C Programming Lab

Advanced Computer Science Studies in Sweden
Introduction to Studies in Embedded Systems
HT 2015

Kai Lampka
Philipp Rümmer

Two lab slots

- Thursday Sept 3rd, 8:00 – 12:00
- Tuesday Sept 8th, 8:00 – 12:00

- Both start with a lecture part, following by time to work on the assignments
- **Refresher!**
- **Not** meant as a course for beginners
  → we assume that you all had some education in programming before

Assignments, Instructions

- Assignments to be solved **individually**, or in groups of **two people**
- Deadline for submitting solutions: **Tuesday September 8th, 12:00**
- Document contains assignments in order of increasing difficulty
  → At least 4 assignments have to be solved to pass the lab

http://www.it.uu.se/edu/course/homepage/avandatintrois/ht15/c-lab/assignments.pdf

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;
}
```

Structure of a C program

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

```
#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)
{
    statemenmt 1;
    int pay;
    pay = price(5);
    return 1;
}
```

Tell pre-processor to load these header files

define function named price

function body

define function main

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

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

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

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

Short forms

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

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

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?
```

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).

if-then-else-statement

An if statement can be extended with an else-statement. In case the expression of the if-statement is 0, the body of the else is executed.

Syntax

```
if (expression){ body of if-statement }
else { body of if-statement }
```

- ✧ The body can be a single statement, than you do not need the curly brackets
- ✧ An else-statement always refers to the previous if-statement, curly brackets improve readability!

```
int mynumber = scanf("%d",&mynumber);

if ( mynumber == 10){
    printf("is equal 10\n");
    printf("closing program\n");
    return 0;
}
else printf("is not equal 10\n");
```

if-then-else-statement (more)

`Elseif-statement.`

- ✧ Does not exists in C, instead one may sue else if { .. }.
- ✧ This works as the if-statement and its body is seen as single line.

```
if( expression1 )
   statement1;
else if(expression2 )
   statement2;
else if(expression3 )
   statement3;
   .
   .
else
   statementN;
```

- ✧ **An else-statement always refers to the previous if-statement, curly brackets improve readability!**

```
int p = 0
if(0)
if(1) p = 5;
else p = 1;
```

?-statement

if-then-else can be replaced with a single statement

Syntax

`result = test-expression ? value1 : value2;`

If `test-expression` evaluates to true result is assigned the value `value1`, otherwise result is assigned the value `value2`.

```
int mynumber = scanf("%d",&mynumber);

if ( mynumber == 10)
    printf("is 10\n");
else
    printf("is not 10\n");
```

```
int mynumber = scanf("%d",&mynumber), c;
c = (mynumber != 10) ? printf("is not 10\n") : printf("is 10\n");
```

```
// another example
int max;
max = (v1 > v2) ? v1 : v2;
```

switch-statement

- The switch statement can have many conditions. You start the switch statement with a `switch-expression` which is evaluated.
- If one of the `case expressions` equals the value of the expression, the instructions are executed until a break is encountered.
- If none of the `case expressions` equals the switch expression the default is executed

```
int main() {
    char myinput;
    printf("Choose: \t a) Program 1 \t b) Program 2\n");
    scanf("%c", &myinput);
    switch (myinput){ // variables are evaluated
        case 'a': //compare value to case-expression
            printf("Run program 1\n");
            break;
        case 'b':
            printf("Run program 2\n");
            printf("Please Wait\n");
            break;
        default:
            printf("Invalid choice\n");
            break;
    }
return 0;}
```

The break-statements are required to exit the switch statement, **otherwise everything behind will be executed as well, until the next break or the end of the switch** *–no re-testing of the variable again!*

while-Loop

```
while (loop-condition)
{
    // loop body
} // brackets can be omitted for a single statement
```

- ✧ The loop condition is the expression which tells us if we can enter (again).
- ✧ No pre- and post-loop statements

```
#include<stdio.h>

int main(){
 int i, howmuch;
 scanf("%d", &howmuch);
 i = 0;
 while(i < howmuch) printf ("Hello World: %d\n”,++i);
 return 0;
}
```

Be aware of endless loops and that variables have always the intended value.

do-while-Loop

- The “do while loop” is almost the same as the while loop. But loop-condition is tested after the body!.
- The “do while loop” has the following form:

```
do
{
   // loop body
} // brackets can be omitted for a single statement
while (loop-condition);
```

```
#include<stdio.h>

int main(){
 int i, howmuch;
 scanf("%d", &howmuch);
 i = 0;
 do {
    printf ("Hello World: %d\n”,++i);
 } while(i < howmuch);
 return 0;
}
```

Be aware of endless loops and that variables have always the intended value.

for-Loop

```
for (pre-loop statement; loop-condition; post-loop statement)
{
   // loop body
} // brackets can be omitted for a single statement
```

- The pre-loop statement is executed before the first loop entry.
- The loop condition is the expression which tells us if we can enter (again).
- post-loop statement is executed after each loop iteration.

```
#include<stdio.h>

int main(){
    int i;
    for (i = 0; i < 10; i++){
      printf ("Hello World: %d\n”,i);
    }
    return 0;
}
```

Be aware of endless loops and that variables have always the intended value.

pre-mature leave or re-entering of a loop

- With a break-statement the loop is left immediately
- with a continue-statement one directly jumps to the loop-head and tests the loop condition again (for and while loop)

```
#include<stdio.h>
int main(){
    int i;
    for (i = 0; i < 10; i++){
     if(!(i % 5)) continue;
     printf ("Hello World: %d\n”,i);
    }
    return 0;}
```

```
#include<stdio.h>
int main(){
 int i, p;
 printf (”\nGive a number to be tested for being prime\n”);
 scanf("%d", &p);
 for (int i = 2; i < p; i++){
  if (p % i == 0) break;
 }
 if (i == p) printf (”%d is a prime \n”,p);
 else printf (”%d is not a prime \n”,p);
 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()

```
#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 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);
}
```

What is a pointer ?

- In C, a pointer variable (or just "pointer") is similar to a reference and it can contain the memory address of any variable
    - A primitive (int, char, float)
    - An array
    - A struct or union
    - Dynamically allocated memory
    - Another pointer
    - A function
  - There's a lot of syntax required to create and use pointers

Pointer Caution

- They are a powerful low-level device.
- Undisciplined use can be confusing and thus the source of subtle, hard-to-find bugs.
  - Program crashes
  - Memory leaks
  - Unpredictable results

Pointer Declaration

The declaration

```
int *intPtr;
```

defines the variable `intPtr` to be a pointer to a variable of type `int`. `intPtr` will contain the memory address of some `int` variable or `int` array. Read this declaration as

- "`intPtr` is a pointer to an `int`", or equivalently
- "`*intPtr` is an `int`"

Caution -- Be careful when defining multiple variables on the same line. In this definition

```
int *intPtr, intPtr2;
```

`intPtr` is a pointer to an `int`, but `intPtr2` is not!

Pointer Operators

The two primary operators used with pointers are
*** (star) and & (ampersand)**

- **The * operator is used to define pointer variables and to *deference* a pointer. "Dereferencing" a pointer means to use the value of the pointee.**
- **The & operator gives the address of a variable.**
  Recall the use of & in `scanf( )`

Pointer Examples

```
int x = 1, y = 2, z[10];
int *ip;          /* ip is a pointer to an int */

ip = &x;          /* ip points to (contains the memory address of) x */
y = *ip;          /* y is now 1, indirectly copied from x using ip */
*ip = 0;          /* x is now 0 */
ip = &z[5];       /* ip now points to z[5] */
```

If ip points to x, then *ip can be used anywhere x can be used so in this example *ip = *ip + 10; and x = x + 10; are equivalent

The * and & operators bind more tightly than arithmetic operators so

y = *ip + 1; takes the value of the variable to which ip points, adds 1 and assigns it to y

Similarly, the statements *ip += 1; and ++*ip; and (*ip)++; all increment the variable to which ip points. (Note that the parenthesis are necessary in the last statement; without them, the expression would increment ip rather than what it points to since operators like * and ++ associate from right to left.)

NULL

- NULL is a special value which may be assigned to a pointer
- NULL indicates that this pointer does not point to any variable (there is no pointee)
- Often used when pointers are declared

```
int *pInt = NULL;
```

- Often used as the return type of functions that return a pointer to indicate function failure

```
int *myPtr;
myPtr = myFunction( );
if (myPtr == NULL){
    /* something bad happened */
}
```

- Dereferencing a pointer whose value is NULL will result in program termination.

Pointers and Function Arguments

- Since C passes all primitive function arguments "by value" there is no direct way for a function to alter a variable in the calling code.

- This version of the `swap` function doesn't work. **WHY NOT?**

```
/* calling swap from somewhere in main() */
int x = 42, y = 17;
Swap( x, y );

/* wrong version of swap */
void Swap (int a, int b)
{
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

A better swap( )

- The desired effect can be obtained by passing pointers to the values to be exchanged.
- This is a very common use of pointers.

```
/* calling swap from somewhere in main( ) */
int x = 42, y = 17;
Swap( &x, &y );

/* correct version of swap */
void Swap (int *px, int *py)
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
```

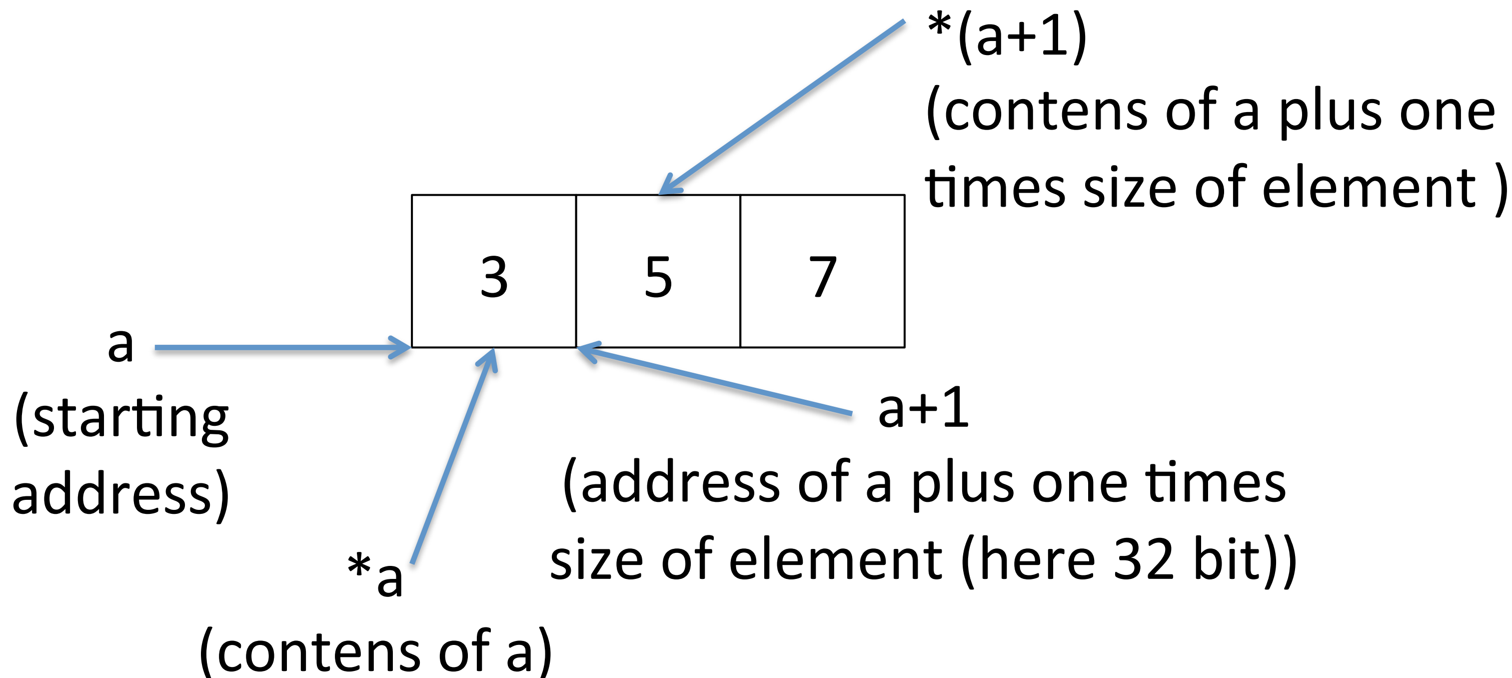
Pointer

✧ Operators to pointers:

* : contents of a pointer

&: address of a pointer

```
int n = 10;
int a[3] = {3,5,7};
```



Exercises

1) Output
2) Input
3) Conditionals
4) Loops
5) Loops II
6) Functions
7) Functions II
8) Arrays
9) Pointers and Strings
10) Malloc, sorting strings
11) Sorting arrays in linear time
12) Recursion
13) Efficient Fibonacci
14) Cmdl. arguments, file I/O
15) Linked lists
16) Function pointers
17) Hamming weight
18) SCCs in graphs