Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

CS Basics
15) Compiling a C prog.

E. Benoist & C. Grothoff

Fall Term 2018-19

Computer Science Division

Compiling a C program

Compiling

- **Compilation needs a lot of information**
  - Standard libraries
  - Own libraries
  - Instructions for compilation
- **Compilation instructions**
  - Compilator definition
  - Compilation Warnings
  - Compilation Optimization
- **Linking instructions**
  - Which standard library to link with (-lm ...)
  - Which local library to link with
  - ...
- **Need for a Makefile**

Example of a small program

Example of Program

- **One Hello World**
  - hello.c, a library
  - hello.h (is protected using a preprocessor directive from being inserted many time)
  - main.c the main file
- **hello.c**

```
#include <stdio.h>
#include <stdlib.h>
void Hello(void)
{
        printf("Hello World\n");
}
```


Example (Cont.)

- **hello.h**

```
#ifndef H_GL_HELLO
#define H_GL_HELLO

void Hello(void);

#endif
```

- **main.c**

```
#include <stdio.h>
#include <stdlib.h>
#include "hello.h"
int main(void){
        Hello();
        return EXIT_SUCCESS;
}
```


Makefile

Makefile

- **Syntax**

```
target: dependences
        commands
```

  - Commands must be preceded with a single tab
- **A target is chosen**

```
make target
```

  - Dependences are analysed
  - Needed commands are executed
  - Usually compilation instruction

Minimal Makefile: compile and link

```
hello: hello.o main.o
        gcc -o hello hello.o main.o

hello.o: hello.c
        gcc -o hello.o -c hello.c -W -Wall -↘
        →ansi -pedantic

main.o: main.c hello.h
        gcc -o main.o -c main.c -W -Wall -↘
        →ansi -pedantic
```

- **Limitations**
  - We can not generate many executables
  - Intermediary files stay on the disk
  - We can not force regeneration

New Rules

- `All` to generate all the executable files
- `Clean` to remove all unused files (mainly `.o`)
- `Mrproper` to remove the generated files (i.e. force total recompilation)

```
all: hello
hello: hello.o main.o
        gcc -o hello hello.o main.o
hello.o: hello.c
        gcc -o hello.o -c hello.c -W -Wall -↘
        →ansi -pedantic
main.o: main.c hello.h
        gcc -o main.o -c main.c -W -Wall -↘
        →ansi -pedantic
clean:
        rm -rf *.o
mrproper: clean
        rm -rf hello
```


Phony Targets

Targets that do not correspond to a file to be generated (like "clean" or "mrproper") are called "phony".
Use:

```
.PHONY: clean
```

to allow running "make clean" even if a file "clean" exists!

Abstract Rules

General definition how to build a file ending in ".x" from files ending in ".y":

```
.c.o:
    gcc -c $<
.java.class:
    javac $<
```

Define Variables

Define Variables

- **One can define new variables in Makefile**
  - Possibility to change all rules in one sentence
  - Syntax: NAME = VALUE
  - Value can be used: $(NAME)
- **Example**
  - Variable CC contains the compilator : gcc
  - Variable CFLAGS contains compilation options
  - Variable LDFLAGS for linking options (libraries to link with)
  - EXEC contains the list of all executables

Example with variables

- **Variables are**

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello
all: $(EXEC)
hello: hello.o main.o
        $(CC) -o hello hello.o main.o $(↘
        →LDFLAGS)
hello.o: hello.c
        $(CC) -o hello.o -c hello.c $(CFLAGS↘
        →)
main.o: main.c hello.h
        $(CC) -o main.o -c main.c $(CFLAGS)
clean:
        rm -rf *.o
mrproper: clean
        rm -rf $(EXEC)
```


Implicit variables

- **Some variables exist without needing to be defined**
  - $@ the target name
  - $< the first dependance
  - $^ the list of all dependances
  - $? all dependances newer than the target
  - $* the name of the file without suffix

Example with implicit variables

- **Names do not need to be repeated**

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello
all: $(EXEC)
hello: hello.o main.o
        $(CC) -o $@ $^ $(LDFLAGS)
hello.o: hello.c
        $(CC) -o $@ -c $< $(CFLAGS)
main.o: main.c hello.h
        $(CC) -o $@ -c $< $(CFLAGS)
clean:
        rm -rf *.o
mrproper: clean
        rm -rf $(EXEC)
```


Compilation options

Useful compilation options

- **Debugging**
  - -g Compilation option for debug information (for gdb debugger)

```
CFLAGS=-W -Wall -ansi -pedantic -g
```

- **Optimizing**
  - -O2 for a good optimization
  - -O4 for a very good optimization
  - -O6 compilation may not finish
  - -Os optimize for small binary size

```
CFLAGS=-W -Wall -ansi -pedantic -O6
```


Automake and Autoconf

Autotools

- **Autotools are used to generate complex Makefiles**
  - Makefiles can become very complex
  - Contain dependencies of multiple c-files and h-files
- **Deployment is dependent of the system where the compilation is done**
  - Can be deployed on a different version of Unix
  - Which compilator is installed
  - Verify that libraries are installed
- **Distribution of software to be compiled**
  - Open Source software,
  - Need to be recompiled on every system where they can be installed.

Autotools: Files

Based on files

- **Makefile.am**
  - Contain the list of executable files
  - Contain for each executable, the list of source files (.c files)
- **configure.ac**
  - Contains the macros to be executed for the configuration
  - Must contain the name of typical files for our project (will be tested if they are present).
  - Tests if the target system can compile the system (has the right libraries for instance).
- **Text Files** contain the description of the project
  - `NEWS`, `README`, `AUTHORS`, `ChangeLog`

Autotools: Process

One need to execute the following commands

- `autoscan`
- `aclocal`
- `autoheader`
- `automake --add-missing`
- `autoconf`

Generates all the required files for the deployment

Autotools: Standard compilation

All C projects for UNIX system follow the same list of commands

- `./configure`
  - Configure the compilation according to the environment
  - Depends on compilators, libraries installed on the machine
- `make`
  - Compile all files, creates the executables
- `make install` (should often be done as "root")
  - Copy the executables and dynamic libraries where they should be

Conditional compilation

Conditional compilation

Target platforms differ:

- Available libraries
- Available system headers and system calls
- Known bugs
- Performance characteristics ...

Solution: use C preprocessor to adapt to platform characteristics:

```
#include "config.h"
#if HAVE_FEATURE
  use_feature ();
#else
  alternative ();
#endif
```

"config.h" (or equivalent) is commonly *generated* by the build system.

Using autotools to generate config.h

```
  AC_CONFIG_HEADERS([config.h])
  AC_CHECK_HEADER([stdbool.h])
  AC_DEFINE([[foo]], [[bar]])
  AC_ARG_ENABLE([[myoption]],
   [AS_HELP_STRING([--myoption],[do something])↘
   →],
   [], [myoption='yes])
␣␣AS_IF([[test␣"x$myoption"␣=␣"xyes"]],
␣␣[AC_DEFINE([[MYOPTION]],␣[[1]],␣[Define␣to␣1␣↘
→to␣set␣myoption])])
```


Conclusion

Conclusion

- **Makefile is necessary**
  - Impossible to know all the compilation details
  - Need to change configuration easily
- **A lot of possibilities**
  - Installation programs written in Makefile
  - Configuration is tested
  - Compilation depends on machine and libraries

Conclusion for the Module

- **Assembly language**
  - Exactly what is executed in the CPU
  - List all the instructions
  - Direct access to Memory
- **C Programming Language**
  - Can be very near to assembly
  - Can manipulate memory addresses (pointers)
- **Goal of this course**
  - See how a computer works

Bibliography

- Makefile Tutorial:
  `http://gl.developpez.com/tutoriel/outil/makefile/`
  `http://timmurphy.org/2010/05/04/pthreads-in-c-a-minimal-working-example/`