y = ");
    scanf("%lf", &y);
    printf("|%f| = %f\n", y, abs(y));

    return 0;
}

double abs(double x)
{
    if ( x < 0 )
        return -x;
    return x;
}
```

File start

File end
Example 2 – finding minimum

The following function gets 2 parameters and returns the minimum between them

```c
double min2(double a, double b) {
    if (a < b)
        return a;
    return b;
}
```

What will the function return if the parameters are equal?

How will we define a function max2
Example 3: Minimum between 3 values

define minimum between three values

define min3(double a, double b, double c) {
    if ( (a <= b) && (a <= c) )
        return a;
    if ( (b <= a) && (b <= c) )
        return b;
    return c;
}

- How can we simplify this function?
Example 3: better code

- As was said, using functions will simplify code
- In this case, we can use the function min2 which we defined before

```c
double min3(double a, double b, double c) {
    double minAB = min2(a, b);
    return min2(minAB, c);
}
```

- and even

```c
double min3(double a, double b, double c) {
    return min2(min2(a,b), c);
}
```
Organization of the file

This is an example of top-down design

And here we have an example of bottom-up design

Less declarations, but less understandable.
Example 4: which number is this?

Goal: write a function which gets a number and checks if it is a prime, a perfect square or neither:

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

enum bool {FALSE, TRUE};
typedef enum bool Boolean;

Boolean  is_square(unsigned int);
Boolean  is_prime(unsigned int);
double   my_round(double);
double   my_trunc(double);
```
int main() {
    int num;

    while ( scanf("%d", &num) == 1 ) {
        if ( num < 0 ) {
            printf("%d is negative\n", num);
            continue;
        }

        if ( is_square(num) )
            printf("%d is a square\n", num);
        else if ( is_prime(num) )
            printf("%d is a prime\n", num);
        else
            printf("%d is neither a prime nor a square\n", num);
    }

    return 0;
}
```
Boolean is_prime(unsigned int n) {
    int i, sqrt_n;

    if    ( n == 2 )
        return  TRUE;
    if    ( n % 2 == 0 || n < 2 )
        return  FALSE;

    sqrt_n = my_round(sqrt(n)); /* is (int)sqrt(n) good too? */

    for   ( i = 3;  i <= sqrt_n;  i += 2 )
        if ( n % i == 0 )
            return  FALSE;

    return  TRUE;
}
```
Functions to check squareness, truncate and round

```c
Boolean is_square(unsigned int n) {
    int sq = my_round(sqrt(n));
    return (sq * sq == n);
}

double my_round(double frac) {
    return my_trunc(frac + frac < 0 ? -0.5 : 0.5);
}

double my_trunc(double frac) {
    return frac < 0 ? ceil(frac) : floor(frac);
}
```
Stack of a program

In computer science, a **stack** is a data structure in which data are inserted and removed in a **LIFO** - Last In First Out order.

A **program stack** or **calling stack** is a structure (implemented in hardware and the compiler) where information regarding executing functions at a given time is kept.

When a function is called, a **stack frame** (**activation frame**) is inserted onto the stack. It contains:
- Argument values at time of call
- Memory location for the return value (if needed)
- Address of the return of the function (after it is executed)
- Memory allocation for local variables.

After the function returns, the **activation frame** is erased from the stack, and control returns to the main program/calling function.

The **activation frame** of the executing function is always on top of the stack.
Calling stack of a program

1. main()
   num = 8

2. is_prime()
   n = 8
   i, sqrt_n

3. main()
   num = 8

4. is_prime()
   n = 8
   i, sqrt_n

5. main()
   num = 8

6. my_trunc()
   frac = 2.8

7. my_round()
   frac = 2.8

8. is_prime()
   n = 8
   i, sqrt_n

9. my_trunc()
   frac = 2.8

10. floor()

   my_round()
   frac = 2.8
Scope of variables

- Any C program is built of blocks. {[…]=
- **Local variables** can be defined at the beginning of any block
- A local variable can be accessed only in the block in which it is defined, or included (lower-level) block
- A variable from an external block exists also in an interior block unless another variable of the same name as been defined

- In case of F1 calling F2, variables present in F1 are not known to F2
- A variable defined outside a block is called an external or global variable
- A global variable is known in the whole code file, sometimes in other files of the code.
```c
#include <stdio.h>
int a = 1, b = 2, c = 3, d = 4;
void v()
{
    int a = 101, b = 102, e = 105;
    printf("%d %d %d %d %d\n", a, b, c, d, e);
    {
        int a = 201, c = 203, d = 204;
        printf("%d %d %d %d %d\n", a, b, c, d, e);
    }
    printf("%d %d %d %d %d\n", a, b, c, d, e);
} int main()
{
    int a = 11, b = 12, d = 14, m = 17;
    printf("%d %d %d %d %d\n", a, b, c, d, m);
    printf("%d %d %d %d %d\n", a, b, c, d, m);
    {
        int a = 21;
        printf("%d %d %d %d %d\n", a, b, c, d, m);
        {
            printf("%d %d %d %d %d\n", a, b, c, d, m);
        }
    }
    v();
    printf("%d %d %d %d %d\n", a, b, c, d, m);
    return 0;
}
1D array and functions

- Arrays and pointers behave similarly in a function call.
- Example:

```c
int main() {
    int a = 7;
    change_a (a);
    printf("%d\n",a);
    return 0;
}

change_a(int a) {
    a = 3;
    return;
}
```

- What will be printed on screen? 7
- The function gets a temporary copy of the variable which is changed, the value in main program is not
Arrays and functions

Here 3 will be printed. Changing the array will influence the change in the function.
Number of elements of an array

How to specify it when sending an array to a function:

Example: Write a function which receives an array and prints out the sum of its elements:

```c
int sum (int a[])
{
    int i, val = 0;
    for (i=0; i< ? ; i++)
    {
        val+= a[i];
    }
    return val;
}
```

```c
int main()
{
    int a[6] = {7,4,7,4,2,1};
    printf("%d\n", sum(a));
    return 0;
}
```

How will we know?
Answer: send the number of elements to the function

```c
int sum (int a[], int n) {
    int i, val = 0;
    for (i=0; i<n; i++)
        val += a[i];
    return val;
}

int main() {
    int a[6] = {7,4,7,4,2,1};
    printf("%d\n", sum(a,6));
    return 0;
}
```
Size of arrays in functions

There are 3 ways of sending the size of an array

1. Send its size explicitly
2. Define what is the last element
3. Define the size at compilation time (#define statement)

In this course option 1 will be for numeric arrays, option 2 will be used in strings, and option 3 will be used for 2D arrays.
2D arrays and functions

```c
int main()
{
    int a[1][1] = {7};
    change_a (a);
    printf("%d\n", a[0][0]);
    return 0;
}

change_a(int a[1][1])
{
    a[0] = 3;
    a[0] = 3;
    return;
}
```
#define N 3
#define M 2

int main()
{
    int a[N][M] = {{2,3},{3,4},{5,6}};
    printf("%d\n", sum(a));
    return 0;
}
Memory types

We have 4 types of memory from which to allocate space:

- **Static Memory** – A location allocated to variables for the whole program life.

- **Automatic Memory** - A space in the execution record (which resides in the calling stack) – allocated to the same variables or parameters for the whole life of the function.

- **Registers** – small memory units in the processor – we don’t use them in this course.

- **Dynamic memory** – A location in memory allocated through an explicit request of the programmer – it is released also through a request or end of program.
# Principal types

<table>
<thead>
<tr>
<th>Static variables</th>
<th>Automatic variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Global variable</strong></td>
<td></td>
</tr>
<tr>
<td><code>int p = 1;</code></td>
<td>Inside block</td>
</tr>
<tr>
<td><strong>Outside block</strong></td>
<td>Where</td>
</tr>
<tr>
<td>Whole program</td>
<td>Block</td>
</tr>
<tr>
<td>Program’s lifetime</td>
<td>Block</td>
</tr>
<tr>
<td>Constant</td>
<td>Any Expression</td>
</tr>
<tr>
<td>0</td>
<td>Constant</td>
</tr>
<tr>
<td>Before program start</td>
<td>Undefined</td>
</tr>
<tr>
<td></td>
<td>Block entry</td>
</tr>
<tr>
<td></td>
<td>Initialization</td>
</tr>
<tr>
<td></td>
<td>Default</td>
</tr>
<tr>
<td></td>
<td>Initialization point</td>
</tr>
</tbody>
</table>
Types of variables

- **Automatic variable**
  - If it is not initialized – it is undefined till it is given a value
  - If initialized – initialized every entry
  - Does not keep its value from call to call

- **Static local variable**
  - If not explicitly initialized – initialized to 0
  - Keeps its value from call to call

- **Global variable**
  - Can be defined several times
  - Every definition must correspond in type and initialization
  - Some of the definition can contain initializations

- **Some tips:**
  - Place variables inside blocks where they belong
  - Use parameters and return values rather than global variables
Example : local static variable

This code controls some hardware device with two states: ON and OFF

- Function which changes the light position:

```c
void toggle_light()
{
    static int light_status = OFF;

    if ( light_status == OFF ) {
        turn_light_on();
        light_status = ON;
    } else {
        turn_light_off();
        light_status = OFF;
    }
}
```
Example : global static variable

```c
static int light_status = OFF;
void turn_light_on()
{
    if ( light_status == ON ) return;
    light_status = ON;
    ... 
}
void turn_light_off()
    if ( light_status == OFF ) return;
    light_status = OFF;
    ... 
}
void toggle_light()
{
    if ( light_status == OFF )
        turn_light_on();
    else
        turn_light_off();
}
```