CH-230-A

Programming in C and C++

C/C++

Lecture 3

Dr. Kinga Lipskoch

Fall 2019

Predefined and User Defined Functions

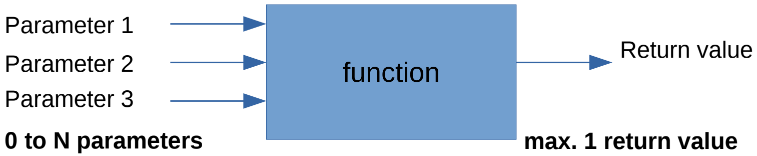
- Predefined functions are functions provided by the language or by the host
- Operating system
  - Library functions: they usually provide general purpose functionalities
- User defined functions are defined by the program
  - Usually targeted to the problem being solved

Functions: Motivation

- Writing a 50000 lines long main function can be really difficult
- Splitting the code into many small pieces has many advantages:
  - Easier to develop
  - Easier to maintain and debug
  - Increased opportunities to reuse the code
- An example: the `printf` function
  - Developed by specialists
  - Up to now we used it without knowing how it works internally
  - Should there be a bug in it, by just using an updated version you can fix your code at once

Some Analogies

- A function can be thought as a mathematical function
- A function can be thought as a black box performing some functionality



Functions in C

- **Function declaration** (prototyping)
- **Function call** (use)
- **Function definition**
- Call should be preceded by prototyping (ANSI C (American National Standards Institute) strongly advises this)
- There can be many declarations and many calls
- There must be exactly one definition

Prototyping

- The prototype is a statement declaring
  `return_type functionname(parameters);`
- Returned type is the type of the data
  - may be empty, default type is `int`
  - always declare the `return_type` explicitly
- Name follows the usual rules
- Parameters specify the number and types of the possible parameters
  - may be empty
  - always use explicit `void`, if function does not take arguments

The `void` Keyword

- `void` can be used to specify that
  - The function does not return any value
  - The function does not take any parameter
- `int unknown(void);`
  - function does not take any parameters
- `int unknown();`
  - function takes arbitrary number of parameters (to be compliant with the old Kernighan & Ritchie style)

Remember the Difference

- `void`
  - No return value
  - No parameter
- `void *`
  - Generic pointer (a pointer with no specific type which can be casted to any type)

Prototyping: Why?

- By having a prototype the compiler can check if the calls are performed correctly
  - Number of parameters, types, etc.
- It is now clear why prototypes should always appear before calls

Prototypes: Examples

- Prototypes of functions in `math.h`
  ```
  double sqrt(double x);
  double pow(double x, double y);
  ```
- User defined function prototypes
  ```
  int find_max(int v[], int dim);
  void print_menu(char *options[], int dim);
  void do_something(void);
  ```
- `void` specifies no return value and empty parameters list

Function Definition

- The function definition specifies what a functions does
- Function definitions can contain everything (variables definitions, cycles, branches, etc) but NOT other function definitions
- A function terminates when
  - it executes the last instruction
  - it encounters a return statement
- Definition starts with the function header
  `return type, name, parameters info`
- Braces to define where the function starts and ends
- Business statements (instructions for carrying out the function's task)

What Happens when a Function is Called?

- The given parameters are copied into the corresponding entry in the parameters list
- The control is transferred to the function
- When the called function terminates, the control goes back to the caller function

Comment your Functions

- Every function should be commented
  - Describe what the function does
  - Describe each parameter (type and meaning)
  - Describe what the function returns
- Look at the UNIX man pages to have an idea of how function documentation should look like

  ```
  man strcmp
  ```

Local Variables

- Variables can be declared inside any function
  - These are called local variables
  - Local variables are created when the function is called (e.g., the control is transferred to the function) and are destroyed when the function terminates
- Local variables do not retain their values between different calls

The Concept of Scope

- The scope of a name (function, variable, constant) is the part of the program where that name can be used
- The scope of a local variable is the function where it is defined
  - From the point of its definition
- Names having different scopes do not clash

Global Scope

- The scope of the names of functions goes from the prototype/definition to the end of file
- After their name is known they can be used, i.e., called
- It is possible to define global variables, i.e., variables outside function
  - Their scope is from the point of definition to the end of the file
  - After their definition is given they can be used, i.e., written and read

Local and Global Scope

```c
#include <stdio.h>

//global variable
int x = 7;

void xlocal(int y)  {
  int x;
  x = y * y;
  printf("xlocal: %d\n", x);
  return;
}

void xglobal(int y)  {
  x = y * x;
  printf("xglobal: %d\n", x);
  return;
}
```

```c
int main() {
  //int x;
  // try to explain if not
  // commented out
  x = 8;
  printf("main: %d\n", x);
  xlocal(x);
  printf("main: %d\n", x);
  xglobal(x);
  printf("main: %d\n", x);
  return 0;
}
```

Do not Misuse Global Variables

- Global variables can be used to communicate parameters between functions
- They can introduce subtle bugs in your code
- In general try to avoid them unless enormous advantages can be gained at a price of low risk
  - Document why you insert them
- Bigger projects will avoid using global variables

Parameters

- Function parameters are treated as local variables
- Local variables within functions and parameters must have different names
- Therefore the scope of a parameter is its function

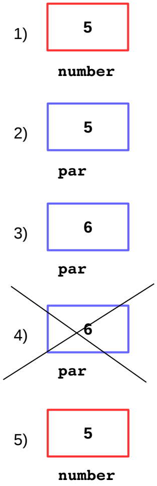
Parameters: by Value and by Reference

- **By value**: variables are copied to parameters
  - Changes made to parameters are not seen outside the function
- **By reference**: variables and parameters coincide
  - Changes made to parameters are seen outside the function
  - In C this is obtained by mean of pointers

Example: Passing by Value (1)

```c
#include <stdio.h>
void increase(int par) {
  par++;
}
/*  In this case no prototype:
    can you tell why? */
int main() {
  int number = 5;
  increase(number);
  printf("Increased number is %d\n", number);
  /* not as expected? */
  return 0;
}
```

Example: Passing by Value (2)



Parameters by Reference in C

- C passes only parameters by value
- For references it is necessary to provide a pointer to the variable
- In order to make a modification visible
- Outside it is necessary to use the dereference ($*$) operator

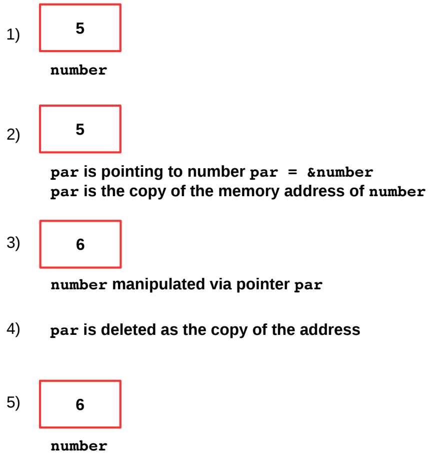
Example: Passing by Reference (1)

```c
#include <stdio.h>

void increase(int *par) {
  *par = *par + 1;
}

int main() {
  int number = 5;
  increase(&number); /* pass pointer */
  printf("Increased number is %d", number);
  return 0;
}
```

Example: Passing by Reference (1)



2) `par` is pointing to number `par = &number`
`par` is the copy of the memory address of `number`

3) `number` manipulated via pointer `par`

4) `par` is deleted as the copy of the address

Indentation Styles (1)

- Use spaces between operators: `a = b + 5;`
- Exception: `b++;`
- Do not use spaces if parentheses act as delimiter (functions)
  `printf("Number %d", b);`
- But use spaces before after `if`, `for`, `while`:
  `while (i <= 10)`
- Always put a space after comma
- Do not put a space before semicolon:
  `printf("Number %d", b);`

Indentation Styles (2)

- Put the opening brace either behind last word (including space) or put it on the next line
- Indent the block inside by tab or 4 (8) spaces
- The closing brace should be on the same column as the opening statement

```
for (i = 0; i < 10; i++) {    // K&R style
  printf("%d\n", i);
}
```

or

```
for (i = 0; i < 10; i++)    // Allman style
{
  printf("%d\n", i);
}
```

Strings

- A string is a sequence of characters
- Strings are often the main way used to communicate information to the user
- Many languages provide a string data type, but C does not
- In C strings are treated as arrays of characters
- `char my_string[30];`

C Strings

- A string is represented as a sequence of chars enclosed by double quotes
  - `"This is it"`
- String are stored in arrays of chars
  - An extra character is always added at the end to mark the end of the string
  - The extra character is the `'\0'` character i.e., the character whose ASCII code is 0

T	**h**	**i**	**s**		**i**	**s**		**i**	**t**	**\0**

fgets versus gets (1)

- `gets` does not check if you type more characters than allowed:
  ```
  char inputString[50];
  gets(inputString);
  ```
- `fgets` allows additional parameters:
  ```
  char line[50];
  fgets(line, sizeof(line), stdin);
  ```
  - Reads up to 49 characters from the input stream
  - The 50th one is used to store the null character '\0'

fgets versus gets (2)

- gets replaces the trailing '\n' with a '\0'
- fgets does not replace '\n', but it leaves it in the string
- Read the man pages for learning more on these functions
  - man gets
  - man fgets
- To make your life easier use fgets and convert to integer via sscanf
- Avoid using gets, it is unsafe

fgets and scanf together

- scanf and fgets do not work well together
- Your code should look like this, if you use both

```
scanf("%d", &number);
getchar();
...
fgets(line, sizeof(line), stdin);
sscanf(line, "%d", &number);
```

String Functions

- Defined in `string.h`

`strlen`	Determines the length of a string
`strcat`	Concatenates two strings
`strcpy`	Copies one string into another
`strcmp`	Compares two strings
`strchr`	Searches a char in a string

- See man pages
  - Do not reinvent the wheel, there are many many functions that will help you