Building, Debugging, Documenting

O.S. Laboratory

Agenda

- Building
  - Make
- Debugging
  - Gdb
- Documenting
  - Doxygen

Build chain

- An automatic build chain is useful to:
  - Compile thousands of source files.
  - Compile quickly: without running manually the compiler and without recompiling up-to-date files.
  - Compile easily: third parties must be able to compile the code.
  - Compile on various platforms.

Build chain: Make

- Make:
  - Traditional build chain in Unix environment.
  - Requires a configuration file (***Makefile***).
  - Support command line parameters.
  - Based on the concept of *dependency*.
  - Based on the concept of *timestamp*.
- Cons:
  - Not easily portable.
  - Space (indentation) sensitive semantics.

Timestamp and dependency

- Dependency:
  - Each target depends on (i.e. is built from) a list of other targets, that must built before it.
    - A .o file depends on the .c from which is built
    - An executable depends on .o files
- Timestamp:
  - A target is considered out-of-date if its last modify timestamp is older than one of its dependencies
  - When out-of-date, the target is re-built
    - When a .c is modified, the matching .o is re-built, and the executable is re-linked

Makefile

- Makefile contains:
  - Configuration variables.
    - E.g. which compiler and linker to use, flags to be passed to compiler and linker.
  - Directives (e.g. compiling instructions)

    <target>: <main source file> <other files...>

    \tab <compiling/linking commands>

    - NOTE: headers depending on other headers can be "updated" by using *touch*

Makefile

- Variables:
  - VARNAME= VALUELIST
    - Assigns a list of values, at reading time (portable).
  - VARNAME:= VALUELIST
    - Assigns to a variable a list of values ***immediately*** (GNU).
  - VARNAME+= VALUELIST
    - Adds the list of values to previous assigned values.
  - $(VARNAME)
    - Reads a variable.

Makefile

- Common variable names:
  - CC: the C compiler
    - CFLAGS: flags for the C compiler.
  - CXX: the C++ compiler.
    - CXXFLAGS: flags for the C++ compiler.
  - LD: the linker.
    - LDFLAGS: flags for the linker
    - LDLIBS: libraries to link.

Makefile

- Predefined macros:
  - $@: the target.
  - $<: the first dependency.
  - $^: all the dependencies.
  - $?: modified files.
  - %: the current file.

Makefile: Special targets

- First target:
  - Executed if make is run without other parameters.
- Common target names:
  - all: compiles and links all the targets.
  - clean: deletes generated files.
  - install: installs all generated files.
  - help: prints a list of possible targets.
  - doc: generates documentation.
- .PHONY
  - Collects as dependencies targets which do not match any generated file name (e.g. all, clean).

Makefile: Running

- Run make with Makefile, on default target:

  make

- Run make in parallel (optional: max number of parallel threads):

  make -j4

- Run make with *MyMake* as config file:

  make -f MyMake

- Run make with a specific target (app.x):

  make app.x

Makefile Example 1

- Files:
  - Implementation: *main.c com.c set.c*
  - Headers: *com.h set.h*, located into *include*.
- Auxiliary libraries:
  - Standard C *math* library.
  - Custom *libsupport.so* located into *lib*.

Makefile Example 1

```
# Configuring:
CC:= gcc
LD:= gcc
CFLAGS:= -c -Wall -Iinclude
LDFLAGS:= -Llib
LDLIBS:= -lsupport -lm
```

Makefile Example 1

```
# Sources:
SRCS:= main.c com.c set.c

# Objects:
OBJS:= $(SRCS:.c=.o)
```

Makefile Example 1

```
# Default target:
all: myapp.x

myapp.x: $(OBJS)
   @echo Linking $@
   @$(LD) $(LDFLAGS) -o $@ $^ $(LDLIBS)

clean:
     @rm *.o myapp.x
```

Makefile Example 1

```
# Compiling, by using
# pattern matching:
%.o: %.c
	@echo $@
	@$(CC) $(CFLAGS) -o $@ $<

.PHONY: all clean
```

Makefile Example 1

- Example 1 pros:
  - Very short.
  - Very reusable.
- Example 1 cons:
  - Not fine tuned: does not capture dependencies related to header files.

Makefile Example 2

```
# Fine tuned compiling:
main.o: main.c include/com.h include/set.h
    @echo $@
    @$(CC) $(CFLAGS) -o $@ $<
set.o: set.c include/set.h
    @echo $@
    @$(CC) $(CFLAGS) -o $@ $<
com.o: com.c include/com.h include/set.h
    @echo $@
    @$(CC) $(CFLAGS) -o $@ $<
```

Makefile Example 2

- Example 2 pros:
  - Captures all the dependencies.
  - Very safe, even in case of parallel compiling.
- Example 2 cons:
  - Longer to write.
  - Cannot be reused.

Agenda

- Building
  - Make
- Debugging
  - Gdb
- Documenting
  - Doxygen

Debugging

- Verification and debugging:
  - One of the hardest phases of development.
  - The most time consuming activity (70-80%)
- It is fundamental to use tools to:
  - Simplify debugging.
  - Speed up the bug fixing.
  - Avoid strange behaviors due to buffered methods in case of segfault.

Debugger

- A debugger allows:
  - To suspend the execution of a program in specific points.
  - To execute the program step by step.
  - To inspect variable values at run-time.

Debugger

- GDB (Gnu Debugger):
  - Free and open source.
  - Standard tool used with gcc.
  - Requires an enriched executable:
    - gcc -g: compile with debugging infos.
    - gcc -ggdb: as -g, but targeting gdb explicitly.
  - Command line tool.
    - DDD is the most famous graphical interface (no more maintained).
    - cgdb: a textual interface written by using Curses.

GDB: Basic Commands

- Load an executable with gdb

  gdb my_exec.x

  - gdb --args my_exec.x <program args>
- Quit from gdb:

  quit / q
- Set eventual executable arguments

  set args <arguments>
- Help about a topic (e.g. about a command)

  help <topic>

GDB Breakpoints

- Breakpoint:
  - A breakpoint marks a place in the source code.
  - The execution of a program is suspended when a breakpoint is reached.
  - Useful to inspect the program at runtime in a specific place.
  - GDB syntax:
    - break / b <file:line>
    - break / b <methodName>
    - delete <number> (to remove breakpoint #<number>)

GDB Watch

- Watch point
  - Suspends the execution whenever a watched expression changes its value.
  - Useful to check the evolution of a variable.
  - GDB syntax:

    watch <expression>

GDB Expression evaluation

- Print
  - Prints the result of an expression, by using the current values for the variables.
  - Useful to inspect variables.
  - GDB syntax:

    print / p <expression>

GDB Stack inspection

- Backtrace
  - Printing the list of last method called.
  - Allows to understand the execution order.
  - GDB syntax:

    backtrace / bt
- Frame
  - Prints infos about the current method stack.
  - GDB syntax:

    frame / f (short descr) info frame / info f (long)

GDB Execution

- Executes a program from the beginning:

  run / r [<arguments>]

- Continues the execution, from the reached point:

  continue / c

- Execute the next line *atomically*:

  next / n

- Execute the next instruction, going into eventual methods called:

  step / s

- Repeat the last command:

  \return

Agenda

- Building
  - Make
- Debugging
  - Gdb
- Documenting
  - Doxygen

Documenting

- Documentation:
  - For users:
    - Manuals, guides, tutorials, etc..
    - API documentation (for libraries).
  - For developers:
    - API documentation.
    - Implementation code documentation.
    - Other (use cases, specification, etc.).

Documenting

- Source code documentation:
  - Comments for the API.
    - Automatic generation of API documentation.
  - Comments on implementation details.
    - Embedded in the source code.

Doxygen

- Doxygen:
  - Tool for automatic generation of API documentation.
  - Runs on many platforms.
  - Free and open source.
  - Supports many languages.
    - C, C++, Java.
  - Supports many output formats.
    - ***HTML***, Latex, RTF.

Running Doxygen

- Generate a configuration file (Doxyfile):

  doxygen -g

- Configure the Doxyfile:

  emacs Doxyfile

- Run doxygen:

  doxygen

- Run with a specific configuration file:

  doxygen ConfigFile

Doxygen Configuration

- Main configuration parameters:
  - PROJECT_NAME (e.g. MyLib)
  - OUTPUT_DIRECTORY (e.g. doc)
  - INPUT / FILE_PATTERNS (e.g. src)
  - RECURSIVE (e.g. YES)
  - EXCLUDE / EXCLUDE_PATTERNS (*.java)
  - GENERATE_* (e.g. GENERATE_HTML)

Doxygen Tags

- Doxygen parses special comments:
  - /** Comment parsed */
    - /* Comment **not** parsed */
  - /// Comment parsed
    - // Comment **not** parsed
- Doxygen accept tags to manage special documentation:
  - @tag
  - \tag

Doxygen Tags

- Main tags:
  - @brief <comment>
    - A short comment.
  - @param <paramName> <comment>
    - For method parameter
  - @return <comment>
    - For method return value.
  - @throw <exceptionName> <comment>
    - For exceptions thrown by a method (not for C)

Doxygen Tags

- Tags for global documentation:
  - Information about file content:

```
/** @file
 * <comment >
 */
```

  - Grouping logically related methods:

```
/** @name <groupName>*/
/*@{ */
<methodsWithTheirDocumentation>
/*@} */
```

Doxygen Example

```
/** @name List accessors. */
/*@{ */

/** @brief Gets the element at given position.
 * Linear complexity.
 * @param l The list.
 * @param pos The position.
 * @return The stored element.
 */
void * getListElement( List * l, int pos );

/*@} */
```