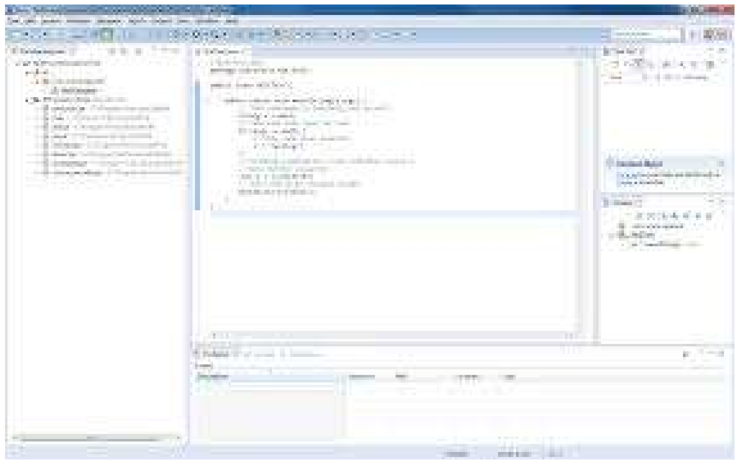
Linux Environment

C Programming Tools

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

IDE

- Integrated Development Environment (IDE)
- Some "free" IDEs are
  - Netbeans
    - C, C++
    - https://www.netbeans.org/
  - Code::Blocks
    - C, C++, Fortran
    - http://www.codeblocks.org/



IDE

- Eclipse
  - Java, C++, etc.
  - http://www.eclipse.org/
- CodeLite
- Geany
  - Very simple, few plug-ins
- MonoDevelop
- Anjuta

Editor

- Editors typically used in UNIX/Linux
  - Sublime
  - Atom
  - **Vim (Vi)**
  - **Emacs**
  - Gedit
  - Nano
  - Brackets
  - Bluefish
  - Spacemacs

Compiler and Debugger

- Compiler
  - GCC
  - G++
  - Makefile
  - Configure
- Debugger
  - GDB

Compiler: gcc

- Open-Source GNU project
  - gcc compiler and linker
  - Supports C and C++

Documentation
Local help : man gcc
Online resources : http://www.gnu.org

- Command syntax
  - gcc <options> <arguments>
    - Options: list of flags that control the compiler and the linker; there are options for compilation only, for linker only, or both
    - Arguments: list of files that gcc reads and process depending on the given options

Examples

- Compilation of a set of files that produces the corresponding object files
  - gcc –c file1.c
  - gcc –c file2.c
  - gcc –c main.c
- Link of the object files produces the executable file
  - gcc –o myexe file1.o file2.o main.o
- Compilation and linking with a single command
  - gcc –o myexe file1.c file2.c main.c

gcc options

- Most common options
  - -c file
    - Compilation only
  - -o file
    - Specifies the executable name; generally indicates the name of the final executable (after the link operation)
  - -g
    - gcc does not produce optimized code, but inserts additional information useful for debugging (see gdb)
  - -Wall
    - Output a warning for all possible code errors

gcc options

Do not insert spaces

- -Idir
  - Specify further directories where searching header files
  - More than one directory can be specified (-Idir1 –Idir2 …)
- -lm
  - Specifies to use the math library
- -Ldir
  - Specifies the search directories for pre-existing libraries to be linked

Example 1

```
gcc -Wall -g -I. -I/myDir/subDir -o myexe \
  myMain.c \
  fileLib1.c fileLib2.c file1.c \
  file2.c file3.c -lm
```

- Compilation of many source files, followed by linking and creation of the executable file
  - Multi-row command
  - Provides “All Warnings”
  - Debug option (i.e., do not optimize code9
  - Find the header files in two directories
  - Links the math library

Makefile

- Support tools for the development of complex projects
- Developed since 1998
- Made up of utilities
  - Makefile
  - Make
- Provides a convenient tool to automate the compilation and linker steps
- Help
  - man make

First scripting language used in this course

Extremely flexible instrument, but its main strength is the verification of dependencies

Makefile

- Makefile has two main aims
  - Automatically perform repetitive tasks
  - Avoid (re)doing unnecessary tasks
    - by verifying the file **dependencies** and **modification times** (e.g., **re-compile** only the **files** that have been **modified** since the previous make command)
- Two phases
  - Write a Makefile file
    - A text file similar to a script (shell script or other)
  - The Makefile file is interpreted with the **make** utility
    - This way you can compile and link

Make options

- Make can be executed using different options
  - Does not execute, just displays the commands
    - -n
  - Ignores possible errors and proceeds with the next commands
    - -i, --ignore-errors
  - Output debug information during the execution
    - -d
  - --debug=[options]
    - Options: a = print all info, b = basic info, v = verbose = basic + other, i = implicit = verbose + other

Makefile options

- The command **make** can take as argument a source file (Makefile), with name different than standard ones
  - The **make** command executes by default
    - the file **makefile** if it exists
    - Or the file **Makefile** if the file makefile does not exist
  - -f <fileName> (or --file <fileName>)
    - Allows you to execute the Makefile with name <fileName>
    - make --file <fileName>
    - make --file=<fileName>
    - make -f <fileName>

Makefile format

Tabulation character

```
target: dependency
  <tab>command
```

- A **`Makefile`** includes
  - Empty lines
    - They are ignored
  - Lines starting with "#"
    - They are comments, and consequently ignored
  - Lines that specify rules
    - Each rule specifies a target, some dependencies, and actions; it can occupy one or more lines
    - Very long lines can be splitted by inserting the "\" character at the end of the line

Makefile format

```
target: dependency
  <tab>command
```

- When a Makefile is executed (with the command make)
  - The default behavior is to execute the first rule
    - i.e., the first target in the file
  - If more targets are specified, the desired target can be passed as an argument to make
    - `make <targetName>`
    - `make -f <myMakefile> <targetName>`

Makefile format

- A makefile consists of "rules" like this:

```
target: dependency
  <tab>command
```

- Each rule includes
  - Target Name
    - Usually the name of a file
    - Sometimes the name of an action (which is named "phony" target)
  - dependency list that must be verified to execute the target
  - Command, or list of commands
    - Each command is preceded by a mandatory **TAB** character, **invisible** but necessary

Example 1: Single target

```
target:
  <tab>gcc -Wall -o myExe main.c -lm
```

Notice: TAB

- Specifies
  - A single target with name **target**
  - The target does not have dependencies
- Executing the Makefile
  - The **target** is executed
  - Since the target does not have dependencies, the execution of the target corresponds to the execution of the compilation command

Example 2: Multiple targets

```
project1:
  <tab>gcc -Wall -o project1 myFile1.c

project2:
  <tab>gcc -Wall -o project2 myFile2.c
```

- The Makefile specifies more rules
  - Need to choose which is the target to execute
  - The default consists in the execution of the first target
- Executing the command
  - make
    - The target project1 is executed
  - make -f project2
    - The target project2 is executed

Example 3: Multiple targets and actions

```
target:
  <tab>gcc -Wall -o my \
  <tab>  main.c \
  <tab>  bst.c list.c queue.c stack.c
  <tab>cp my /home/myuser/bin

clean:
  <tab>rm -rf *.o *.txt
```

Command on more rows

- Specify more rules
  - Rules have no dependencies
  - The first target executes two commands (gcc and cp)
    - This first target is executed with the commands
      - make
      - make -f target

Example 3: Multiple targets and actions

```
target:
  <tab>gcc -Wall -o my \
  <tab>  main.c \
  <tab>  bst.c list.c queue.c stack.c
  <tab>cp my /home/myuser/bin

clean:
  <tab>rm -rf *.o *.txt
```

Command on more rows

- The second target removes all the files with extension .o and all the files with extension .txt
  - This second target is executed with the command
    - make -f clean

Example 4: dependencies

```
target: file1.o file2.o
  <tab>gcc -Wall -o myExe file1.o file2.o

file1.o: file1.c myLib1.h
  <tab>gcc -Wall -g -I./dirI -c file1.c

file2.o: file2.c myLib1.h myLIb2.h
  <tab>gcc -Wall -g -I./dirI -c file2.c
```

- Execution of multiple targets in the presence of dependencies
  - It checks if target dependencies are more recent than the current target
  - In this case, dependencies are performed before the execution of the current target
  - This process iterates recursively

Example 4: dependencies

```
target: file1.o file2.o
  <tab>gcc -Wall -o myExe file1.o file2.o

file1.o: file1.c myLib1.h
  <tab>gcc -Wall -g -I./dirI -c file1.c

file2.o: file2.c myLib1.h myLIb2.h
  <tab>gcc -Wall -g -I./dirI -c file2.c
```

- Target has file1.o and file2.o as dependencies
  - rule file1.o is checked
    - If file1.c (or myLib1.h) is more recent than file1.o, this rule (i.e., the gcc command) is executed
    - Otherwise this rule is not executed
  - The same is done for the file2.o rule
  - At the end the target is executed **if necessary**

Example 4: dependencies

Action name ("phony" target)

```
target: file1.o file2.o
  <tab>gcc -Wall -o myExe file1.o file2.o

...

file2.o: file2.c myLib1.h myLIb2.h
  <tab>gcc -Wall -g -I./dirI -c file2.c
```

File name

- If the target is not a file name, it is a "phony" target that should always be executed
- To be sure that is always executed
  - .PHONY : target

Regardless the existence of a file with the same name and more recent than dependencies

Implicit rules and modularity

- There exist very powerful rules for improving modularity and make more efficient the writing of makefiles
  - Use of macros
  - Use of implicit rules
    - The dependence between .o and .c is automatic
    - The dependence between .c and .h is automatic
    - Recursive dependencies are analyzed automatically
    - etc.

Example 5: Macro

```
CC=gcc
FLAGCS=-Wall -g
SRC=main.c bst.c list.c util.c

project: $(SRC)
  <tab>$(CC) $(FLAGS)-o project $(SRC) -lm
```

Definition of macro:
macro=name
(with or without spaces)

Use of the macro:
$(macro)

- Macro allows to define
  - Symbols
    - Compilers, compilation flags, etc.
  - Lists
    - Object files, executables, directories, etc.

Example 6: Multi-Folder

```
CC=gcc
FLAGCS=-Wall -g
SDIR=source
HDIR=header
ODIR=obj


project: $(ODIR)/main.o $(ODIR)/bst.o
  <tab>$(CC) $(FLAGS)-o $@ $^

$(ODIR)/main.o: $(SDIR)/main.c $(HDIR)/main.h
  <tab>$(CC) $(FLAGS) -c $^

$(ODIR)/bst.o: $(SDIR)/bst.c $(HDIR)/bst.h
  <tab>$(CC) $(FLAGS) -c $^
```

The macro $@ copies the current "target name"

The macro $^ copies the list of files reported in the list of dependencies

The macro $< would copy the first file reported in the list of dependencies

Debugger: gdb

- Software package used to analyze the behavior of another program in order to identify and eliminate errors (bugs)
- GNU debugger **`gdb`** is available for almost all Operating Systems
- It can be used
  - As a "stand-alone" tool
    - Particularly inconvenient use
  - Integrated with many editors (e.g., emacs)
  - Embedded in some graphical IDE
- Abbreviate form of commands can be given

Debugger: gdb

Action	Command
Execution commands	run (r) next (n) next <NumberOfSteps> step (s) step <NumberOfSteps> stepi (si) finish (f) continue (c)

Breakpoint commands	info break
	break (b), ctrl-x-blank
	break LineNumber
	break FunctionName
	break fileName:LineNumber
	disable BreakpointNumber
	enable BreakpointNumber

Debugger: gdb

Action	Command
Print commands	print (p) print expression display expression
Stack operations	down (d) up (u) Info args Info locals
Code listing commands	list (p) list LineNumber list FirstLine, LastLine
Miscellaneous commands	file fileName exec filename kill