Programming in C
week 1 meeting

2.9.2015

Tiina Niklander

Course structure

- Based on C programming course in Aalto, but with some exercises created locally
- Course structure is in wiki: https://wiki.helsinki.fi/display/Cprogramming2015/
- Exercise submission on our own server: http://tmc.mooc.fi/hy
    - use your student ID as your username!
- Grading based on TMC exercises + Exam!!!

Course material

- Use a course book and some extra on-line material,
  - e.g. Aalto on-line material: http://src.aalto.fi/c
- ONE good C course book is a MUST:
  - Kernighan & Richie: C programming language
  - Müldner : C for Java programmer
  - King: C Programming, A Modern Approach

Learning goal objectives

- Language structures, data structures, modules, functions+parameters
- Files
- Pointers, structured elements
- Using properly dynamic and static data structures like tables, lists, queues, stacks, trees (from data structures and algorithms course!)

NOTE: No C++ features allowed, pure ANSI C only

TMC Deadline Tuesdays at 19.00

TMC checks automatically the programming tasks and maintains the list of accepted tasks.

On-line tool, so can be used from home or at university

Paja times (in B221):

- Thu 3.9. at 12.15-14 is only paja this week.
- Usually on Mon 13-18 & Tue 13-18

Exam

- Programming on paper!!!
- Last Fall's exam had 4 questions
  - Calculate averages (function + main + command line parameters/arguments)
  - Personal contacts (create data structure + handling functions – write comments, but not all codes)
  - File handling and I/O streams – essay
  - Errors and mistakes – evaluate an existing program
- 2,5 hours

C language

C assumes that programmer is intelligent enough to use all of its constructs wisely, and so few things are forbidden.

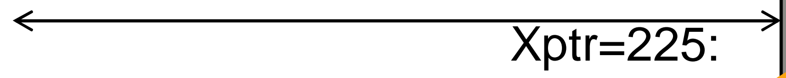
- Closely tight with UNIX – same original developers

- ’Low-level’ language
  - explicit memory allocation and deallocation!
  - allows operations on the memory addresses and bit-level
  - allows dynamic type changes of variables
  - important concept: POINTER (= memory address)

C can be a very useful and elegant tool. People often dismiss C, claiming that it is responsible for a ”bad coding style”. The bad coding style is **not the fault of the language**, **but** is controlled (and so caused) by **the programmer**.

Addresses and values

```
Xptr  DC   0
X     DC   12
      LOAD  R1, =X       ; R1 ← 230
      STORE R1, Xptr
      LOAD  R2, X        ; R2 ← 12
      LOAD  R3, @Xptr    ; R3 ← 12
```



TITO

- Variable X's address is 230 (&X)
- Variable X's value is 12 (X)
- Pointer Xptr's address is 225 (&Xptr)
- Pointer Xptr's value is 230 (Xptr)
  - Address of some data (now the address of X)
- The value of the integer that Xptr points to is 12 (*Xptr)

In C language as: Y = *Xptr;
/* value of the location Xptr points to */

Example program

```
#include<stdio.h>          /*Header file of a library*/
int main(int argc, char **argv) /* The main function */
{
   int x, *y, z;   /*Variable Declaration*/
  if ( (y = malloc (sizeof(int))  ==NULL) return(1);
   x = 5;
   *y = 10;
   z = x + *y;
   printf ("The sum is %d", z); free(y);
   return (0);       /* standard return value for success */
}
```

Compiling:
gcc -ansi -pedantic -Wall

- Options –Wall and –pedantic show all possible warnings (used by TMC)
- Option –ansi makes sure that the compiler follows standard (some differences in gcc exist)

```
gcc -ansi -pedantic -Wall -o helloworld helloworld.c
/home/fs/niklande/C-luennot/esimerkit/helloworld.c: In function `main':
/home/fs/niklande/C-luennot/esimerkit/helloworld.c:3: warning:
implicit declaration of function `printf'
```

```
int main (void)
{
  printf(”Hello world \n”);
  return 0;
}
```

```
#include <stdio.h>
int main (void)
{
  printf(”Hello world \n”);
  return 0;
}
```

Multiple modules
(separate source files with headers)

```
/* main.c */
#include <stdio.h>
#include ”eka.h”
#include ”toka.h”
int main (void)
{
  eka(); toka ();
  return 0;
}
```

```
/* eka.c */
#include <stdio.h>
#include ”eka.h”
void eka (void)
{
  puts(” eka ”);
}
```

```
/* eka.h */

void eka (void);
```

```
/* toka.c */
#include <stdio.h>
#include ”toka.h”
void toka (void)
{
  puts(” toka ”);
}
```

```
/* toka.h */

void toka (void);
```

Could write everytime:

```
gcc –c main.c
gcc –c eka.c
gcc –c toka.c
gcc –o ohjelma main.o eka.o toka.o
```

→ Instead, use **makefile**

makefile

```
gcc –c main.c
gcc –c eka.c
gcc –c toka.c
gcc –o ohjelma main.o eka.o toka.o
```

make

Write makefile just once

Use several times with command make

```
# makefile
CC = gcc –ansi –pedantic –Wall
ohjelma: main.o eka.o toka.o
	$(CC) –o ohjelma main.o eka.o toka.o
eka.o: eka.c eka.h
	$(CC) –c eka.c
toka.o: toka.c toka.h
	$(CC) –c toka.c
main.o: main.c eka.h toka.h
	$(CC) –c main.c
```

Modular programming in C

C does not support objects or modular programming, but it has features that can be used to mimic them

- Functions and function prototypes
- Header files

Using these features you can separate the interface and implementation

’hidden’ implementation

public interface

Structuring a small C program: order of elements in the file

#include directive to use (standard) libraries

Defining constants ad types

- Easy to locate, modify and control

Function prototypes (before the functions in used)

Function main (every program has one)

Function definitions (=code of the function)

What if you have a larger project?

- Split to multiple files and libraries

main

sub1 sub2 sub3 sub4

subsub1 subsub2 subsub2

Functions of standard libraries

Programmers own libraries of functions

Modularity at the function level

Standard libraries

- Each library contains a set of useful functions
- A header file gives the function signatures and type definitions for that particular library
- stdio.h is usually always used, since it contains the functions for reading and writing
- math.h contains mathematical functions
- assert.h macro assert to help locate mistakes

- 29 header files in total (see wikipedia for the list)

Comparison of C and Java

from Mülder: C for java programmers

- *Primitive data types*: character, integer, and real.
  In C, they are of different sizes,
  there is no Unicode 16-bit character set
- *Structured data types*: arrays, structures and unions.
  In C, arrays are static
  there are no classes
- *Control structures* are similar
- *Functions* are similar

Comparison of C and Java

- Java references are called pointers in C.

- Java constructs missing in C (ANSI):
  - packages
  - threads
  - exception handling
  - garbage collection
  - standard Graphical User Interface (GUI)
  - built-in definition of a string
  - standard support for networking
  - support for program safety.

Programming style

Write clear and easily understandable code (Java programming course style is good!)

Cryptic and concise code has no value itself, the clarity is more important

```
do {
  if (scanf(”%d”, &i) !=1 ||
       i == SENTINEL)
    break;
  if (i>maxi)
    maxi = i;
} while (1);
```

```
void show (char *p) {
  char *q;
  printf(”[ ”);
  for (q=p; *q != ’\0’; q++)
    printf(”%c ”, *q);
  printf(”]\n”);
}
```

What do the following sentences do?

```
while (*q++ = *p++);
```

```
if ((c=fgetc(fileHandle)) == EOF)
```

```
for (i=a, j=b; i<=j; i += 2, j += 2)
```

Another example

Emply lines would
make it more readable

```
#include<stdio.h>
#include<string.h>
#define SUCCESS 0
#define FAILURE -1
int main(void)
{
   char pass_buff[50] = {0};
   printf("\n Enter the password...");
   //Get the password from user
   fgets(pass_buff,sizeof(pass_buff)-1,stdin);
   //Make sure that the extra(last) character
   //picked up from stdin is washed off
   //from buffer. pass_buff[strlen(pass_buff)-1] = '\0';
   if(! (strcmp(pass_buff,"Linux")))  {
      // Passwords match
      printf("\n Passwords Match..SUCCESS\n");
      return SUCCESS;
   } else {
      // Passwords do not macth
      printf("\n Passwords do not match...FAILURE\n");
      return FAILURE;
} }
```

From: mylinuxbook.com

Programming in C

- Command-line interface
  - needs only shell, no graphical interface
  - the ’old fashioned’ way
  - most UNIX experts still do it this way
  - explicit compilation and linking
- Integrated Development Environments (IDEs)
  - TMC is integrated with NetBeans
  - Eclipse is another very popular one
  - Program still needs to be compiled and linked before execution – IDE might hide these phases

Development process with command-line interface

Write program

- Using suitable editor (vim, nano, emacs, ...)
- Must produce plain text file

Compile

- Select the correct compiler and arguments (gcc -c)

Link

- Link the compiled modules to form a program (gcc)

Execute

- Executing or running the program

gcc --help

Usage: gcc [options] file...

Option

-- help lists all possible options and features of the compiler, there are plenty

-c compile and assemble, but do not link

-o <file> place the output into <file>

Compiling (short program, all in one file)

Compile

gcc helloworld.c

or

*gcc –o helloworld *

helloworld.c

Compiler does

preprocessing,

actual compilation and

linking

```
int main (void)
{
  printf(”Hello world \n”);
  return 0;
}
```

... and produces the executable file

a.out

or

helloworld

Program has multiple files and modules

Every program file is compiled separately to create object module (or library)

gcc –c main.c

The modules are then linked together to form the complete program

gcc –o program main.o eka.o toka.o

Usually compiled with **make** (executes a sequence of commands)

makefile

```
gcc –c main.c
gcc –c eka.c
gcc –c toka.c
gcc –o ohjelma main.o eka.o toka.o
```

Create makefile once,

use it multiple times

↓

```
make
```

```
# makefile
CC = gcc –ansi –pedantic –Wall
ohjelma: main.o eka.o toka.o
	$(CC) –o ohjelma main.o eka.o toka.o
eka.o: eka.c eka.h
	$(CC) –c eka.c
toka.o: toka.c toka.h
	$(CC) –c toka.c
main.o: main.c eka.h toka.h
	$(CC) –c main.c
```

NOTE: tab, not spaces

After compiling and linking?

We have executable program, but does it work?

- Try to execute it and test the program
- Look for errors, locate mistakes
  - Extra print statements
  - Read the code, simulate execution, THINK!
  - Use debugger
- Write automated tests, evaluate test coverage (-> course: Software testing)

  On this course we are satisfied with smoke testing or sanity check. (Program seems to satisfy the desciption.)

Testing

Target: find mistakes

Covers wide spectrum of possible input values

Can be automated (using scripts or special test tools/packages)

Not part of the learning goals of this course

On this course (smoke test, sanity check) usually enough to check

- Correct and incorrect input values
- Boundary values(-1,0,1)

Extra print statements

```
#ifdef EXTRAPRINT
printf (”Fname: Variable name %d \n”, variable);
#endif
```

Target: try to understand the functionality of the code.

Placed: around the most probably mistake location

Usually more handy than debugger, if you have an idea of the error location in advance.

Debugger gdb

(gdb) help

List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

core dump

Crashed program usually produces a file (code dump) that contains state of the memory and registers at the time of crash.

This file can be accessed by debugger to find out values of variable and the location of the instruction that crashed the program.

This is not covered on the course, but could be very useful to study at some point.