Programming for Beginners

Lecture 2: Diving into C

Material from: http://www.codingunit.com/

Kai Lampka

Uppsala University

kai.lampka@it.uu.se

Structure of a C program

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

const double tax = 0.2;
int price, items;

int price( int items)
{
    statement 1;
    statement 2;
    return ((1+tax)* items);
}

int main ( void)
{
    statenmt 1;
    int pay;
    pay = price(5);
    return 1;
}
```

Tell pre-processor to load these header files

declare some (module) global variables (scope module global)

define function named price

function body

define function main

Function Body,
This is the scope of function-local declarations (binding of names)!

Comments in C

- Example

```
/* This is and example of a comment
put into a C program */
```

- begin with /* and end with */ indicating that these two lines are a comment.
- **You insert comments to document programs and improve program readability.**
- Comments do not cause the computer to perform any action when the program is run. (They are removed by the pre-processor).

built-in data types in C

The most important base data types in C can be grouped into character, integer and floating point data types

Character data types

Name	Range	Size	Application
`char`	Alpha-numeric character	1 Byte	characters are put in quotes char a = ‘a’;
`char`	-128 to 127	1 Byte	we store integer values char a = 128; (??)
`unsigned char`	0 to 255	1 Byte	positive integer values char a = 256; (??)

Remember: size of a Byte is fixed (8 Bits). Size of a word depends on the architecture. 64-Bit architecture has words of 8 Bytes

integers

Name	Range[1]	Size
`short int`	-32768 to 32767	2 Byte
`int`	architecture dependent	
`unsigned int`	architecture dependent	
`long int`	- 2,147,483,648 to 2,147,483,647	4 Byte
`unsigned long int`	0 to 4,294,967,295	4 Byte
`long long int`	-9,223,372,036,854,775,808 to -9,223,372,036,854,775,8087	8 Byte
`unsigned long long int`	0 to 8,446,744,073,709,551,615	8 Byte

`int` and `unsigned int` have architecture dependent sizes. For a 64-Bit architecture size is 8 Byte.

floating point

Name	Range[1]	Size	Precision
`float`	$1.18 * 10^{-38}$ to $3.4 * 10^{38}$	4 Byte	7 digits
`double`	$2.23 * 10^{-308}$ to $1.79 * 10^{308}$	8 Byte	15 digits
`long double`	$3.37 * 10^{-4932}$ to $1.18 * 10^{4932}$	16 Byte	33 digits
`long long int`	-9,223,372,036,854,775,808 to -9,223,372,036,854,775,8087	8 Byte	
`unsigned long long int`	0 to 8,446,744,073,709,551,615	8 Byte	

Implementation of `long double` is architecture dependent

Remarks

- For the non-signed data types one my use the keyword sign to emphasize the signed character. But one does not need to do this (and nobody actually does)
- For `short, long, signed` and `unsigned int`, the keyword `int` can be omitted
- function **`sizeof(xyz)`** gives you the number of byte of data type xyz

variables

- **Variables refer to locations in memory where a value is stored**. --We see later how we can reference the address an get access to this location.
- **Syntax: data_type identifier;**
- Identifiers: consist of letters, digits (cannot begin with a digit) and underscores(_)
- Identifier are case sensitive
- Declarations appear before executable statements
- If an executable statement references and undeclared variable it will produce a syntax (compiler) error
- **Assignments to variables are done with operator =**

Syntax	Example
`data_typeA name;`	`char a;`
`data_typeA name1, name2;`	`char a, b;`
`data_typeA name1 = value;`	`char a = 'b';`
`data_typeA name1 = expression;`	`char a = 255/2;`

constants

- **A constant is a variable which does not change its value**
- we can only assign a value once to a constant, namely upon declaration
- during the life time of the program, we can only read from this memory location
- **Syntax: const data_type identifier;**
- Name convention of identifier as before
- one may also use a macro definition (preprocessor directive) to define constants (#define tax 0.2). But this is **not the same and should be avoided**. By using a constant of a data type via keyword const you allow the compiler to do type checking. Incase of a macro this is not possible.

Syntax	Example
`const data_typeA name = expression;`	`const double PI = 3.14;`

Example

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

const double tax = 0.2;

int price( int items)
{
    return ((1+tax)* items);
}

int main ( void)
{
    int pay;
    pay = price(5);
    printf("You need to pay:%d", pay);
    return 1;
}
```

Operators

The distinguish between

- ✧ unary, one operand, e.g. negation !A
- ✧ binary, two operands, e.g., addition a+a,
- ✧ ternary operator a?b:c; (if a is true give b else c)

Arithmetic operators

Operator	Example	Remark
Addition: +	b = a + a;	first addition than assignment to variable c
Subtraction: -	b = a - a;	as expected
Multiplication: *	b = a * a;	as expected
Division: /	b = a/ a;	as expected
Modulo: % (division with remainder)	b = a % a;	as expected (gives 0).

Shortforms (combined with assignment)

Operator	Example	Remark
Increment: ++	b++;	gives b = b+1;
Decrement: --	b--;	gives b = b-1;
Addition to a variable	b +=a;	gives b = b + a;
Subtraction, multiplication, division and modulo to and with a variable	a -= b; a*= b; a /= b; a%=b;	as expected

Relational operators

Operator	Example	Remark
smaller: <	b < c	evaluates to true, i.e. 1, if and only if variable b is smaller than variable c
larger : >	b > c	as expected
smaller equal: >=	b >= c	as expected
larger equal : =<	b =< c	as expected
equal: ==	b == c	as expected
not equal: !=	b != c	as expected

Logical operators

Operator	Example	Remark
`and: &&`	`a == 5 && b == 3`	evaluates to true, i.e. 1, if and only if variable a is 5 and b is 3
`or: ||`	`a == 5 || b ==3`	as expected
`not: !`	`!(a == 5)`	evaluates to true if a is not 5.

Bit operators

<table>
<tr><th>Operator</th><th>Example</th><th>Remark
let a = 0011 and b = 1001</th></tr>
<tr><td>and: &</td><td>c = a & b;</td><td>c is …..</td></tr>
<tr><td>or: |</td><td>c = a | b;</td><td>c is ….</td></tr>
<tr><td>xor: ^</td><td>c = a ^ b;</td><td>c is …..</td></tr>
<tr><td>left shift <<</td><td>c = a << b;</td><td>c is …..</td></tr>
<tr><td>right shift >></td><td>c = a >> b;</td><td>c is …..</td></tr>
<tr><td>bitwise negation: = ~</td><td>c = ~b;</td><td>c is …..</td></tr>
</table>

Short forms

Operation short version	long version	Remark let a = 0011
a &= 4;	a = a & 4;	a is …..
a |= 6;	a = a | 6;	a is ….
a ^= 5;	a = a ^5;	a is …..
a >>=2;	a = a >> 2;	a is …..
a << = 2;	a = a << 2;	a is …..

Conversion of data types

In case one uses different data types implicit type conversion rules apply. This may yield:

✧ loss of bit positions or

✧ precision of the floating point

To avoid implicit conversion, one can do an explicit type conversion denoted **cast**

Operation short version	Remark
`int i = 5;` `double b = (double) i;`	The value of variable i is converted into a double and assigned to variable b
`double a = 3.2, b = 4.5;` `double c = (double) ((int) a + (int) b))`	b is …..

Functions

For making programmes

✧ Readable

✧ Re-useable (sub parts)

✧ Isolation of errors

it is highly recommend to partition the programme code into subprogrammes, respectively functions.

Definition of a function.

```
Data-type-of-return-value Name-of-Function ( Parameters )
{ //Body of Function starts here
    statement 1;
    statement 2;
    statement …
    return value;
} //End of function

// Example
int add (int a, int b) { return(a+b);}
```

Functions

For making programmes

- ✧ Readable
- ✧ Re-useable (sub parts)
- ✧ Isolation of errors

it is highly recommend to partition the programme code into subprogrammes, respectively functions.

Syntax of the definition of a function.

```
Data-type-of-return-value Name-of-Function ( Parameters)
{ //Body of Function starts here
    statement 1;
    statement 2;
    statement …
    return value;
} //End of function

// Example
int add (int a, int b) { return(a+b);}
```

Functions

Parameters:

✧ data_type identifier, e.g., int a, int b, double c

✧ entries are separated by komma.

Parameters are function local variable:

✧ identifier is only visible within function

✧ the actual passed in variable is a copy, i.e., any manipulation is not made to the original variable but the copied input parameter.

```
int addAndAssign (int a, int b)
{
    a += b; //value of a here?
    return(a);
}

//somewhere in main()
int a = 10;
addAndAssign(a, 5); //value of a here?
```

Functions

Parameters:

- ✧ data_type identifier, e.g., int a, int b, double c
- ✧ entries are separated by komma.

Parameters are function local variable:

- ✧ identifier is only visible within function
- ✧ the actual passed in variable is a copy, i.e., any manipulation is not made to the original variable but the copied input parameter.

```
int addAndAssign (int a, int b)
{
   a += b;    return(a);
}
```

Remember parameters are copies. One uses new variables here, named a and b. They have the value which is passed in by the function call **call by value**

```
//somewhere in main()
int a = 10;
addAndAssign(a, 5); //value of a here?
```

Functions

- Think of functions as keywords not built in to the programming language.
- All library functions one uses are functions implemented that way.
- There is a large set of functions provided by libraries, using these functions:
    - reduces errors and
    - increase efficiency

as they are well tested and highly optimized.

functions can be stored

- in the same file with main, or
- in a separate file

`#include` precompiler directive or with

`extern Function-Definition`

This is why you need to include the pre-defined functions at the top of your main.c file.

Functions

```
//main.c
#include myTest.h

int main (){
 return(Add(10, 5));
}
```

```
//myTest.c
#include myTest.h

int add(int a, int b){
    return(a+b);
}
```

```
//myTest.c
#ifndef MYTEST
#define MYTEST

int add(int a, int b);

#endif
```

```
//main.c

#extern int Add(int a, int b);

int main (){
    return(Add(10, 5));
}
```

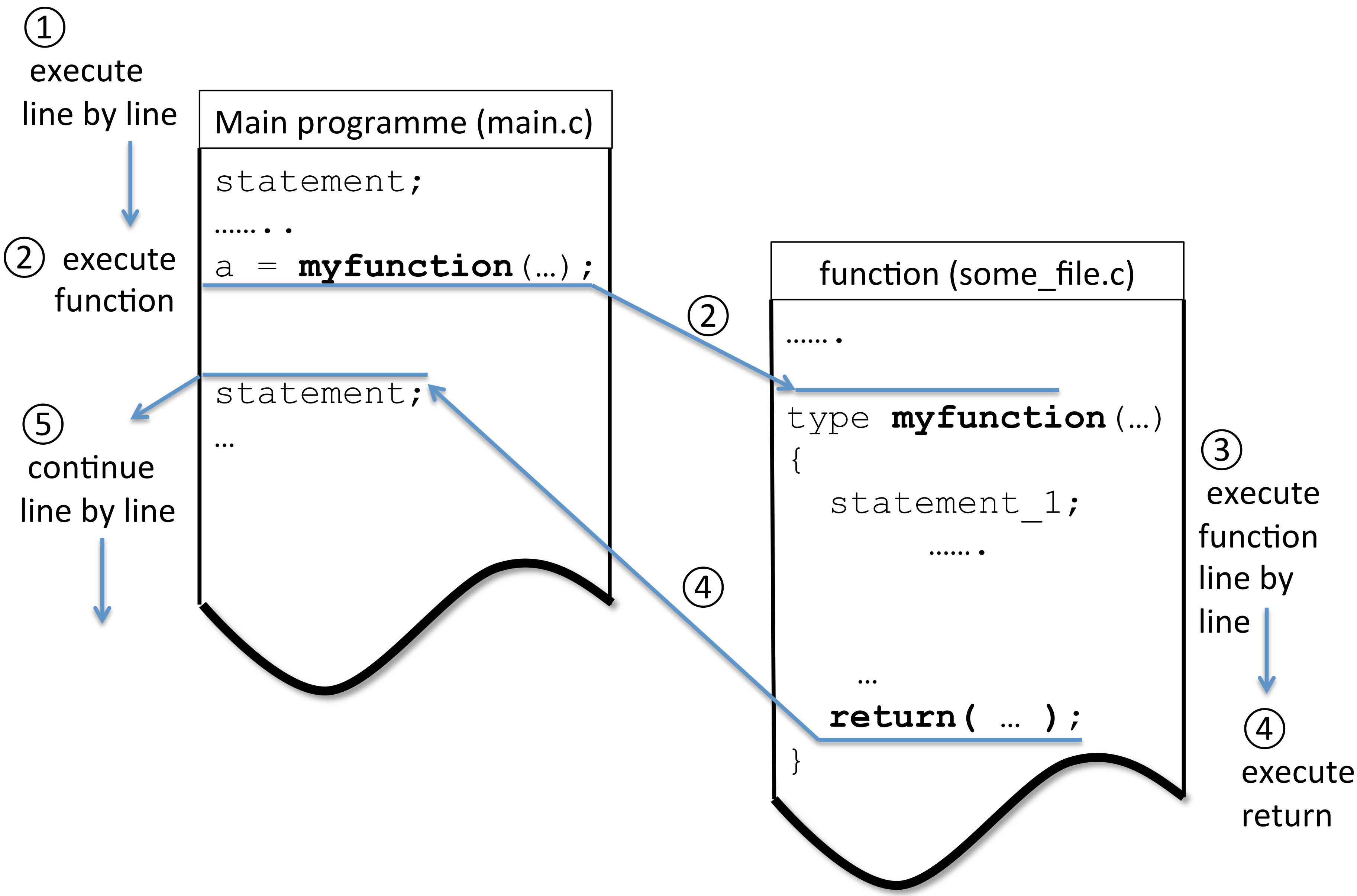
```
//myTest.c
#include myTest.h

int add(int a, int b){
    return(a+b);
}
```

Some final remarks:

- Functions are called by their name and the parameters filled in correctly
- The number and types of parameters must macth, otherwise the compiler will issue an error or a warning
- the return value of a function can be used in an assignment or an expression
- if the function is defined to give a return value, there must at least be one return statement in the definition.
- There might be more than one return statement in the function
- To actually use a function, it must have been declared, most likely via include of the respective header file

Functions



① execute line by line
② execute function
⑤ continue line by line
Main programme (main.c)
statement;
……. .
a = myfunction(…);
statement;
…
②
④
function (some_file.c)
…….
type myfunction(…)
{
statement_1;
…….
…
return(…);
}
③ execute function line by line
④ execute return

Reading from STDIN

- Function `int getchar(void)` reads characters from STDIN (here keyboard), reading starts as soon as ↵ is pressed.
- The return value corresponds to the ASCII-value of the supplied character (you find the Tabelle on the web).
- Library: stdio.h / Prototype: int getchar(void); /Syntax: ch = getchar();

```
#include <stdio.h>
#define RETURN '\n'
// \n == return in UNIX \r == return in DOS      */

int main(){
  int count=0;
  puts("Please enter some text.");

  // Count the letters in the 'stdin' buffer.
  while ( getchar() != RETURN) count++;

  printf("You entered %d characters\n", count);
  return 0;
}
```

Writing to STDOUT

- Function `int putchar(int c)` writes a character (an unsigned char) specified by the argument c to stdout.
- The return value corresponds to the ASCII-value of the written character c (you find the Tabelle on the web).
- Library: stdio.h / Prototype: int putchar(int c);
- Syntax: ch = putchar(c);

```
#include <stdio.h>

int main ()
{
   char ch;

   for(ch = 'A' ; ch <= 'Z' ; ch++) {
      putchar(ch);
   }

   return(0);
}
```

function printf()

- The `printf` function is another useful function from the standard library
- Syntax: `printf(“expression”, variable 1, … );`
- `expresion` is text mixed with format specifiers for the variables
- the format specifiers are mapped to the variables 1:1 in the order of appearance

%i or %d	int
%c	char
%f	%f float (see also the note next page)
%s string	string

function printf()

%f stands for float but…..

- ✧ Default argument promotions happen in variadic functions. Variadic functions are functions (e.g. printf) which take a variable number of arguments. When a variadic function is called, after lvalue-to-rvalue, array-to-pointer, and function-to-pointer conversions, each argument that is a part of the variable argument list undergoes additional conversions known as default argument promotions:
  - float arguments are converted to double as in floating-point promotion
  - bool, char, short, and unscoped enumerations are converted to int or wider integer types as in integer promotion
- ✧ So for example, float parameters are converted to doubles, and char's are converted to int's. If you actually needed to pass, for example, a char instead of an int, the function would have to convert it back.

function printf()

```
#include<stdio.h>

main(){
    int a,b;
    float c,d;

    a = 15;
    b = a / 2;
    printf("%d\n",b);
    printf("%3d\n",b);
    printf("%03d\n",b);

    c = 15.3;
    d = c / 3;
    printf("%3.2f\n",d);
}
```

<u>Useful special signs to be used in the expression passed to printf():</u>

- \n (newline)
- \t (tab)
- \v (vertical tab)
- \f (new page)
- \b (backspace)
- \r (carriage return)
- \n (newline)

function printf()

```
include<stdio.h>

main(){
 int F;

 for (F = 0; F <= 300; F += 20)
  printf("Fahrenheit:%3d Celsius:%06.3f\n", F, (5.0/9.0)*(F-32));
}
```

- As you can see we print the Fahrenheit temperature with a width of 3 positions.
- The Celsius temperature is printed with a width of 6 positions and a precision of 3 positions after the decimal point.
- Examples of format specifiers

  %d (print as a decimal integer)

  %6d (print as a decimal integer with a width of at least 6 wide)

  %f (print as a floating point)

  %4f (print as a floating point with a width of at least 4 wide)

  %.4f (print as a floating point with a precision of four characters after the decimal point)

  %3.2f (print as a floating point at least 3 wide and a precision of 2)

function scanf()

- The `scanf()` function is another useful function from the standard library and it reads formatted input from stdin.
- `scanf(const char *format, variable 1, … );`
- format is the C string that contains one or more of the following items: **Whitespace character**, **Non-whitespace character** and **format specifiers.** **F**ormat specifier is as before:

`[=%[*][width][modifiers]type=]` see below:

`*`	This is an optional starting asterisk indicates that the data is to be read from the stream but ignored, i.e. it is not stored in the corresponding argument.
`width`	This specifies the maximum number of characters to be read in the current reading operation
`modifiers`	Specifies a size different from int (in the case of d, i and n), unsigned int (in the case of o, u and x) or float (in the case of e, f and g) for the data pointed by the corresponding additional argument: h : short int (for d, i and n), or unsigned short int (for o, u and x) l : long int (for d, i and n), or unsigned long int (for o, u and x), or double (for e, f and g) L : long double (for e, f and g)
`type`	A character specifying the type of data to be read and how it is expected to be read. See next table

function scanf()

```
#include <stdio.h>

int main()
{
   char str1[20], str2[30];

   printf("Enter name: ");
   scanf("%s", &str1);

   printf("Enter your website name: ");
   scanf("%s", &str2);

   printf("Entered Name: %s\n", str1);
   printf("Entered Website:%s", str2);

   return(0);
}
```

function scanf()

Types

`c`	Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.	char*
`d`	Decimal integer: Number optionally preceeded with a + or - sign	int *
`e,E, f,g, G`	Floating point: Decimal number containing a decimal point, optionally preceeded by a + or - sign and optionally folowed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float*
`o`	OctalInteger:	int*
`s`	String of characters. This will read subsequent characters until a whitespace is found (whitespace characters are considered to be blank, newline and tab).	char *
`u`	Unsigned decimal integer.	unsigned int *
`x,X`	Hexadecimal Integer	int *

The `data_type *` says that we actually referring to the address where the resp. data is stored, we come back to this.