

Environnements et Outils de Développement

Cours 6 — Building with make

Stefano Zacchioli

`zack@pps.univ-paris-diderot.fr`

Laboratoire PPS, Université Paris Diderot

URL <http://upsilon.cc/zack/teaching/1314/ed6/>

Copyright © 2012–2013 Stefano Zacchioli

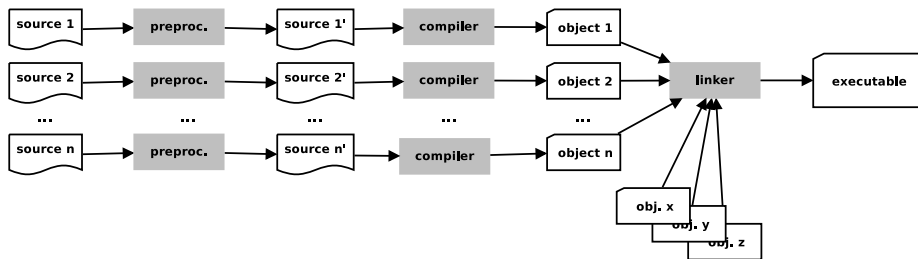
© 2009 Luca Padovani

License Creative Commons Attribution-ShareAlike 4.0 International License

http://creativecommons.org/licenses/by-sa/4.0/deed.en_US



The build process (reminder)



Monolithic solution

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

void swap(int* x, int* y)
{
    ...
}

int gcd(int m, int n)
{
    ...
}

int main(int argc, char* argv[])
{
    ...
}
```

```
$ gcc main.c
$
```

Modularization

Let's split main program into 3 modules :

- 1 main program (main.c)
- 2 GCD (Greated Common Divisor) computation (gcd.c)
- 3 value exchange (swap.c)

Either :

```
$ gcc -c main.c  
$ gcc -c gcd.c  
$ gcc -c swap.c  
$ gcc main.o gcd.o swap.o
```

or

```
$ gcc -c main.c gcd.c swap.c  
$ gcc main.o gcd.o swap.o
```

or even

```
$ gcc *.c  
$
```

Inter-module code consistency

```
void swap(int* x, int* y)
{
    int t = *x;
    *x = *y;
    *y = t;
}
```

```
int gcd(int m, int n)
{
    while (m > 0) {
        if (m < n) swap(m, n);
        m -= n;
    }
    return n;
}
```

```
$ gcc -o gcd *.c
./gcd 5 6
Segmentation fault
$
```

Headers

```
#ifndef __SWAP_H__
#define __SWAP_H__

void swap(int*, int*);

#endif /* __SWAP_H__ */
```

```
#include "swap.h"
```

```
int gcd(int m, int n)
{
    while (m > 0) {
        if (m < n) swap(m, n); /* error! */
        m -= n;
    }
    return n;
}
```

```
$ gcc -c gcd_wrong.c
```

```
gcd_wrong.c: In function 'gcd':
```

```
gcd_wrong.c:7: warning: passing argument 1 of 'swap' makes
pointer from integer without a cast
```

```
gcd_wrong.c:7: warning: passing argument 2 of 'swap' makes
pointer from integer without a cast
```

Correct version

```
#ifndef __GCD_H__
#define __GCD_H__

int gcd(int, int);

#endif /* __GCD_H__ */

#include "gcd.h" /* important! */
#include "swap.h"

int gcd(int m, int n)
{
    while (m > 0) {
        if (m < n) swap(&m, &n);
        m -= n;
    }

    return n;
}
```

Planning build with make

make allows to **plan** and **describe** once and for all the build process in a Makefile

```
gcd: main.c gcd.c swap.c
    gcc -o gcd main.c gcd.c swap.c
clean:
    rm -f gcd
```

- gcd and clean are **targets** of specified **rules**
- after ":" you have the target's **prerequisites**
- in subsequent line(s), prefixed by a *tabulation character*, you have the **commands** that allow to build a target

make behavior

If a Makefile (or a makefile) exists, executing make will try to obtain its **first target**

An alternative Makefile can be specified using `make -f`

The variant `make target` will try to obtain the given *target* (instead of the first one in the Makefile)

With `make -n` we can examine the command sequence that make *would execute*, without actually executing them (**dry run** mode)

```
make -n  
gcc -o gcd main.c gcd.c swap.c
```

Makefile variables

It often comes handy to make parametric rules, based on the value of **variables**.

```
SOURCES = main.c gcd.c swap.c  
gcd: $(SOURCES)  
    gcc -o gcd $(SOURCES)  
clean:  
    rm -f gcd
```

The reference `$(SOURCES)` expands to the string “main.c gcd.c swap.c”

Note

In spite of having the same syntax, Make's `$(SOURCES)` has nothing to do with Shell's command substitution `$(code...)`. One is expanded by make, the other by the shell, and the semantics is radically different.

How make works

When you invoke `make target`, `make` computes the (acyclic!) **graph of dependencies** that are needed to obtain `target`

```
SOURCES = main.c gcd.c swap.c
```

```
gcd: $(SOURCES)
    gcc -o gcd $(SOURCES)
```

```
clean:
    rm -f gcd
```

- to obtain `gcd` we need `main.c gcd.c swap.c`
- to obtain `main.c` we don't need anything, as the file is already available
- ditto for `gcd.c`
- ditto for `swap.c`
- the prerequisites for `gcd` (i.e. its dependencies) are now satisfied, so we run the command(s) associated to the target `gcd`

How make works : unsatisfied prerequisites

```
SOURCES = main.c gcd.c swop.c
```

```
gcd: $(SOURCES)
```

```
    gcc -o gcd $(SOURCES)
```

```
clean:
```

```
    rm -f gcd
```

make does not know that the Makefile “is wrong”

```
$ make -f Makefile.err
```

```
make: *** No rule to make target 'swop.c', needed by 'gcd'. Stop.
```

How make works : time

make laziness

Before running the command(s) associated to a target t , make verify two things :

- 1 the time T_t of last modifications of target t
 - ▶ assumption : targets are files
- 2 the *most recent* time T_d of modifications of t dependencies

If $T_t > T_d$, the target is more recent than *each* of its dependencies

Therefore, no need to rebuild t , and the associated command(s) will be ignored

How make works : time — example

```
$ ls -l
total 32
-rwxr-xr-x 1 zack zack 7412 mar  6 19:23 gcd
-rw-r--r-- 1 zack zack  155 mar  6 19:23 gcd.c
-rw-r--r-- 1 zack zack   81 mar  6 19:23 gcd.h
-rw-r--r-- 1 zack zack  327 mar  6 19:24 main.c
-rw-r--r-- 1 zack zack   75 mar  6 19:28 Makefile
-rw-r--r-- 1 zack zack   65 mar  6 19:27 swap.c
-rw-r--r-- 1 zack zack   88 mar  6 19:23 swap.h
```

```
$ make
gcc -o gcd main.c gcd.c swap.c
```

```
$ ls -l
total 32
-rwxr-xr-x 1 zack zack 7412 mar  6 19:30 gcd
-rw-r--r-- 1 zack zack  155 mar  6 19:23 gcd.c
-rw-r--r-- 1 zack zack   81 mar  6 19:23 gcd.h
-rw-r--r-- 1 zack zack  327 mar  6 19:24 main.c
-rw-r--r-- 1 zack zack   75 mar  6 19:28 Makefile
-rw-r--r-- 1 zack zack   65 mar  6 19:27 swap.c
-rw-r--r-- 1 zack zack   88 mar  6 19:23 swap.h
```

```
$ make
make: 'gcd' is up to date.
```

Towards separate compilation

```
$ make
make: 'gcd' is up to date.

$ touch gcd.c
$ ls -l
-rwxr-xr-x 1 zack zack 7412 mar  6 19:30 gcd
-rw-r--r-- 1 zack zack  155 mar  6 19:31 gcd.c
-rw-r--r-- 1 zack zack   81 mar  6 19:23 gcd.h
-rw-r--r-- 1 zack zack  327 mar  6 19:24 main.c
-rw-r--r-- 1 zack zack   75 mar  6 19:28 Makefile
-rw-r--r-- 1 zack zack   65 mar  6 19:27 swap.c
-rw-r--r-- 1 zack zack   88 mar  6 19:23 swap.h

$ make
gcc -o gcd main.c gcd.c swap.c
```

- all `.c` files get recompiled even if only one has changed!

Separate compilation

```
SOURCES = main.c gcd.c swap.c
```

```
OBJS = main.o gcd.o swap.o
```

```
gcd: $(OBJS)
```

```
    gcc -o gcd $(OBJS)
```

```
main.o: main.c gcd.h
```

```
    gcc -c main.c
```

```
gcd.o: gcd.c gcd.h swap.h
```

```
    gcc -c gcd.c
```

```
swap.o: swap.c swap.h
```

```
    gcc -c swap.c
```

```
clean:
```

```
    rm -f gcd $(OBJS)
```

- dependencies are now more precisely associated to each target
 - ▶ and we also take into account .h files now

Separate compilation — example

```
$ make
gcc -c main.c
gcc -c gcd.c
gcc -c swap.c
gcc -o gcd main.o gcd.o swap.o
```

```
$ touch gcd.c
```

```
$ ls -l
```

```
total 40
```

```
-rw-r--r--  1 zack  zack      231 Dec 12 13:12 Makefile
-rw-r--r--  1 zack  zack      322 Dec 12 13:08 main.c
-rw-r--r--  1 zack  zack      836 Dec 12 13:13 main.o
-rwxr-xr-x  1 zack  zack    12668 Dec 12 13:13 gcd
-rw-r--r--  1 zack  zack      156 Dec 12 13:13 gcd.c
-rw-r--r--  1 zack  zack       81 Dec 12 13:08 gcd.h
-rw-r--r--  1 zack  zack      436 Dec 12 13:13 gcd.o
-rw-r--r--  1 zack  zack       65 Dec 12 13:08 swap.c
-rw-r--r--  1 zack  zack       88 Dec 12 13:08 swap.h
-rw-r--r--  1 zack  zack      312 Dec 12 13:13 swap.o
```

```
$ make
gcc -c gcd.c
gcc -o gcd main.o gcd.o swap.o
```

Towards a more “proper” Makefile

```
SOURCES = main.c gcd.c swap.c
OBJS = $(SOURCES:%.c=%.o)

gcd: $(OBJS)
    gcc -o gcd $(OBJS)
main.o: main.c gcd.h
    gcc -c main.c
gcd.o: gcd.c gcd.h swap.h
    gcc -c gcd.c
swap.o: swap.c swap.h
    gcc -c swap.c

clean:
    rm -f gcd $(OBJS)
```

Other useful transformation

```
$(basename src/foo.c src-1.0/bar hacks) => src/foo src-1.0/bar hacks
$(addprefix src/,foo bar) => src/foo src/bar
$(wildcard *.c) => gcd.c main.c swap.c
```

See make manual, *Functions for transforming text*

Automatic variables

Automatic variables are predefined (and context-dependent) variables that you can use in Makefile-s.

Using automatic variables simplify writing rules.

Using automatic variables will become *necessary* with implicit rules.

```
SOURCES = main.c gcd.c swap.c
OBJS = $(SOURCES:.c=%.o)
```

```
gcd: $(OBJS)
    gcc -o $@ $(OBJS)
main.o: main.c gcd.h
    gcc -c $<
gcd.o: gcd.c gcd.h swap.h
    gcc -c $<
swap.o: swap.c swap.h
    gcc -c $<
```

```
clean:
    rm -f gcd $(OBJS)
```

Some (not so) frequently used automatic variables

`$$` full target name

`$(<)` name of the first dependency

`$(?)` names of the dependencies that are more recent than target

`$(^)` names of all dependencies (merging duplicates)

`$(+)` names of all dependencies (not merging duplicates)

`$(@D)` target's directory (`dir/foo.o` \Rightarrow `dir`)

`$(@F)` target's base name (`dir/foo.o` \Rightarrow `foo.o`)

`$(<D)` directory of the first dependency

`$(<F)` base name of the first dependency

...

See make manual, *Automatic Variables*

Implicit rules

```
main.o: main.c gcd.h
    gcc -c $<          # same command
gcd.o: gcd.c gcd.h swap.h
    gcc -c $<          # same command
swap.o: swap.c swap.h
    gcc -c $<          # same command
```

An implicit rules defines a rule pattern

```
SOURCES = main.c gcd.c swap.c
OBSJS = $(SOURCES:%.c=%.o)

gcd: $(OBSJS)
    gcc -o $@ $(OBSJS)
%.o: %.c
    @echo "building $(@:%.o=) ..."
    @gcc -c $<
main.o: main.c gcd.h
gcd.o: gcd.c gcd.h swap.h
swap.o: swap.c swap.h

clean:
    rm -f gcd $(OBSJS)
```

Larger project, more directories

```
$ ls -lR
.:
total 12
-rw-r--r--  1 zack  zack  394 Dec 12 14:03 Makefile
-rw-r--r--  1 zack  zack  137 Dec 12 13:42 common.h
drwxr-xr-x  7 zack  zack  238 Dec 12 14:09 io
-rw-r--r--  1 zack  zack  386 Dec 12 13:42 main.c
drwxr-xr-x  5 zack  zack  170 Dec 12 14:09 sorting

./io:
total 20
-rw-r--r--  1 zack  zack  260 Dec 12 14:08 Makefile
-rw-r--r--  1 zack  zack  367 Dec 12 14:05 reader.c
-rw-r--r--  1 zack  zack  121 Dec 12 13:59 reader.h
-rw-r--r--  1 zack  zack  159 Dec 12 13:45 writer.c
-rw-r--r--  1 zack  zack  124 Dec 12 13:45 writer.h

./sorting:
total 12
-rw-r--r--  1 zack  zack  221 Dec 12 14:08 Makefile
-rw-r--r--  1 zack  zack  499 Dec 12 14:02 sort.c
-rw-r--r--  1 zack  zack  113 Dec 12 14:01 sort.h
```

main.c

```
#include <stdio.h>
#include "common.h"
#include "reader.h"
#include "writer.h"
#include "sort.h"

string lines[MAXLINES];

int main() {
    int n_lines = read_lines(lines, MAXLINES);

    if (n_lines > 0) {
        sort(lines, 0, n_lines - 1);
        write_lines(lines, n_lines);
        return 0;
    } else if (n_lines < 0) {
        fprintf(stderr, "error: too many input lines\n");
        return 1;
    }
}
```

common.h

```
#ifndef __COMMON_H__
#define __COMMON_H__

#define MAXLINES 5000
#define MAXLEN 1000

typedef char* string;

#endif /* __COMMON_H__ */
```


io/reader.h & io/reader.c

```
#ifndef __READER_H__
#define __READER_H__

#include "common.h"
int read_lines(string [], int);

#endif /* __READER_H__ */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "reader.h"

int read_lines(string line [], int max_lines) {
    int len;
    int n_lines = 0;
    char buffer[MAXLEN + 1];
    while (fgets(buffer, MAXLEN, stdin)) {
        line[n_lines] = (string) malloc(sizeof(char) * strlen(buffer));
        strcpy(line[n_lines], buffer);
        n_lines++;
    }
    return n_lines;
}
```

io/writer.h & io/writer.c

```
#ifndef __WRITER_H__
#define __WRITER_H__

#include "common.h"

void write_lines(string[], int);

#endif /* __WRITER_H__ */
```

```
#include <stdio.h>

#include "writer.h"

void write_lines(string line[], int n_lines) {
    int i;
    for (i = 0; i < n_lines; i++)
        fputs(line[i], stdout);
}
```

sorting/sort.c

```
#include <string.h>
#include "sort.h"

void swap(string line[], int i, int j) {
    string temp = line[i];
    line[i] = line[j];
    line[j] = temp;
}

void sort(string line[], int left, int right) {
    if (left < right) {
        int i, last;
        swap(line, left, (left + right) / 2);
        last = left;
        for (i = left + 1; i <= right; i++)
            if (strcmp(line[i], line[left]) < 0)
                swap(line, ++last, i);
        swap(line, left, last);
        sort(line, left, last - 1);
        sort(line, last + 1, right);
    }
}
```

Makefile

```
NAME = sort
SOURCES = main.c
OBJS = $(SOURCES:%.c=%.o)
INCLUDES = -I. -lio -lsorting
IOLIB = io/libio.a
SORTINGLIB = sorting/libsorting.a

all:
    make -C io
    make -C sorting
    make $(NAME)

$(NAME) : $(OBJS) $(IOLIB) $(SORTINGLIB)
    gcc -o $@ $^

%.o : %.c
    gcc -c $(INCLUDES) $<

main.o : io/reader.h io/writer.h common.h

clean:
    make -C io clean
    make -C sorting clean
    rm -f $(NAME) $(OBJS)
```

io/Makefile

```
SOURCES = reader.c writer.c
OBJS = $(SOURCES:%.c=%.o)
INCLUDES = -I../

all : libio.a

libio.a : $(OBJS)
        ar cru @$ $^
        ranlib @$

%.o : %.c
        gcc -c $(INCLUDES) $<

reader.o : reader.h ../common.h
writer.o : writer.h ../common.h

clean:
        rm -f libio.a $(OBJS)
```

sort/Makefile

```
SOURCES = sort.c
OBJS = $(SOURCES:%.c=%.o)
INCLUDES = -I../

all : libsorting.a

libsorting.a : $(OBJS)
    ar cru @$ $^
    ranlib @$

%.o : %.c
    gcc -c $(INCLUDES) $<

sort.o : ../common.h

clean:
    rm -f libsorting.a $(OBJS)
```

Building the project — example

```
$ make
make -C io
make[1]: Entering directory 'io'
gcc -c -I../ reader.c
gcc -c -I../ writer.c
ar cru libio.a reader.o writer.o
ranlib libio.a
make[1]: Leaving directory 'io'
make -C sorting
make[1]: Entering directory 'sorting'
gcc -c -I../ sort.c
ar cru libsorting.a sort.o
ranlib libsorting.a
make[1]: Leaving directory 'sorting'
make sort
make[1]: Entering directory '.'
gcc -c -I. -lio -lsorting main.c
gcc -o sort main.o io/libio.a sorting/libsorting.a
make[1]: Leaving directory '.'
$
```

Dependencies on external libraries

Most programs rely on functionalities offered by **external libraries**, i.e. libraries other than the C standard library (that comes with the compiler).

To the end of building an executable that uses external libraries, we need to take care of :

- 1 ensure that external library header files (.h) are available at **compile time**
- 2 ensure that libraries (static .a and/or dynamic .so) are available at **link time**

common.h & main.c

```
#ifndef __COMMON_H__
#define __COMMON_H__

typedef char* string;

#endif /* __COMMON_H__ */
```

```
#include <stdio.h>
#include <glib.h>
#include "reader.h"
#include "writer.h"
#include "sort.h"

int main() {
    GPtrArray* lines = read_lines();
    g_assert(lines);

    if (lines->len > 0) {
        sort(lines);
        write_lines(lines);
        return 0;
    } else
        return 1;
}
```

```
#define MAXLEN 1000

GString* read_line() {
    gint len;
    GString* buffer = g_string_new("");
    gchar chunk[MAXLEN];
    do {
        if (fgets(chunk, MAXLEN, stdin))
            g_string_append(buffer, chunk);
        else
            return NULL;
    } while (!strchr(chunk, '\n'));
    return buffer;
}

GPtrArray* read_lines() {
    GPtrArray* lines = g_ptr_array_new();
    GString* line;
    while ((line = read_line()))
        g_ptr_array_add(lines, line);
    return lines;
}
```

io/writer.h & io/writer.c

```
#ifndef __WRITER_H__
#define __WRITER_H__

#include <glib.h>

void write_lines(GPtrArray*);

#endif /* __WRITER_H__ */
```

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

void write_line(GString* line, gpointer user_data) {
    g_return_if_fail(line != NULL);
    fputs(line->str, stdout);
}

void write_lines(GPtrArray* lines) {
    g_return_if_fail(lines != NULL);
    g_ptr_array_foreach(lines, (GFunc) write_line, NULL);
}
```

io/sort.h & io/sort.c

```
#ifndef __SORT_H__
#define __SORT_H__

#include <glib.h>

void sort(GPtrArray*);

#endif /* __SORT_H__ */
```

```
#include <string.h>
#include "sort.h"

gint compare_lines(const GString** s1, const GString** s2) {
    g_return_val_if_fail(s1 != NULL, 0);
    g_return_val_if_fail(s2 != NULL, 0);
    return strcmp((*s1)->str, (*s2)->str);
}

void sort(GPtrArray* lines) {
    g_return_if_fail(lines != NULL);
    g_ptr_array_sort(lines, (GCompareFunc) compare_lines);
}
```

Makefile

```
NAME = sort
SOURCES = main.c
OBJS = $(SOURCES:%.c=%.o)
INCLUDES = -l. -lio -lsorting `pkg-config glib-2.0 --cflags`
LIBS = `pkg-config glib-2.0 --libs`
IOLIB = io/libio.a
SORTINGLIB = sorting/libsorting.a

all:
    make -C io
    make -C sorting
    make $(NAME)

$(NAME) : $(OBJS) $(IOLIB) $(SORTINGLIB)
    gcc -o $@ $^ $(LIBS)

%.o : %.c
    gcc -c $(INCLUDES) $<
main.o : io/reader.h io/writer.h common.h

clean:
    make -C io clean
    make -C sorting clean
    rm -f $(NAME) $(OBJS)
```

io/Makefile

```
SOURCES = reader.c writer.c
OBJS = $(SOURCES:%.c=%.o)
INCLUDES = -I../ 'pkg-config glib-2.0 --cflags'

all : libio.a

libio.a : $(OBJS)
        ar cru @$ $^
        ranlib @$

%.o : %.c
        gcc -c $(INCLUDES) $<

reader.o : reader.h ../common.h
writer.o : writer.h ../common.h

clean:
        rm -f libio.a $(OBJS)
```

sorting/Makefile

```
SOURCES = sort.c
OBJS = $(SOURCES:%.c=%.o)
INCLUDES = -I../ 'pkg-config glib-2.0 --cflags '

all : libsorting.a

libsorting.a : $(OBJS)
    ar cru @$ $^
    ranlib @$

%.o : %.c
    gcc -c $(INCLUDES) $<

sort.o : ../common.h

clean:
    rm -f libsorting.a $(OBJS)
```

Building the project — example

```
$ make
make -C io
make[1]: Entering directory 'io'
gcc -c -I../ 'pkg-config glib-2.0 --cflags' reader.c
gcc -c -I../ 'pkg-config glib-2.0 --cflags' writer.c
ar cru libio.a reader.o writer.o
ranlib libio.a
make[1]: Leaving directory 'io'
make -C sorting
make[1]: Entering directory 'sorting'
gcc -c -I../ 'pkg-config glib-2.0 --cflags' sort.c
ar cru libsorting.a sort.o
ranlib libsorting.a
make[1]: Leaving directory 'sorting'
make sort
make[1]: Entering directory 'sort-glib'
gcc -c -I. -lio -lsorting 'pkg-config glib-2.0 --cflags' main.c
gcc -o sort main.o io/libio.a sorting/libsorting.a 'pkg-config \
  glib-2.0 --libs'
make[1]: Leaving directory 'sort-glib'
$
```

- 1dd sort

State of the project — discussion

- tool names (gcc, ar, make, ...) and their invocation syntax are hard-coded in Makefile-s
- building *needs* the existence of specific libraries and fails abruptly if the corresponding prerequisites are not available
- maintaining the project across several directories is clumsy and error-prone
- Makefile-s do not implement targets to package, install, and uninstall the software

All those aspects are tolerable for small projects. They become too constraining for large and complex projects.

References

- GNU Make homepage : <http://www.gnu.org/software/make/>
- GNU Make manual
 - ▶ \$ info make
 - ▶ M-x info in Emacs
 - ▶ <http://www.gnu.org/software/make/manual/>
- book : *Managing Projects with GNU Make*, Robert Mecklenburg, 3rd ed. O'Reilly, 2004.
 - ▶ available online at <http://oreilly.com/catalog/make3/book/index.csp> under the terms of the GNU Free Documentation License