



Denne forelesningsøkten vil bli tatt opp og lagt ut i emnet i etterkant.

Hvis du ikke vil være med på opptaket:

	La være å delta med webkameraet ditt.
	La være å delta med mikrofonen din.
To: Marianne Sundby (Privately) Type message here...	Still spørsmål i Chat i stedet for som lyd. Hvis du ønsker kan spørsmålet også sendes privat til foreleser.



Høyskolen
Kristiania

PG3401 Programmering i C for Linux

Bengt Østby

Exercises and the road ahead

An easy start to a complex course these two weeks. Important is that everyone now has an environment where you can:

- Use Linux to work
- Compile programs using gcc
- Use makefile

Without being able to code you will not be able to learn, and it will be too late to set up an environment for the first time in November...

Remember 200 hours minimum for a 7.5 studypoints course; which translates to 10+ hours of self-guided work to each lecture.

A warning; a lot will be covered today, it might feel overwhelming at first, but after working 20+ hours in a code editor it should start to make sense 😊

Interest in getting solution to exercises?

From today's lecture (and onwards) I have solution files to the exercises. We have some ways of dealing with that 😊

- I upload the solutions with the exercises
- I upload solutions the following week
- I give you the solutions on November 3rd, to prepare for exam
- I only give them to the student supervisors

What do you think will give you the best learning? Remember that being given the answer will not let you learn as much as finding the answer.

Recap

Linux OS

Terminal interface

Files and folder structure

Compilation tools

- gcc
- make
- cmake

Debugging tools

- gdb
- valgrind

Makefile

target_file : *list of dependent files*
produce **target_file** from *list of dependent files*

program : program.c
gcc -O2 program.c -o **program**

gnuredutt.zip : file1.txt file2.txt file3.txt
zip **gnuredutt.zip** file1.txt file2.txt file3.txt

gnuredutt.zip : file1.txt file2.txt file3.txt
zip **gnuredutt.zip** \$^

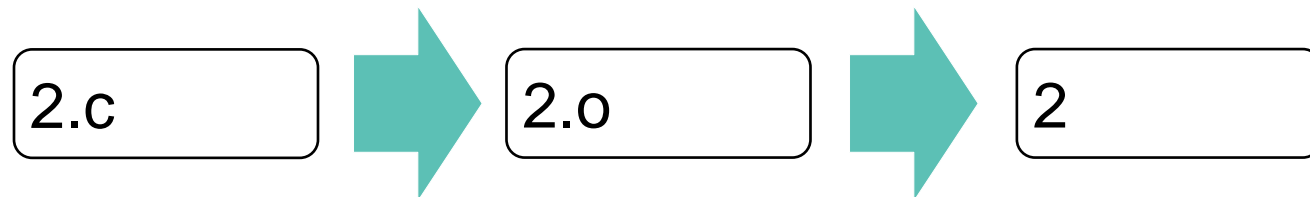
Makefile

```
gnuredutt.zip : file1.txt file2.txt file3.txt  
    zip gnuredutt.zip $^  
    cp gnuredutt.zip ./backup
```


More about makefiles...

```
CC = gcc
CFLAGS = -g -O0

1: 1.o
    $(CC) -o 1 $(CFLAGS) 1.o
2: 2.o
    $(CC) -o 2 $(CFLAGS) 2.o
3: 3.o
    $(CC) -o 3 $(CFLAGS) 3.o
clean:
    rm -f *.o 1 2 3
```



```
%.o: %.c $(DEPS)
    $(CC) $(CFLAGS) -c -o $@ $<
```

More about makefiles...

```
#  
# Generic makefile for compiling and linking more than one source file  
#  
  
OBJS = hello.o number.o # List object files here  
DEPS = number.h makefile  
# CFLAGS = -O2  
CFLAGS = -O0 -ggdb  
  
%.o: %.c $(DEPS)  
    gcc $(CFLAGS) -c -o $@ $<  
  
hello: $(OBJS)  
    gcc -o $@ $^ $(CFLAGS)  
  
.phony: clean  
clean:  
    rm -f *.o
```

More about makefiles...

```
CC = gcc
CFLAGS = -g -O0

# Implicit rule - no need to type it in ...
%.o: %.c $(DEPS)
    $(CC) $(CFLAGS) -c -o $@ $<

1: 1.o
    $(CC) -o 1 $(CFLAGS) 1.o
2: 2.o
    $(CC) -o 2 $(CFLAGS) 2.o
3: 3.o
    $(CC) -o 3 $(CFLAGS) 3.o

clean:
    rm -f *.o 1 2 3
```



History of C

BCPL (Basic Combined Programming Language)
1969-1973 by Dennis Ritchie at AT&T Bell labs

Standards :

- K&R
- ANSI C89 [American National Standards Institute]
- ANSI C99
- ANSI C11

Background

- Often fastest
- Systems programming
- OS kernels, drivers
- Scientific simulations with performance requirements
- The code is compact – allows it to run on small devices like mobiles, embedded devices
- Modular – functions, structs

Cross Platform

- Extremely Portable
 - Write once, compile anywhere
- Possible to create machine specific code
- A good replacement for assembly

Procedural programming vs object oriented programming

- Emphasis on procedures unlike OOP which focuses on behavior
- Data structures [struct, union] – much like classes without methods
- Programming is writing procedures to do something with these structures
- Recursion – calling procedure within itself

Hello world!

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

```
int main(int argc, char *argv[]){  
    //My first C program  
    printf("Hello world!\n");  
    return 0; //All OK  
}
```

```
-$ gcc -O2 -Wall -Werror -std=c11 hello.c -o hello
```

```
-$ ./hello
```

VOILA!

#include

Order preprocessor to include other source file

It simply reads the named file, then continue in your file

< > implies the system include folders

- /usr/include
- /usr/local/include
- Folders given by "-i" option on **gcc** command line.

You can also include your own files by using “ “

- #include "myHeader.h"
- #include "/home/vamsi/myHeader.h"
- #include "../inc/myHeader.h"

Main

- The main entry point of your program
- C99 requires it to return an **int** (newer standards can have **void**)
- But you can return void depending on the compiler
- Return 0 for success and an error-code otherwise
- It can help in scripts with multiple executables

Arguments to main

int main(), int main(int argc, char* argv[]) – does not enforce any number of arguments or their type

int main(void) – implies that it must be passed with no arguments

If used

- argc – Number of arguments
- argv – array that holds arguments in **string** format

Example : \$./hello a1 b2

- argc = 3
- argv[0] = './hello'
- argv[1] = 'a1'
- argv[2] = 'b2'

printf

Prints out text to stdout

print format string

Some example formats:

- %d – int
- %f – float
- %s – string
- %x – hexadecimal number
- Check man page

```
./Hello "%s%s%s"  
Printf(argv[2]);
```

```
Printf("%s", argv[2]);
```

'\n' is used for line break

```
int a=3;
```

```
printf("a is: %d\n", a);
```

gcc inputs

‘-o’ to inform where to put the output file

‘-std=c11’ to ask the standard that should be used

- careful : C11 is 2011 and the new standard
- OpenCL still uses C99

‘-Wall’ : Show all warnings

‘-Wextra’: Show even more warnings

‘-O2’: Optimize.

‘-O0’: Must be turned off for debug.

‘-Werror’ : Make all warnings as errors

- sometimes it is a bad idea

Commenting is just like Java

Block commenting - `/* */`

- Example :
- `/* Hello this is
and example of
multiline block
commenting */`

Line commenting - `//`

- Example :
- `// This is how you can comment
// multiple lines`

Style guidelines

Good for readability!

Tab is usually 3 spaces

Limit the length of the line to 80 characters

Line breaks are better with an operator at the end

- Example :

```
int iNumberOfDaysUntilTheEndOfTheUniverse = 7;
```

```
iNumberOfDaysUntilTheEndOfTheUniverse = iNumberOfDaysUntilTheEndOfTheUniverse -  
    iNumberOfDaysLeft + 42;
```

For more guide lines read **Google code guide**

Variable scope

You can have global variable and local variables

Global : declare it once and it is available for all the code that comes next (*not always a good idea...*)

Local : Limited scope usually in the functions

Variable types

Declaration is similar to java, but some types have architecture-dependent definitions.

- char
- short
- int
- long
- long long
- float
- double

You can have signed and unsigned

inttypes.h

Makes your code more portable

unsigned integer: uint8_t, uint16_t, uint32_t

signed : int8_t, int16_t

Read more from the standard documentation

C has no native boolean type

```
#include <stdbool.h>
```

- 'bool', 'true', 'false'
- Unsafe type
- '0' is false, everything else is true

Common: using int with '1' as true and '0' as false

Similar to Java

while

for

if/else/else if

switch

do while

No iterators in C! (*for each...*)

+, -, *, /, +=, %, &&, &, =, ==,

!=, <, >, <=, ^, !, ~ etc

```
unsigned Int counter = 0;
```

```
Counter = 0;  
while (counter > 0) {  
    Ddd();  
    Counter--;  
}
```

```
For (counter = 5; counter > 0; counter--) {  
    ddd();  
}
```

```
Counter = NumberOfWowPlayers();  
Do {  
    ddd();  
    counter--; // = 0xffffffff  
    counter = counter - 1;  
    counter -= 1;  
} While (counter > 0);
```

Functions are used to make the program modular

A Function is declared with a name, arguments and return type

- example: `int sum(int a, int b){ return a+b; }`

The function must be declared before it is used, but not necessarily implemented.

- example :

```
int sum(int a, int b);
```

```
int main(int argc, char *argv[]){  
    int result;  
    result = sum(9,10);  
    return 0;  
}
```

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

Recursion is supported in C

Main is also a function!

getchar()

Simplest input

Reads a char from keyboard

returns a character

```
char c = getchar();  
printf("%c is the character", c);
```

Blocking input – waits forever until a char is seen

Static Array

```
int sorted[10]; //integer array
```

```
int length = sizeof(sorted);
```

sizeof() returns bytes

Stored on the stack if it is not declared with static/ as a global array

scanf(format string)

```
char buf[80];
```

```
int num;
```

```
scanf("%s", buf);
```

```
scanf("%d", &num); // Reference to num
```

```
scanf("%d %s", &num, buf);
```

scanf() returns the number of elements read

Type casting

Casting can be “implicit” or “explicit”

```
char c = 'N';  
int d = c; //implicit cast  
float f = (float) d //explicit cast  
printf("%f", f); // output : 78.00000
```

Casting can be unsafe since C lets most casting through

Pointers can be casted too – more on this later.

Datatype

The **datatype** of an object in memory determines the set of values it can have and what operations that can be performed on it

C is a *weakly* typed language. It allows implicit conversions as well as explicit casting.

Example : int, char, long etc.

Sizes are different depending on platform

Operators

Operators specify how an object can be manipulated (eg., numeric vs. string operations).

Operators can be

- Unary : ++, --, ~
- Binary : +, -, *, /, %
- Ternary : ?:
 - Example-> result = (condition)? (result1):(result2)

```
int bigger(int a, int b){  
    return (a>b) ? a:b;  
}
```

Expressions

An **expression** in a programming language is a combination of values, variables, operators and functions

Example :

- $y = x + 2;$
- $y = x * 3;$
- $y = x \% 4;$

Variables

A **variable** is a named link/reference to a value stored in the system's memory or an expression that can be evaluated.

Example :

- `int *a;`
- `int x,y;`
- `char z;`

`a, x, y, z` are all variables

Variable names

Variable names can contain letters, digits and _

Variable names can *not* start with digits

Keywords cannot be used as variable names

Variable names are case sensitive.

Examples:

- Allowed : int x, X, radiusX, _radius, _circle1, _circle2
- Not Allowed: int x\$, xyz\$, 8low, 8high
- Better: int iSze, iCounter;

Data type categories

Numeric (int, float, double)

Character (char)

User defined (struct, union)

Numeric data types

short

int

long

float

double

Signed Vs Unsigned

Range depends on signed and unsigned

Sizes of variables

Sizes are dependent on compiler and machine

Guaranteed :

- $\text{sizeof}(\text{char}) < 2 \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq 4 \leq \text{sizeof}(\text{long})$
- $\text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{float}) \leq \text{sizeof}(\text{double})$

These are important for overflow.

What are the sizes on your machine?[exercise]

Overflow

Limited number of bits to store the values

Unsigned has bigger range:

- 8 bit : 0 to 255
- 16 bit : 0 to 65535
- so on

Signed has lower range because of usage of one bit for sign:

- 8 bit : -128 to 127
- 16 bit : -32,768 to 32,767
- so on

When you add two integers – C doesn't check for overflow

- Ex : `int a = 32767, b = 32767;`
 - `int c = a + b ;//Oh well! this will go to hell!`

Overflow can be expensive – Ariane 5 rocket!

<https://www.bbc.com/future/article/20150505-the-numbers-that-lead-to-disaster>

Literals

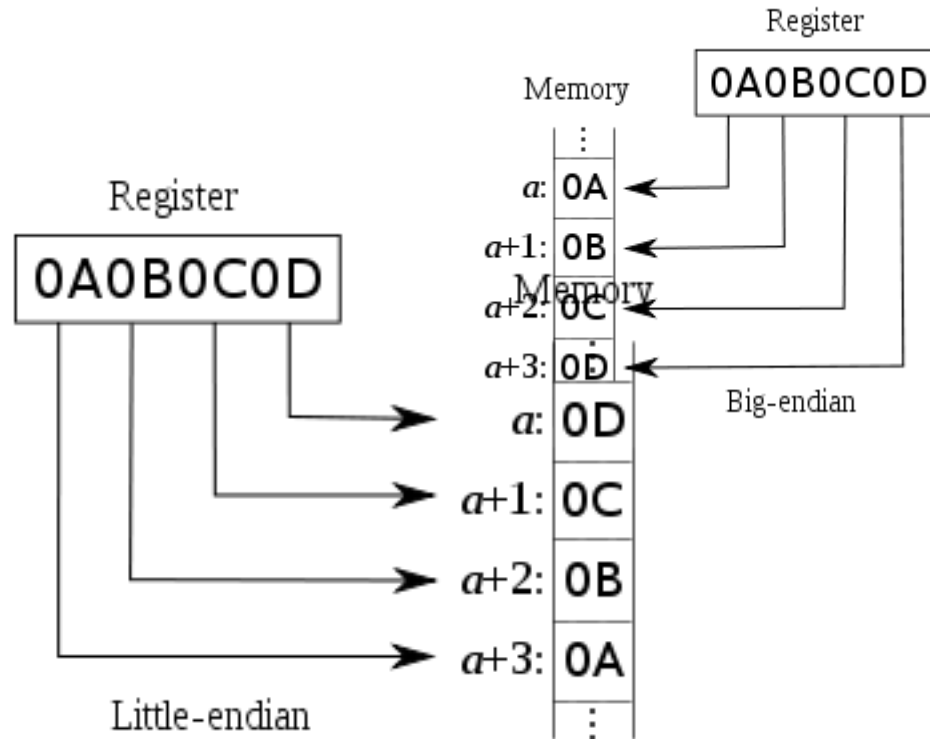
Literals are literal/fixed values assigned to variable or used directly in expressions

Examples:

- Integer : 4, 5UL, 0xC, 071
- Floating point : 3.1415, 3.14F, 2.0, 3.1415265359L
- Character : 'A', '\x41', '\0101'
- String "this is a string"
- Enumeration : enum eAnswers {NO, YES, MAYBE};

Endian-ness (mess...)

- Little Endian – when lower memories hold low significant bits (Intel x86 etc.)
- Big Endian – When lower memories hold most significant bits (PowerPC, 68K, 88K and networks)



Operators

Arithmetic operators

Relational operators

Logical operators

Bitwise operators

Assignment operators

Arithmetic operators

Addition - '+'

Subtraction - '-'

Multiplication - '*'

Division - '/' => quotient in integer case

Modulus - '%' => of course usually used on int

Increment - '++'

Decrement - '--'

Relational operators

Greater than : >

Greater than or equal to: >=

Lesser than : <

Lesser than or equal to : <=

Equal to : ==

Not equal to : !=

Avoid using ==/ != for float/double, instead use an expression

- $\text{abs}(f1-f2) < e$

All return 0/1 !

Logical operators

And - &&

Or - ||

Not - !

Xor - ^

Usage of x^1 for alternative on/off flag

Caution :

The expressions are evaluated from left to right, short circuited.

Example : `(4!=5) || ((c=getchar())=='n')`

This will never wait for you to type n

Bitwise operators

AND - &

OR - |

XOR - ^

Left shift - <<

Right shift - >>

Blocks

```
{  
    int a= 20;  
    //This is block1  
    {  
        //This is an inner block  
        printf("%d", a);  
    }  
}  
  
{  
    //This is a different block  
    printf("%d", a); //error a undefined  
}
```

Blocks

Compiled as a single unit

Allows variable declaration

A variable in a block is accessible over the entire block including all the inner blocks

If a variable using the same name is defined in the inner block, then the first variable's scope is not valid in the inner block. It will be replaced with the new variable.

```
{  
    int a = 20;  
    //a here is 20  
    {  
        //a here changes to 40  
        int a = 40;  
        printf("%d", a);  
    }  
    printf("%d", a); //a here is 20  
}
```

Conditional expression

An expression or a collection of expressions that evaluates to false/true

Expression = zero implies false, non zero implies true

Example :

- $(A > 5)$
- $(a < 3) \ \&\& \ (b > 5)$

Control structures

if

for

while

do - while

switch

break

continue

if

if , else if, else

Same as java

Allows for empty statements, blocks and single lines.

Switch

```
switch (iVar) {  
    case 0:  
        statements;  
        break;  
    case 1 :  
        statements;  
        break;  
    default :  
        statements;  
}
```

Works for int and char!

for

Similar as Java

for (initializations; conditions; processing) ...

```
for (int i=0; i<10; i++) {  
    ...  
}
```

```
for (;;) {  
    ...  
    if (...) break;  
}
```

```
for (;;) ;
```

while

```
while(condition) {  
    //do stuff here  
}
```

Doesn't guarantee an execution of stuff!

The condition is evaluated first and then stuff are executed

do-while

Enables for at least one guaranteed execution

```
do{  
    //do stuff here  
}while(condition)
```

Sequencing can lead to confusing behavior...

```
a = a++; // Nothing happens;...  
a = ++a;  
a++; ++a;
```

```
#include <stdio.h>  
  
int a() {puts("2"); return 2;}  
int b() {puts("3"); return 3;}  
  
void main(void) {  
    int x=a() + b();  
    printf("%d\n",x);  
}
```

Practical

Fetch <http://www.eastwillsecurity.com/pg3401/leksjon3.zip> (or Canvas)

Extract the folder and read the instructions

File names should be ex1.c, ex2.c, ex3.c and then work in this folder for today.

We will follow this practice for the rest of the course to keep your code organized.

Today's exercises:

- Size of variables
- Increment/decrement
- Write a program that output if a number is odd/even
- 99 bottles of beer
- Conversion between if \Leftrightarrow switch, for \Leftrightarrow while \Leftrightarrow do-while
- Fibonacci series
- Quadratic equation solving – why does it fail?

Quadratic equation

The general quadratic equation is

$$ax^2 + bx + c = 0.$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$