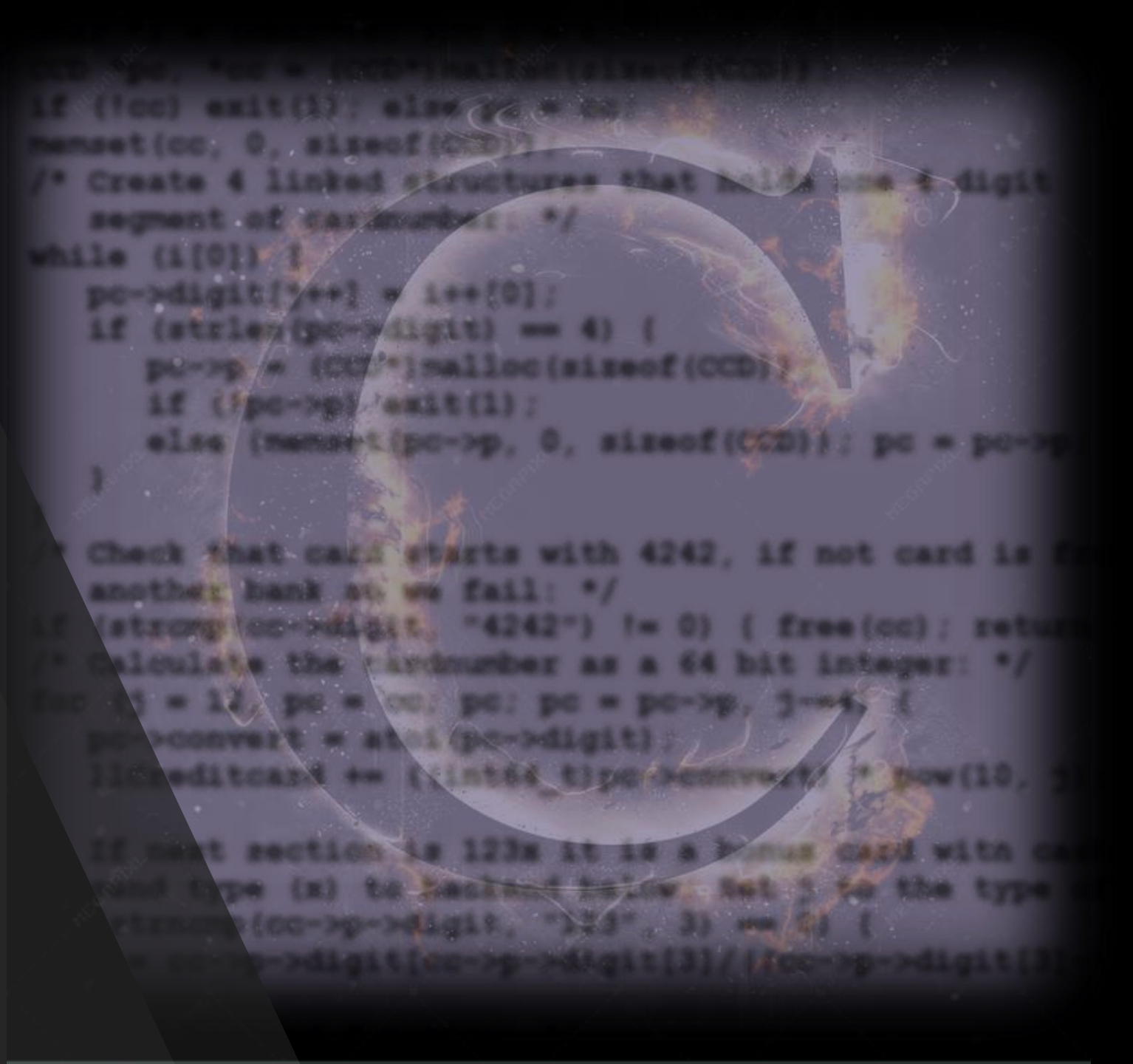
Høyskolen Kristiania

PG3401
C Programmering
for Linux

Lecture 6 (week 07)

Input, output



Course literature

- K&R chapter 7; input and output
- K&R chapter 8; Unix system interface

Recap

- Arrays
- String
- Structs

I/O

- Important to talk to the program!
- Or let the program talk to us!
- Or let programs talk to each other

Without I/O a program can only do math calculations, without I/O we cannot have any user-interactions :-)

Standard I/O

- stdio.h – library for I/O in C
- Flexibility in I/O
- linux + C = powerful!

Input to program

```
int main(int argc, char *argv[])
```

- argc – number of arguments
- argv – array of strings
- Example :
  - $ ./myprog ar1 ar2 ar3
  - argc is 4
  - argv[0] = ./myprog
  - argv[1] = ar1
  - argv[2] = ar2
  - argv[3] = ar3
  - argv[4] = *who knows*

Keyboard and screen

I/O during run-time

- Standard Input – usually the keyboard
- Standard Output – usually the display
- C also supports file I/O

Redirection

- ‘<‘ and ‘>’ can be used to redirect standard output/input to files.
- Output redirection is trivial!
- Example :

```c
#include <stdio.h>

int main(int argc, char *argv[]){
    int a=42;
    printf("Hi this is just an output\n");
    printf("It supports %d formats", a);
    return 0;
}
```

- Example redirection:
  - $ ./a.out > randomFile

Terminal

`int putchar(int)`

- puts a character to the *standard* output
- returns the character or EOF on error

`int getchar()`

- a ***blocking*** input for a character
- returns the ASCII-value of next character from the *standard* input
- returns **EOF** on error

- @ is 64, A is 65, a is 97
- Google ASCII character table
- printf (“as char %c as number %d”, 65, ‘A’);
  → as character A as number 65.

A “problem” with getchar is that it reads 1 character, but it does not return until the user has hit a character (a keyboard key) AND the Enter key – when calling getchar again you will get that Enter key…

A programmer needs to handle this in a way, which is a task I always leave up to the students to figure out because this learning process is very valuable :-)

ASCII character table

I expect that students learned the ASCII table during the first year, if you don't remember repeat lecture 0x01 and 0x02 in TK11 (or similar course in BDC), for your conveniance a copy has been added under "External resources"

www.asciitable.com

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01		SOH
^B	2	02		STX
^C	3	03		ETX
^D	4	04		EOT
^E	5	05		ENQ
^F	6	06		ACK
^G	7	07		BEL
^H	8	08		BS
^I	9	09		HT
^J	10	0A		LF
^K	11	0B		VT
^L	12	0C		FF
^M	13	0D		CR
^N	14	0E		SO
^O	15	0F		SI
^P	16	10		DLE
^Q	17	11		DC1
^R	18	12		DC2
^S	19	13		DC3
^T	20	14		DC4
^U	21	15		NAK
^V	22	16		SYN
^W	23	17		ETB
^X	24	18		CAN
^Y	25	19		EM
^Z	26	1A		SUB
^[	27	1B		ESC
^\	28	1C		FS
^]	29	1D		GS
^^	30	1E	▲	RS
^-	31	1F	▼	US

Dec	Hex	Char
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂

* ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL + BKSP key.

printf()

`int printf(…)`

The first argument is a string that holds the formatting pattern.

- Following arguments should make sense!
- Compiler will (hopefully) warn you!
- The number of arguments is basically your responsibility!
- returns the number of characters printed (rarely any need to check it)
- More intuitive than java’s output mechanism?

snprintf() and fprintf()

snprintf – printf to a string

```
snprintf (char *str, size_t size, char * format, ...);
```

fprintf – printf to file

```
fprintf (FILE *stream, char *format, ...);
```

Formatted string

- The format is the following :
  - %[parameter][flags][width][.precision][length]type
  - parameter – n$ - used to specify the number of argument
  - flags - +, <space>, …, leading zeroes
  - width – *minimum* number of chars
  - precision – *maximum* number after . or chars
  - length – deciding the length
  - type – type of the argument
  - * - to pass the width and precision as arguments

Using "%%" will simply print the %-character

- How do I print % ?

scanf()

`int scanf(…)`

- Much like printf() but for input from standard input
- scanf() needs a *pointer* to where to store the values read
- returns the number of items read
- The return value must be checked for a safe execution!
- matches all the formatted string to the input
  - Example : scanf(“%d, %d”);

const ??

Declaring a function argument as const means that the function will NOT change the value a const pointer points to...

pc++ is allowed, it is allowed to change the pointer

*pc = 'a' is not allowed

```
int MyStrLen (char *pc)
{
   int iLen = 0;
   *pc = 'a';
   if (pc != NULL) {
     while (*pc++) ++iLen;
   }
   return iLen;
}

int MyStrLen2 (const char *pc)
{
   int iLen = 0;
   *pc = 'a';
   if (pc != NULL) {
     while (*pc++) ++iLen;
   }
   return iLen;
}
```

Høyskolen Kristiania

File I/O

Files

- Everything is a file on linux!
- Steps in handling a file:
  - Open the file
  - Perform all the I/O
  - Close the file

- Proper opening and closing of file are important!
- Typically two types :
  - text – usually line oriented
  - binary

See https://www.tutorialspoint.com/c_standard_library/stdio_h.htm

FILE *

- Pointer to the type FILE
  - Which is a file handle!
  - Usually passed into library functions
  - Hence passed as pointer

Opening a file

```
FILE *fopen (const char *filename, const char *mode)
```

- opens file and returns a file handle
- first argument is a file name
- mode can be
  - r Open for read. File must exist.
  - w Open for write. Existing file will be truncated.
  - a Append. Everything written will be appended.
  - r+ Read and write. The file must exist.
  - w+ Read and write. Existing file will be truncated.
  - a+ Read and append. The file must exist.
- suffix ‘b’ for binary files
- returns NULL on failure (check global int errno for reason.)

Writing a file

```
int fputc(int c, FILE *f)
int fputs(const char *s, FILE *f)
int fprintf(FILE *f, ...)
```

Reading from a file

```
int fgetc(FILE *f)
char *fgets(char *buf, int n, FILE *f)
char *fscanf(FILE *f, ...)
```

Using scanf or fscanf with %s is dangerous as the size is not provided, can lead to buffer overflow.

Use fgets instead!

Important:
Buffer overflow is a serious security issue, on the exam the assessor will deduct points for such flaws...

EOF & EOL

In C `'\n'` is end-of-line

EOF is end-of-file – actually a character.

EOF & EOL

```
FILE *f = NULL;
char szLine [160]; // Assume max line length
int iLine = 0;

f = fopen ("test.txt", "r"); // Text read.
if (f != NULL) {
   while (!feof(f)) {
      if (fgets (szLine, sizeof(szLine), f)) {
         printf ("%3d: %s", ++iLine, szLine);
      }
   }
   fclose (f);
}
```

Buffered input/output

the i/o is always buffered

```
int ungetc(int c, FILE *f)
int fflush(FILE *f)
int fseek(FILE *f, long int p, int o)
int fsetpos(FILE *f, const fpos_t *p)
int fgetpos(FILE *f, fpos_t *p)
fpos_t ftell(FILE *f)
void rewind(FILE *f)
FILE *tmpfile()
```

Closing a file

```
int fclose(FILE *f)
```

- All the buffers will be flushed, so an important call
- returns EOF on failure else 0
- Be careful, if error then pretty much everything is lost!

Important:
Failure to close file handles is even worse than a memory leak, handles are typically system wide so in theory you can crash the operating system by using all handles!

Binary files

- Why binary files?
- Example :
  - double values = ‘8901928.7381029’ – 15 bytes as text!
  - communication between programs
- Treat like text files but the input and output are byte streams

- Opening flag is suffixed with ‘b’
  - Example : `fopen(“some/binary/file.mp3”, “rb”);`

- Not strictly platform independent –
  - Big Endian
  - Little Endian

Reading binary files

```
size_t fread(void *buf, size_t size, size_t count, FILE *f)
```

- buf is the place to store your data
- size is the size of each element
- count is the number of elements to be read
- f is the file handle

- returns the number of elements read (should be count)

Writing to binary files

```
size_t fwrite(const void *ptr, size_t size,size_t count, FILE *f)
```

- ptr – data source
- size – size of each element
- count – number of elements
- f - file handle

- Returns number of elements written

Getting the size of a file…

```c
#include <stdio.h>

void main (void)
{
   long lSize = 0;
   FILE *f;

   f = fopen ("adventures.txt", "r");
   if (f != NULL) {
      if (fseek(f, 0, SEEK_END) == 0) {
         lSize = ftell(f);
         printf ("Size of file: %ld\n", lSize);
         rewind(f);
      }
      fclose (f);
   }
}
```

There is no "get file properties" function, move the filepointer to the end of the file and check it's value

Getting the size of a file…

```c
#include <sys/stat.h>
#include <stdio.h>

void main (void)
{
  struct stat sBuffer;
  int iRc;

  iRc = stat("adventures.txt", &sBuffer);
  if (iRc == 0) {
    printf ("Size of file: %ld\n", sBuffer.st_size);
  }
}
```

This function basically does the same as we did by moving the filepointer, for some such a "helper-function" might be more userfriendly?

See also http://www.cplusplus.com/reference/cstdio/

Høyskolen Kristiania

Other I/O

Named Pipe

- Essentially a FIFO
  - First In First Out - queue
- ***unidirectional data channel***
- Simply a file on your machine
- Example :
  - mkfifo myPipe
  - See https://linuxprograms.wordpress.com/tag/mkfifo/
  - https://www.softprayog.in/programming/interprocess-communication-using-fifos-in-linux

Høyskolen Kristiania

Working with pipes

- Just as any files!
- Blocking I/O
  - Must be handled at both ends
- Typically used for using third-party programs and asynchronous communication between applications

Høyskolen Kristiania

Exam example – H21:03

Exam task (2021 exam)

- Download the file PG3401_Lecture_06_Hjemmeeksamen-14dager-H21_OPPG3.zip from Canvas, it contains source files

kasseapparat.c	6.520	1.776	C Source	14/02/2022 12:...	8167FDCE
kasseapparat.h	1.897	695	C/C++ Header	09/02/2022 14:...	B2604E0C
main.c	10.247	2.888	C Source	09/02/2022 14:...	0077173D
pgdbglog.c	4.821	1.452	C Source	03/02/2022 09:...	2FE8F6E8
pgdbglog.h	1.193	528	C/C++ Header	03/02/2022 09:...	898A2438

This zip file also includes a "debugger" code, we will cover that in lecture 7

Høyskolen Kristiania