# Programming in C

## Week3

**2.9.2015**

**Tiina Niklander**

# Structure

- Previous week
  - Discussion
  - Need to revisit some topics?
- This week
  - Files
  - Some extra material (not covered in TMC tasks)
    - Preprocessor control

# Pass the papers

- Write on the paper questions you want to discuss today or next week.

- unclear concepts
- clarification requests
- …

# Discussion about week2

- Over 80 persons did almost all tasks on time
- Some issues
  - Not checking the return values
  - Omitting the null character from the string ends
  - Using uninitialized variables
  - Not passing addresses properly to functions

# Files

# Files

**(Müldner, chapter 5)**

Text file ⇔ binary

- File handle (vs. file pointer)

Open and close

- How to use a file
- Errors in opening (and closing)

Standard Files

Basic I/O-operations for files

- Read and write
- Examples and idioms

# Files

Just sequence of bytes.

**EOF** byte at the end of the file.

Two types; only difference in handling

**Text files** handled by rows. At the end of each line a end-of-line byte '**\n**' (Begining of new line)

Binary file has NO special bytes within the file.

*NOTE: Different operating systems may use different ways (e.g. byte sequences) to indicate end of line or file!*

# File pointer (a.k.a file handle)

File pointer is a ’handle’ to the FILE structure, which is passed to functions handling the file.

Defining a variable for file pointer:

```
FILE *handle;
FILE *myFile1, *myFile2;
```

```
typedef FILE* P_FILE;
P_FILE  myFile1, myFile2;
```

/* you can define a NEW type!!! */

# Open

Open with **fopen()** before using.

Open connects the file pointer and the file.

Must give file name and the usage mode.

```
handle = fopen(”testitiedosto”, ”r”);
myFile1 =  fopen(”MyFile.txt”, ”w”);
myFile2 =  fopen(”test.out”, ”wb”);
```

# Usage modes as string

- **`"r"`** read from *existing* file (from beginning)
- **`"w"`** write to existing file (overwrite) or
  to new file that is created here (at beginning)
- **`"a"`** write to end of existing file (append) or to new file (at end)
- **`"r+"`** read and write, behaves like **`"r"`**
- **`"w+"`** read and write, behaves like **`"w"`**
- **`"a+"`** read and write, behaves like **`"a"`**

If you want to handle binary file, add **b** to the string somewhere: **`"r+b“.`**

If file has been opened for both reading and writing, between I/O operations one of the functions must be called::

fseek(), fsetpos(), rewind() or fflush().

# File open can fail!

If the open fails, fopen returns NULL

SO: Every time must check!!

Idioms

```
if (fileHandle = fopen(fname, fmode)) == NULL)
    /* statements for the error handling */
```

# About file names and maximum number of open files

File name can contain also path.

- Different OS use different path names. Might contain character with special meaning in C language. (E.g. DOS/Windows uses \)
  - In C \ is used as an indicator that the next character will have special meaning => In DOS \ must be replaced with \\
- FILENAME_MAX (stdio.h) indicates the maximum allowed length of a file name.

Number of concurrently open files is limited: at most FOPEN_MAX (stdio.h) allowed.

# Close

Close the file, when not used anymore **fclose()**.

Argument File pointer to the file to be closed.

If fclose fails, it returns EOF.
ALWAYS CHECK!

Idioms

```
if (fclose(myFile1)  == EOF)
 /* statements for situation where close failed */
```

# Standard files: stdin, stdout, stderr

Always usable, predefined file pointers

No open or close needed!

| | |
|---|---|
| **stdin** | standard in; keyboard or redirection by OS |
| **stdout** | standard out; screen or redirection by OS |
| **stderr** | standard err; separate from stdout (usually screen also) |

```
FILE *inhandle;
……
inhandle = stdin;
```

/* inhandle becomes a synonym to stdin*/

# Basic I/O-operations for files

Functions for files similar than the ones always using stdin or stdout:

| | |
|---|---|
| **int fgetc (fileHandle)** | ~ int getchar() |
| **int fputc (int, fileHandle)** | ~ int putchar(int) |
| **int fscanf (fileHandle, …..)** | ~ int scanf(…) |
| **int fprintf (fileHandle, ……)** | ~ int printf(…) |

# Reading one char or int from a file

\dioms

Reading one character

```
if ((c=fgetc(fileHandle)) == EOF)
/*statements for the end-of-file handling */
```

Reading one integer

```
if (fscanf (fileHandle, ”%d”, &i) != 1)
/* statements to do when read fails*/
```

# Short task (2 minutes)

Write a program that reads three decimal numbers from file *num* and prints their sum.

- Structure of the program
- statements in each phase

# Example: Read three decimal numbers from file *num* and print their sum.

```
int main() {
    FILE *f;
    double x, y, z;

    if((f = fopen("num", "r")) == NULL) {
        fprintf(stderr, "Not opened: %s\n",
        "num");
        return EXIT_FAILURE;
    }
```

File open idiom!

```
 if(fscanf(f, "%lf%lf%lf", &x, &y, &z) != 3) {
   fprintf(stderr, “Read from file failed\n");
   return EXIT_FAILURE;
}
```

File read idiom

```
printf("%f\n", x + y + z);

if(fclose(f) == EOF) {
  fprintf(stderr, “File close failed\n");
  return EXIT_FAILURE;
}
return EXIT_SUCCESS;
}
```

File close idiom

# End of line, end of file

```
/* Read one line and print it in stdin*/
while((c = fgetc(handle)) !='\n')
  if (c == EOF) break;
    else putchar(c);
if(c != EOF) putchar(c);

/* Just locate the end of line */
while((c = fgetc(handle)) !='\n‘);

/* Count the number of characters on the line */
while((c = fgetc(handle)) !='\n‘) ccount++;
```

# Write a program that counts and prints the number of lines in file “test.txt”.

STRUCTURE OF THE PROGRAM:

Definitions (header files, types, variables etc.)

Open file

Count the number of lines in file

    While not EOF

        If read ‘\n’  increment counter

    Print the number of lines

Close file

# More functions: ungetc, feof

## ungetc (char, fileHandle);

’Return the read character back to buffer’. Actually moves the read position backwards on character. Does not change the file content.

- while (cond. (c = fgetc (filehandle)))

  process c;

  ungetc (c, filehandle);

## feof(fileHandle);

Test the end of file. Works only **after** read!!!
Returns 0, if at end of file, otherwise something else.

# Idioms

File open

```
if ((fileHandle = fopen(fname, fmode)) ==NULL) …  /* failed */
```

```
if (close(fileHandle) == EOF) ….  /* failed */
```

File close

```
if ((c = fgetc(fileHandle)) == EOF) …   /* error */
```

Read one char

```
if (fscanf (fileHandle, ”%d”, i) !=1)  …  /* error */
```

Read one number

```
while ((c= fgetc(fileHandle)) != ’\n’)
```

Read to end of line

```
while ((c= fgetc(fileHandle)) != EOF)
```

Read to end of file

# Precompiler, more about macros, conditional compilation

# Precompiler, conditional compilation, macros (Müldner Ch. 6)

C precompiler

Macros

- Including external files ( using since week1)
- Simple macros  (last weeks slides)
- Macros with parameters (Week5 exercises)
- Predefined macros

Conditional compilation

- Use in debugging
  - Assert-macro
- Use with and in header files
- Use to increase portability

# C precompiler

## # at the beginning of the line

Line for precompiler => separate syntax

Processed before compilation of the actual code

## Purpose?

**Macros**: replace text with some other text

**Include external files**

- #include <stdio>

**Conditional compilation:** only part of the source code in compiled under certains conditions; used for debugging, portability,

# Macros

Simple macros (no parameters)

- Shortening?; macro name replaced always with the same text

Parameterized macros

- Parameters effect the replacement => More advanced, but has easily unexpected side effects

Predefined macros

- Part of the C compiler implementation
- Useful in error situations

# Simple macro

- `#define  macroName  macrovalue`

```
#define  PI                  3.14
#define  PII                 3.14159265358979323\
846264338327950
#define SCREEN_W     80
#define SCREEN_H     25
```

**MacroName usually with CAPITAL LETTERS!**

**MacroValue until end of the line (‘\n’ char)**

**‘\’ = continues to next line**

**NOTE: NO = or ;  (equal or semicolon)!**

# Macro name replaced with macro value

Precompiler replaces every appearance of macroName with **text** of macroValue in source file.

```
#define  PI    3.14
i=PI;
```

**In compilation  =>  i=3.14**

```
#define  PII=3.14;
i=PII;
```

**In compilation => i ==3.14;;**

**INCORRECT!!!!**

# Examples

```
#define PROMPT  printf("Enter real value: ")
#define SKIP    while(getchar() != '\n');
```

**Use parenthesis especially with math ops!**

```
#define A       2 + 4
#define B       A * 3
```

=> B = 2 + 4 * 3 = 14

Was this wanted result?

```
#define A       (2 + 4)
#define B       (A * 3)
```

=> B = ((2+4) * 3) =18

# Macros with parameters *(exercises in week5)*

**#define macroName(parameters) macroValue**

e.g.

#define READ(c, filepointer) (c=fgetc(filepointer))

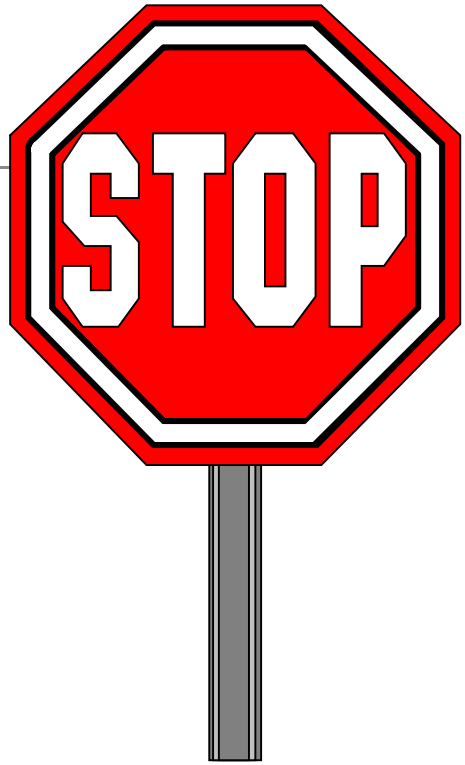
….

if (READ(char, filep1) ==’x’) => if ( (char = fgetc(filep1))==’x’)

<Different than> if ( char = fgetc(filep1)==’x’)

=>>> Use parenthesis a lot!

# Be careful with macros !




Easy to cause unwanted side effect with macro usage

Macros can hinder the readability of the code, while making the writing 'easier'!

```
#define SQR(x)  (x*x)
        SQR(z+1);
=> (z+1*z+1)
```

```
#define SQR(x)  ((x)*(x))
        SQR(z+1);
=> ((z+1)*(z+1))
```

```
#define  EMPTY  (maxUsed == 0)
#define ASSERT   if (!(EMPTY ?  current  == 0 : \
        0 < current && current <=maxUsed)) {\
  fprintf(stderr, ”invariant  failed; current = %d \t; \
  maxUsed= %d\n”, current, maxUsed); \
     exit(1); }
```

EMPTY ?  current  == 0 : 0 < current && current <=maxUsed

Conditional operation

# Predefined macros

__LINE__ row number of this line in source code

__FILE__ name of the source file

__TIME__ compilation time

__STDC__ 1, if compiled according to C standard

```
if (n>10) {  /* error situation */
    fprintf (stderr, " too large value of n  in file  %s
                        row %d ! \n", __FILE__, __LINE__);
     return EXIT_FAILURE;
}
```

# Undefining a macro

#undef PI

If you want to redefine P, it must first be undefined

- Can course problems otherwise!
- Can be undefined on the commmand line to compiler

# External files (header files)

Two formats: affect the search path

\dioms

**#include ”filename”** /* User’s own file, searched first from the local directory*/

**#include <filename>** /*System file, searched first from the system directory*/

Usually with file extension **.h**

# Standard Header Files

**stdio.h** - the basic declarations needed to perform I/O

**ctype.h** - for testing the state of characters

**math.h** - mathematical functions, such as abs() and sin()

+ many more

# Conditional Compilation

= under certain conditions, some parts of the code are left out of the compilation

Used with

**debugging, header files, portable code**

```
#ifdef macroname
  part1
#else
  part2
 #endif
```

```
#ifndef macroName
   partl
#else
   part2
#endif
```

```
#if constantExpression1
   part 1
#elif constantExpression2
   part2
#else
  part3
#endif
```

```
#if constantExpression1
   part 1
#elif constantExpression2
   part2
#else
  part3
#endif
```

- Can be many #elif
- #else can be missing

```
#if  defined (name) …
/*Is name defined?*/
```

```
#error textMessage
```

```
#if defined (__STDC__)
……..
#else
#error ”Something wrong”
#endif
```

# Used in debugging

Conditional compiling is kind of ’opt out’

remove unnecessary parts of the code.

e.g. extra printfs added for debugging

C does not allow multiple levels of comments!

```
# if  0
   conditional part
#endif
```

The same code can be used for debugging and as final product

\dioms

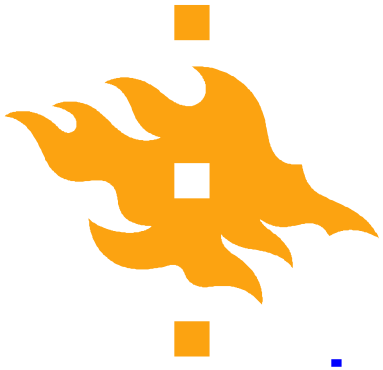
# Debugging code

```
#define DEB   /* just defining */
#ifdef DEB  /*some debugging statements*/
    printf("value of i = %d", i);
#endif
        /*  actual code  */
```

**Command line!**

**gcc –UDEB prog.c** *undefine macro*

**gcc –DDEB prog.c** *define macro*



Example:

```
int main() {
    int i, j;
    printf("Enter two integer values: ");
    if(scanf(%d%d, &i, &j) != 2)
        return EXIT_FAILURE;
#ifdef DEB
  printf("entered %d and %d\n", i, j);
#endif
    printf("sum = %d\n", i + j);
    return EXIT_SUCCESS;
}
```

Advantages?

```
int i, j;
#ifdef DEB
  int res;
#endif
if(
#ifdef DEB
  (res =
#endif
  scanf(%d%d, &i, &j)
#ifdef DEB
  )
#endif
      ) != 2 )
```

```
#ifdef DEB
 {
  switch(res) {
  case 0: printf("both values were
                  wrong\n");
          break;
  case 1: printf("OK first value
                 %d\n", i);
          break;
  case EOF: printf("EOF\n");
          break;
  case 2: printf("both OK\n");
                    break
        }
#endif
 ...
```

More information!

```
int main() {
     const   char SENTINEL = ´.´;
     int       aux, maxi=0;
#ifdef DEBUG
    printf(" Debug on: copying all chars\n");
#endif
    while(1) {
        if ((aux = getchar()) == EOF || aux == SENTINEL) break;
#ifdef DEBUG
    putchar(aux);
    putchar('\n');
#endif
    if (aux > maxi)
#ifdef DEBUG
    printf("Greatest char so far is: %c\n", aux);
#endif
   maxi = aux;
 }
#ifdef DEBUG
   putchar('\n');
#endif
  printf("Greatest char is: %d\n", maxi);
  return EXIT_SUCCESS;
 }
```

# Assert–macro (1)

assert (int lauseke) *(assert.h)*

Writing diagnostic information to stderr

If condition untrue (false, 0), write to stderr condition, source code file name and line number:

*Assertion failed: cond, file filename, line linenumber*

and stop the program execution with abort().

assert-macro is used to check that the program is working as expected: preconditions, post conditions, assumptions about variable values. etc.

Example macros:

```
assert (i>=0)
assert (b*b – 4*a*c >=0
assert (0>=i  && i <size)
```

# Assert–macro (2)

**`assert()`** is usually in use and monitors the program performance.

Operation controlled by macro NDEBUG *( = no debug)* definition. If NDEBUG defined, assert does nothing.

Define **`NDEBUG`** either with #define NDEBUG in code or as command line argument for the compiler gcc –DNDEBUG.

# Example: using assert

```
#include<assert.h>
void open_record(char *record_name) {
assert(record_name!=NULL);
/* Rest of code */
}

 int main(void) {
    open_record(NULL);
}
```

# Macro protection for header files

Conditional compilation can be used to assert that every header file is compiled just one time. Wrap the file content with condition based on macro definition.

In a multi part program the same header file can be easily included **multiple times => compilation error**

Macro named after the header file:

screen.h => use macro name SCREEN_H

Idioms

```
#ifndef SCREEN_H
#define SCREEN_H
/*otsaketiedoston sisältö */
#endif
```

```
#include ”screen.h”
……
#include ”screen.h”
```

Compiled just once!

# Conditional compilation and portability

For developing programs to be used in several different environments:

```
#if IBMPC   /* value of IBMPC */
#include <ibm.h>
#else
#include <generic.h>
#endif
```

```
#ifdef IBMPC /*existence of def.*/
typedef int    MyInteger
#else
typedef long MyInteger
#endif
```

## What does this program print?

```
#define  LOW     -2
#define HIGH  (LOW+5)
#define PR(arg)     printf(”%d\n”, (arg))
#define FOR(arg)   for(;   (arg); (arg)--)
#define  SHOW(x)  x
int main(){
int i = LOW;
int j = HIGH;
    FOR(j)
    switch(j)  {
        case 1:  PR(i++);
        case 2:  PR(j);
                    break;
        default:  PR(i);
    }
  printf (”\n%s\n”,  SHOW(3));
  return EXIT_SUCCESS;
}
```

## So, what does it print?

```
#define  LOW     -2
#define HIGH  (LOW+5)
#define PR(arg)     printf(”%d\n”, (arg))
#define FOR(arg)   for(;   (arg); (arg)--)
#define  SHOW(x)  x
int main(){
int i = LOW;     int i = -2;
int j = HIGH;    int j = (-2 + 5);  /* = 3*/
    FOR(j)         for (; (j); (j)--)
    switch(j)  {
        case 1:  PR(i++);    printf(”%d\n”, (i++));
        case 2:  PR(j);       printf(”%d\n”, (j));
                 break;
        default:  PR(i);      printf(”%d\n”, (i));
    }
  printf (”\n%s\n”,  SHOW(3));
  return EXIT_SUCCESS;
}
```