



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

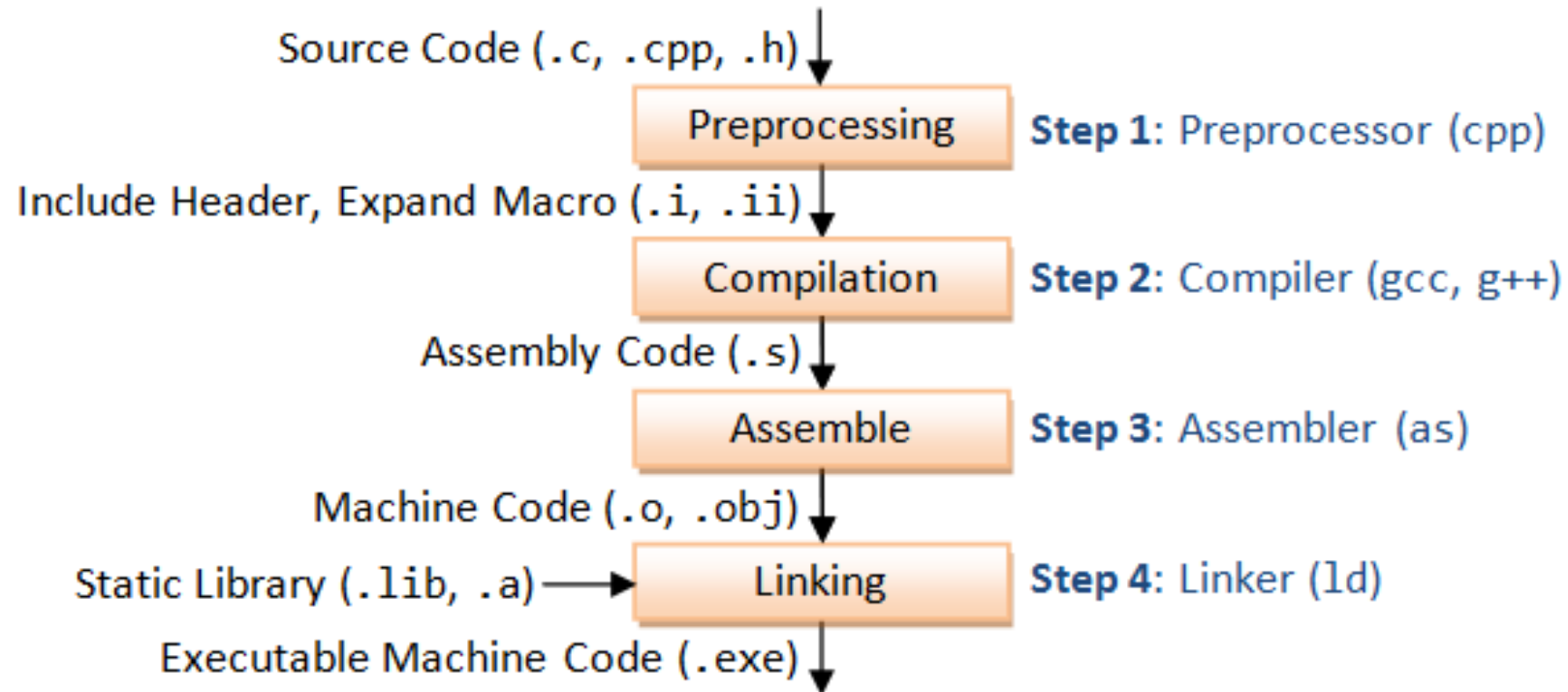
PG3401 Programmering i C for Linux

Bengt Østby

Recap

- I/O
- terminal
- pipe
- files:
 - text
 - binary

Life of C



Preprocessor

- First things to be handled by compiler
- Used to control what is compiled and how!
- Example :
 - `gcc -E hello.c -o hellopre.c`

#include

- Preprocessor to include other source
- It simply copies the sources to the current file
- < > implies the system include folders
 - /usr/include
 - /usr/local/include
- You can also include your own files by using “ “
 - #include “myHeader.h”
 - #include “/home/raakje/myHeader.h”
 - #include “../inc/myHeader.h”

Other preprocessor

#define - #undef

#if - #elif - #else - #endif

#ifdef - #else - #endif

#pragma – set compiler *behaviour*

variable and function scopes

- Local
- Global
 - External – across files (default)
 - Internal – one file only (static)
- Scope precedence!
- Use for information hiding to replace “private” and “public”.

Storage classes

- auto, default for variables in functions
- static, storage. Can be both global and in functions
- register, no address of (&) hints at using register
- volatile, forces NO optimization!

"const" keyword blocks changing value programmatically

- Used for code safety as well as optimization.
- `int i;`
- `const int ci = 123;`
- `const int *cpi; // declare a pointer to a const..`
- `int *ncpi; // ordinary pointer to a non-const`
- `cpi = &ci;`
- `ncpi = &i;`
- `cpi = ncpi; // this is allowed`
- `ncpi = (int *)cpi; // this needs a cast`
- `//because it is usually a big mistake, see what it permits below.`
- `*ncpi = 0; // now to get undefined behaviour...`

- Source: http://publications.gbdirect.co.uk/c_book/chapter8/const_and_volatile.html

const can be applied both to a pointer and to its contents

- `const int *constInt; // A pointer to a const int`
- `int *const constPointer// A const pointer to a normal int`
- `/*A const pointer to a const int*/`
- `const * const int constIntconstPointer;`
- A const structure has only const members.

Functions

- 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!

function scope

- { }
- Must be declared before first usage!
- You can keep the internal state by using static keyword!
- Declaration Vs Definition
- Declaration declares existence of
- Definition makes
- Can have multiple declarations, only one definition

extern

- Normal variables are both declarations AND definitions
- extern is just a declaration – not definition!
- Keyword to re-declare a variable in other files
- This will be externally linked
- You say “this variable exists *somewhere*”
- Usage :
 - In 1.c – extern int count;
 - in 2.c – int count;

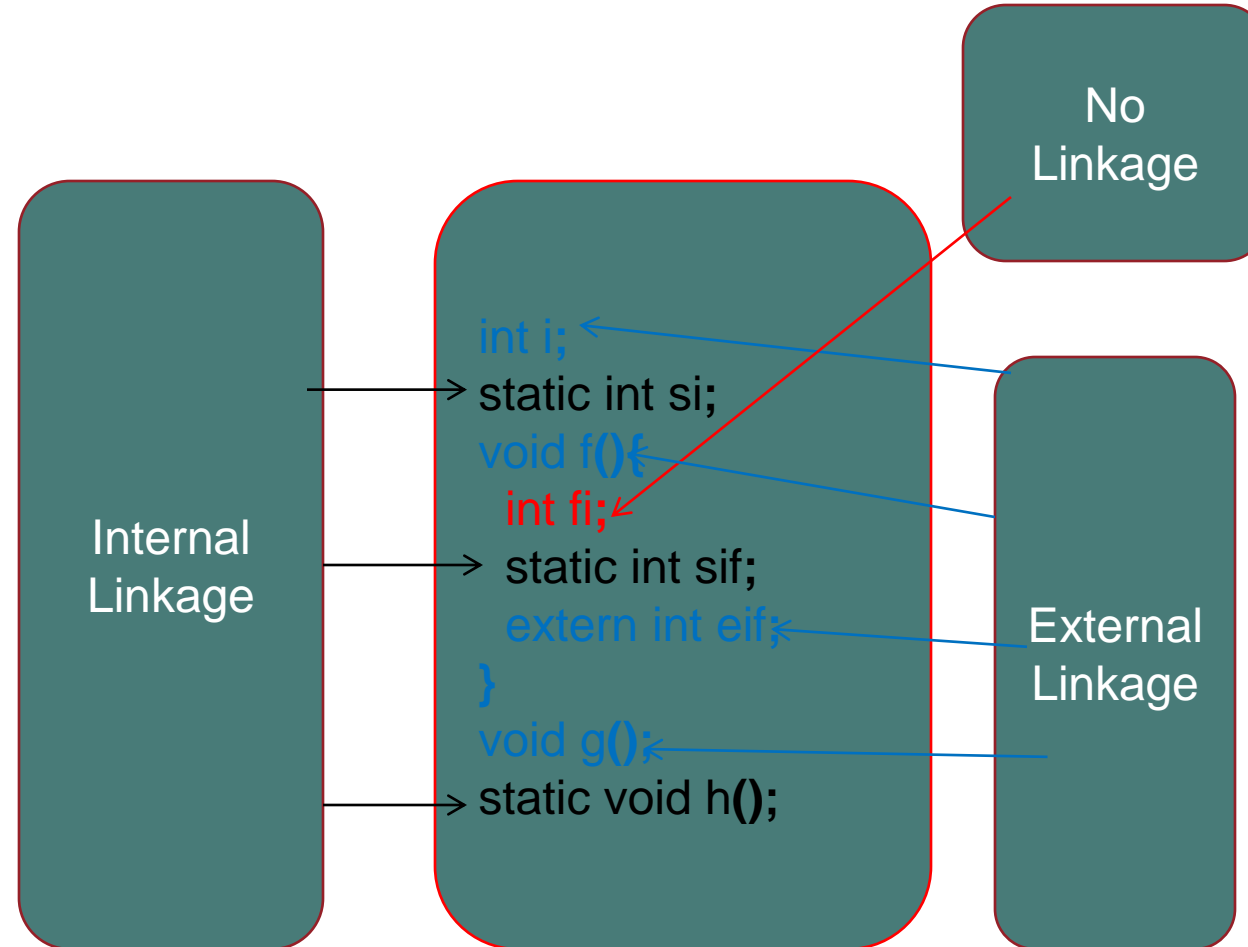
global variables

- Variable scope – local, global
- To declare global variables – they must be placed outside all functions, including main.
- The variable is declared from the first instance it is defined
- Example:

```
int i;  
void some () {  
    /*blah;  
    */  
}  
int j;  
  
void other () {  
    /*otherBlah  
    */  
}  
  
int main () {  
    /*  
    More blah  
    */  
}
```


Linkage

- internal
- external



Modularity

- interface \Leftrightarrow implementation
- header \Leftrightarrow source

header files

- Contains the interface
- Often written keeping the user of functionality
- Example :

```
/* Write formatted output to stdout.
```

```
    This function is a possible cancellation point and therefore not  
    marked with __THROW.  */
```

```
extern int printf (const char *__restrict __format, ...);
```

source files must #include their own header

- Contains the implementation
- Written by the programmer implementing the library
- Example :

```
printf (const char *format, ...)  
{  
    va_list arg;  
    int done;  
  
    va_start (arg, format);  
    done = vfprintf (stdout, format, arg);  
    va_end (arg);  
  
    return done;  
}
```

Headers should have "include guards"

```
#ifndef __HEADER_H__
```

```
#define __HEADER_H__
```

```
//Header stuff goes here
```

```
#endif //__HEADER_H__
```

Source files can be compiled together directly

- `gcc a.c b.c -o prog`

Makefiles are the traditional way of joining multiple files.

```
# Generic makefile for compiling and linking
# more than one source file
#
TARGET = ex4
OBJS = $(TARGET).o md5.o # List object files here
DEPS = md5.h makefile
CFLAGS = -O2

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

$(TARGET): $(OBJS)
    gcc -o $@ $^ $(CFLAGS)

.phony: clean
clean:
    rm -f *.o $(TARGET)
```

Making and using static libraries:

- Making a static library

```
gcc -c obj1.c obj2.c obj3.c -O0 -g  
ar rcs libname.a obj1.o obj2.o obj3.o
```

- Linking a static library

```
gcc -o program obja.o objb.o libname.a
```


Summary

Summary

- Preprocessor
- Scopes – controlling them
- functions

Starting to get more difficult?

- Remember; coding in your cohort and with me is the best way to learn! 😊
- Show code to each other, discuss how it could be improved
- Ask me and TAs how to improve your code further
- Code, code, code, code... 😊

Exercises

wget <http://www.eastwillsecurity.com/PG3401/leksjon7.zip>

+

Make the debugger I will show next 😊

**That's the theory
-
let us combine all this**

A debugger (of sorts)

Single-step debugger:

A (good) debugger will let you run your program line by line (single-step debugger), and view all variables, memory and other state as you go, some even let you modify both code and variables during execution.

Just running the code in a debugger will help a lot if it crashes – it will show you the exact line of code it crashed at.

Problem:

- What if you can't run a second application (the debugger)?
- What if you don't have a screen, or a keyboard?
- What if the problem appears only in production, at some of your customers? (You cannot ask an end-user to install a debugger...)
- What if all of this is on an embedded device with limited features?
- What if it happens during boot, in your system driver, on Ring-0?

A debugger (of sorts)

Solution:

- Insert the following line between EVERY line in your code:

```
printf("\nC PROGRAMMERING ER GØY");
```

- If you see that line 4 times in your console, you know the program is crashing before the 5th line 😊

New problem:

What if your application consists of millions of lines of code?

- We do the same, just smarter!

A debugger (of sorts)

Real-world solution:

- Norman Anti Virus used a program called ELOGGER, that stood for Elling's Logger, the developer that created it named it after himself...
- (Not any worse than Linux, just so you remember that)
- This was a debug tool that would receive debug output live from all applications from Norman, and the tool could save the data to a txt file that could be sent to support. This saved us developers on so many cases!
- This was replaced in Norman Security Suite v10 by SfLogger - (Norman) Safeground Flightrecord logger
- Saved encrypted and signed "Blackbox files", with the .norman_bf extension
- The file extension BF stands for Bengt's File (yup, the developer that created it named it after himself!)

A debugger (of sorts)

Let us make such a logger, but a bit simpler – we will log to a file

You will thank me for this exercise when the refrigerator you wrote the micro controller for keeps crashing all around the world... - And you can SSH to the crashing refrigerators and extract the debug files that you decided to build into the system, using the code from this course! 😊

Task 1: Add 2 new files to your makefile

To any of your exercise solutions (that uses a makefile) add 2 new files

I would like to call them:

pgdbglog.h and pgdbglog.c

Create a function in the C file:

```
void PgDbgLogger(unsigned long ulErrorType) {  
    return;  
}
```

And the prototype for the same file in the header file.

Task 2: Call the debug function from your source

We want to call this function from our source code, this task ensures that we understand include files, and that the makefile works correctly.

Add this to the top of the main source file:

```
#include "pgdbglog.h"
```

Call the function at least 3-4 times in your program (note that you should be VERY carefull about calling it in a loop that runs more than 10 times – it is better to add it once before a for() and then once after the loop has ended...)

```
PgDbgLogger(1) ;
```

Task 3: Add INCLUDE GUARDS to the header file

If you don't already, create a “project” include file; include.h for instance, that holds a function prototype for your exercises function (that is called from main).

Now; add the #include “pgdbglog.h” statement to both that header file and to the C file, try to compile and you might get an error because the function prototype is defined twice.

To solve this add include guards to pgdbglog.h:

```
#ifndef __PGDBGLOGGER_H__
```

```
#define __PGDBGLOGGER_H__
```

```
void PgDbgLogger(unsigned long ulErrorType);
```

```
#endif //__PGDBGLOGGER_H__
```

Task 4: Add code that logs to a file

Extend the functionality of the logging function to print to a file:

- Create a static local variable `fLogFile` that opens the file (as we did last week)
- For now the file can be called `debug.txt` and it can be truncated on open
- Also create a static local variable that counts from 1 and the number of times it is called, we will use this as the `linenumber` in the file
- We don't close the file, we leave that to the OS – since this breaks with Best Practice it warrants a comment in the source file of why we chose to do this

```
fprintf(fLogFile, "%04i: error type is %i", iCallCounter, ulErrorType);
```

Task 5: Make it look like printf

We want to use printf style parameters, in other words we want to have a dynamic number of arguments...

How do we do that?

One of the coolest features in C/C++: `va_args`

Va_args

We want to use printf style parameters, in other words we want to have a dynamic number of arguments...

See <https://linux.die.net/man/3/vsnprintf> for man page for **vsnprintf & vfprintf**. See also Section 7.3 on page 155 in The C Programming Language.

```
#include <stdarg.h>
```

```
void PgDbgLogger(unsigned long ulErrorType, const char *pszFormat, ...) {  
    va_list vaArgumentPointer;  
    /* ... */  
  
    va_start(vaArgumentPointer, pszFormat);  
    vfprintf(fLogFile, pszFormat, vaArgumentPointer);  
    va_end(vaArgumentPointer);  
  
}
```

Task 5: Make it look like printf

You can expand the complexity of this as much as you prefer, the hint there is using `vsnprintf` function into a static array.

```
va_list vaArgumentPointer;  
char szOutputString[256] = {0};  
char *pszType = (ulErrorType==1)?"Error":"Debug";  
/* ... */  
  
va_start(vaArgumentPointer, pszFormat);  
vsnprintf(szOutputString, 256 - 1, pszFormat, vaArgumentPointer);  
va_end(vaArgumentPointer);  
  
fprintf(fLogFile, "%04i: %s %s", iCallCounter, pszType, szOutputString);
```


Task 6: Add line number to the function and to output

When we pass a few hundred thousand lines of code, we need to know where the error came from.

To the rescue comes two macros: `__FILE__` and `__LINE__`

By adding line number and filename to the log function:

```
void PgDbgLogger(unsigned long ulErrorType, int iLine, const  
char *szFile, const char *pszFormat, ...);
```

It can then be called as:

```
PgDbgLogger(1, __LINE__, __FILE__, "%s", "C er gøy");
```

Task 7: Need a shorter name so we use it more

But now the code for printing a debug line is sooooo long, who will ever bother to call that function now? Well, this is where MACROS come in!

In the header file we define some “shortcuts” with `#define`

```
#define pgdebug(...) PgDbgLogger(0, __LINE__, __FILE__, __VA_ARGS__)  
#define pgerror(...) PgDbgLogger(1, __LINE__, __FILE__, __VA_ARGS__)
```

And in our code we can now write the simple:

```
pgdebug("Test");  
pgerror("malloc failed to allocate %i bytes", ulBytes);
```

Task 8: Config files are nice

We want to control the output from these functions.

One way to improve speed is to change the MACRO for debug to:

```
#define pgdebug(...) if (glob_bTraceEnabled == 1) {  
PgDbgLogger(0, __LINE__, __FILE__, __VA_ARGS__);}
```

And we create a global variable that we can set on the first call to the PgDbgLogger function. (Notice the glob_ prefix to the variable – it is just a name, but it is good practice to show clearly that this is a global variable. In general “we” hate global variables...)

Experiment with reading a text file with some settings, it can be as easy as a file that has either the value 1 or 0, or it could be a complex XML file – you chose 😊 Read the settings file at the same time as you create the debug file, or create a new function PgDbgInitialize(void).

Task 9: Dynamic filename for logfile

The final thing we need to do is determine the filename of the debug file.

We want a new file for every time the application runs, and the easiest way to accomplish that is using the time functions to create a “unique” filename.

```
#include <time.h>

/* ... */

char szFileName[256] = {0};
time_t tTimeAndDate = 0;
if (fLogFile == NULL{
    tTimeAndDate = time(NULL);
    snprintf(szFileName, 256 - 1, "debug_%i.txt", tTimeAndDate);
    // More userfriendly possible, this is num secs since 1970...
}
```

Task 10: We need a library for this...

Now we have a good enough debug tool

Copy this .c and .h file to any project, include them in the makefile, and you are set

BUT – it would be great to not have to copy it, it would be better to have one “project” for the debug tool and built it is a LIBRARY. Then this library file could be included from all the makefiles.

If you are done with task 1-9, try to figure out how to build a LIBRARY, for that extra few points that can earn you an A in this course. If task 1-9 was difficult enough for you, then you don't have to (to pass) 😊

More advanced?

For some applications this will do...

For embedded this might actually be the best approach, just note that writing to a file is a costly operation – but it does survive a crash!

Also note that this has no cleanup (we actually ignored that in production in Norman, but on some installations the debug files could be quite large...) Be especially aware of this on an embedded system!

For other uses you might want to set up a Daemon for this (or a Service on Windows) that can read on a pipe or shared memory and write data to a file only if needed. (We will learn about inter-process communication later in this course.)

(For kernel mode drivers on Windows the operating system features this, function is called DbgPrint and can be viewed by installing SysInternal's DebugView.)