# Programmation in C

Adrien Poteaux

CRIStAL, Université de Lille

Year 2022-2023



# Very important: write your code well !

- Indent it,
- Comment it,
- Almost never more than one instruction per line,
- Use header files (`.h`) properly,
- Use several files if needed,
- *Always* use a makefile !

# This is not good !

International Obfuscated C Code Contest (http://www.ioccc.org):

- Example 1:

```
m(char*s,char*t) {
return *t-42?*s?63==*t|*s==*t&&m(s+1,t+1):
          !*t:m(s,t+1)||*s&&m(s+1,t);
}
main(int c,char **v) { return!m(v[1],v[2]); }
```

- Example 2:

```
#include <stdio.h>
int O,o,i;char*I="";
main(l){O&=l&1?*I:~*I,*I++||(l=2*getchar(),i+=O>8
?o:O?O:o+1,o=O>9,O=-1,I="t8B~pq`",l>0)?main(l/2):
printf("%d\n",--i);}
```

# Makefile

- `make` detects automatically which pieces of a large program need to be recompiled and compile them.
- Need a `makefile` file to describe dependencies and compilation commands:

```
target: dependency list
<TAB>      Unix commands
```

- A first (bad) example:

```
a.out: test.c fact.c fact.h
        gcc test.c fact.c
```

Running the command `$ make a.out`
will create the executable `a.out` by the command

```
$ gcc test.c fact.c
```

# A better makefile

```
test.out: test.o fact.o
	gcc -o test.out test.o fact.o

test.o: test.c fact.h
	gcc -o test.o -c test.c

fact.o: fact.c
	gcc -o fact.o -c fact.c
```

Without parameter, `make` uses the first target:

```
$ make
gcc -o test.o -c test.c
gcc -o fact.o -c fact.c
gcc -o test.out test.o fact.o
```

If we change only one file:

```
$ touch fact.c ; make
gcc -o fact.o -c fact.c
gcc -o test.out test.o fact.o
```

```
$ touch fact.h ; make
gcc -o test.o -c test.c
gcc -o test.out test.o fact.o
```

# Additional targets

- `all` (first one) to group all executables,
- `clean` removes all intermediary files,
- `fclean` removes everything generated files.

```
all: test.out
test.out: test.o fact.o
	gcc -o test.out test.o fact.o
test.o: test.c fact.h
	gcc -o test.o -c test.c
fact.o: fact.c
	gcc -o fact.o -c fact.c
.PHONY: clean fclean
clean:
	rm *.o
fclean: clean
	rm test.out
```

(`.PHONY` to tell `make` which target are not files to be built)

# Macros

- `MACRO = whatever` ; later on, `$(MACRO)` is `whatever`.
- `CC` for the name of the compiler we use,
- `CFLAGS` for the options of compilation,
- `LDFLAGS` for the options of the link edition step,
- several used by make for automatic rules

```
CC = gcc
CFLAGS = -ansi -Wall -pedantic -c
LDFLAGS =
EXEC = test.out
all: $(EXEC)
test.out: test.o fact.o
	$(CC) -o test.out test.o fact.o $(LDFLAGS)
test.o: test.c fact.h
	$(CC) -o test.o test.c $(CFLAGS)
fact.o: fact.c
	$(CC) -o fact.o fact.c $(CFLAGS)
```

```
.PHONY: clean fclean
clean:
	rm *.o
fclean: clean
	rm $(EXEC)
```

# Predefined macros

- $@ stands for the name of the target,
- $< gives the name of the first dependency,
- $^ is the list of all dependencies,
- $* designs the target file without any suffix.

The previous makefile ca be written as follows:

```
CC = gcc
CFLAGS = -ansi -Wall -pedantic -c
LDFLAGS =
EXEC = test.out
.PHONY: clean fclean
all: $(EXEC)
test.out: test.o fact.o
        $(CC) -o $@ $^ $(LDFLAGS)
test.o: test.c fact.h
        $(CC) -o $@ $< $(CFLAGS)
fact.o: fact.c
        $(CC) -o $@ $< $(CFLAGS)
clean:
        rm *.o
fclean: clean
        rm $(EXEC)
```

# Generic rules

```
CC = gcc
CFLAGS = -ansi -Wall -pedantic -c
LDFLAGS =
EXEC = test.out
.PHONY: clean fclean
all: $(EXEC)
# rule to construct the executive files
% .out: %.o
        $(CC) -o $@ $^ $(LDFLAGS)
# rule to construct object files
% .o: %.c
        $(CC) -o $@ $(CFLAGS) $<
test.out: test.o fact.o
test.o: test.c fact.h
fact.o: fact.c
clean:
        rm *.o
fclean: clean
        rm $(EXEC)
```

# Make can do more than compilation...

- Useful to any task where files have to be updated from others whenever they change.

```
PLOT   = gnuplot
VIEWER = evince
.PHONY: clean fclean usage showplot

usage:
        @echo 'How to use make for these programs:'
# (...)

data.dat: test.out
        ./test.out > data.dat
# here plot.txt contains the command for gnuplot
plot.ps: plot.txt data.dat
        $(PLOT) <plot.txt
        @echo plot.ps generated

showplot: plot.ps
        $(VIEWER) plot.ps &
```

- @ before a command: the command is not printed
- - before a command: make does not stop if an error occurs,

# Multiple inclusions

main.c

```
/* whatever */
# include "a.h"
# include "b.h"

/* whatever */
int main(){
/*
 * using functions
 * and structures
 * in a.h and b.h
 * ( not tata1 ! )
 */
  return 0;
}
```

a.h

```
# include "b.h"
struct toto tata2();
int tata3(int);
```

a.c

```
# include "b.h"
int tata1(int n){
/* code */
}
struct toto tata2(){
/* code */
}
int tata3(int n){
/*code using tata1*/
}
```

b.h

```
struct toto {
/* definition */ }
int titi1(int);
struct toto titi2();
```

b.c

```
/* whatever */
# include "b.h"

int titi1(int n){
/* code */
}
struct toto titi2(){
/* code */
}
```

# Use include guards !

```
#ifndef NAME_FILE_H
#define NAME_FILE_H

/* definition of structures */

/* declaration of (extern) global variables */

/* declaration of functions */

#endif /* NAME_FILE_H */
```

- Files are included only once !

# Function pointers

- Same as the prototype, just adding a *,
- Must declare the returned type and the type of the parameters,
- Warning: be cautious with priorities !
  - `int (*pf)(int,int)` is a pointer on a function that takes two integers as input and outputs one.
  - `int *f(int,int)` is a function that takes two integers as input and outputs a pointer on integer.
- One can use typedef:

```
# include <stdio.h>

typedef int fct_t(int);
typedef fct_t* fctpt_t;
typedef int (*fctpt2_t)(int);

fct_t toto;
```

```
int main(){
  fctpt_t pf=toto;
  fctpt2_t pf2=toto;
  printf("t1:%d\nt2:%d\n"
         ,(*pf)(5),(*pf2)(5));
  return 0;
}
int toto(int a){
  return a;
}
```