Les projets en C

ICC-C Cours 9

Paramètres

- Les paramètres d'une fonction sont des "entrées" et sa valeur de retour est une "sortie"
- En mathématiques
  $f : \mathbb{R} \to \mathbb{R},\ \ f(x) = x^2$
- On peut calculer f en la valeur 4 et on obtient $f(4) = 16$
- En C aussi - on retourne la valeur calculée

```c
#include <stdio.h>

double f(double x)
{
    return x * x;
}

int main()
{
    printf("f(%.2lf) = %.2lf\n",
           4.0, f(4.0));
}
```

Affiche: f(4.00) = 16.00

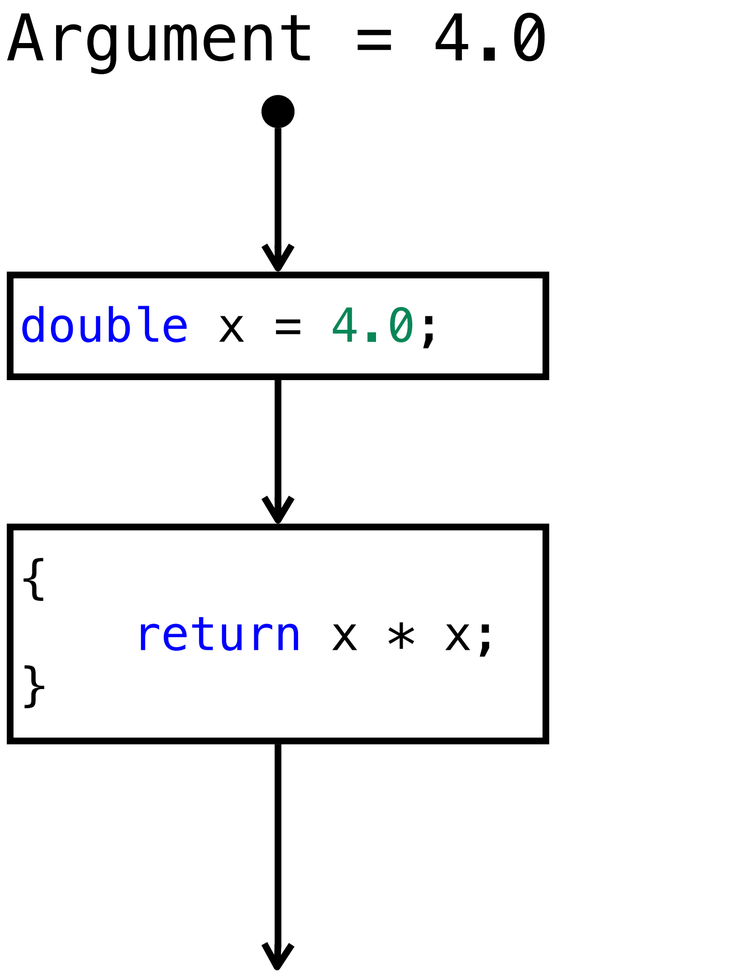
Appel de fonction

```
double f(double x)
{
    return x * x;
}
```

```
f(4.0)
```

1. “Les paramètres reçoivent la valeur des l’arguments” (passage par valeur)

2. “Le corps de la fonction est exécuté”



Paramètres “de sortie”

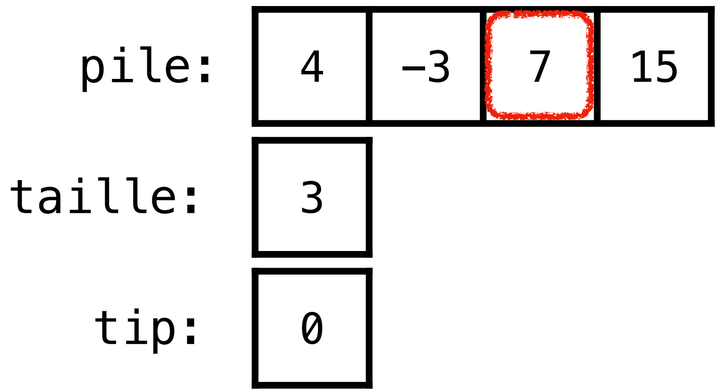
- En C on peut avoir des “paramètres de sortie”
- La fonction `pop` met à l’adresse contenue dans `p_objet` l’élément du haut de la pile
- `pop` **ne modifie pas** `p_objet`, mais l’emplacement où il pointe!
- Cette fonction a un effet secondaire

```c
int pile[100];
int taille, taille_max;

int pop(int *p_objet)
{
    if (taille > 0)
    {
        taille -= 1;
        *p_objet = pile[taille];

        // Opération réussie
        return 1;
    }
    // Échec, la pile est vide
    return 0;
}
```

Appel de fonction

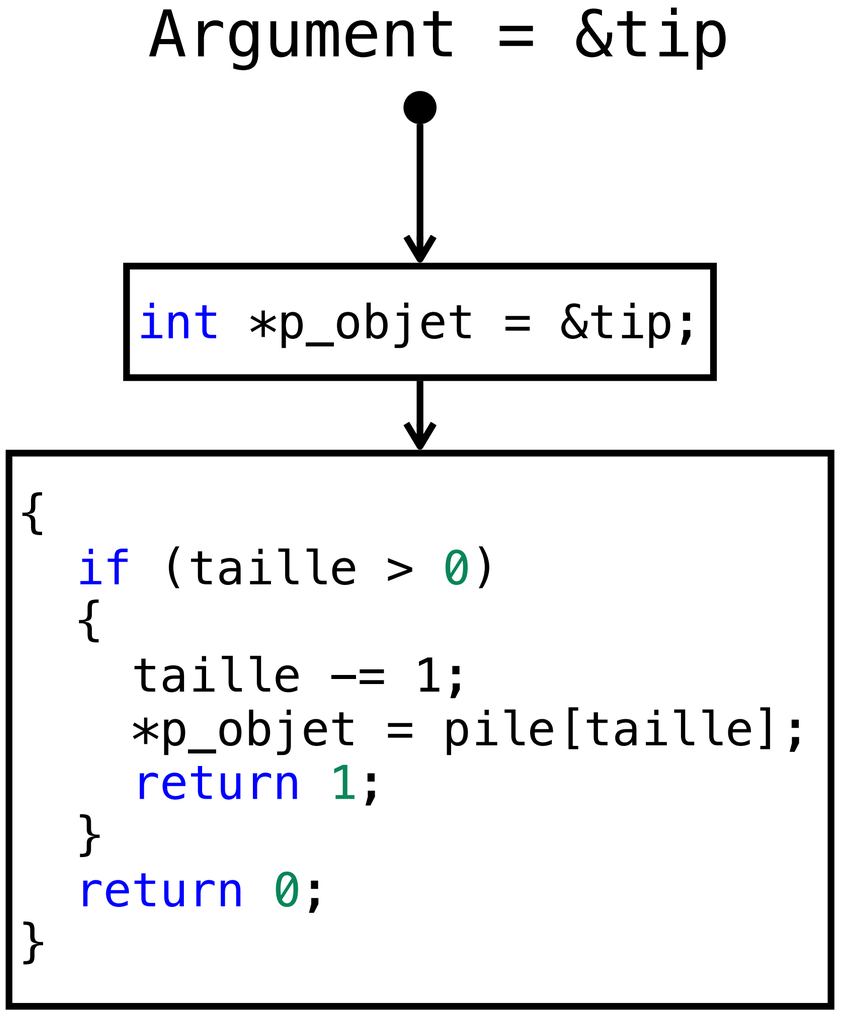
```
int pile[100];
int taille, taille_max;

int pop(int *p_objet)
{
    ...
}
```

```
int tip = 0;
pop(&tip);
```



1. passage par valeur

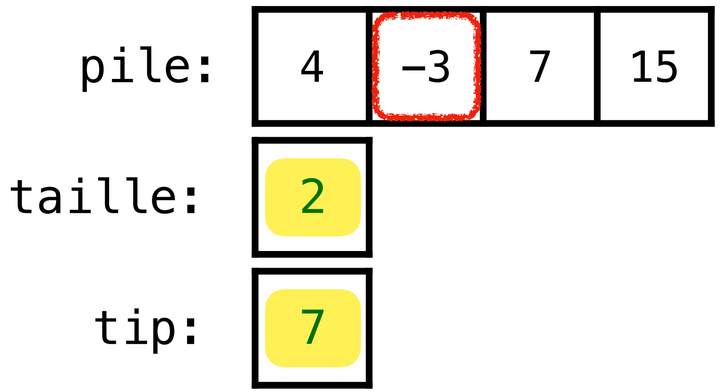
2. “Le corps de la fonction est exécuté”



Appel de fonction

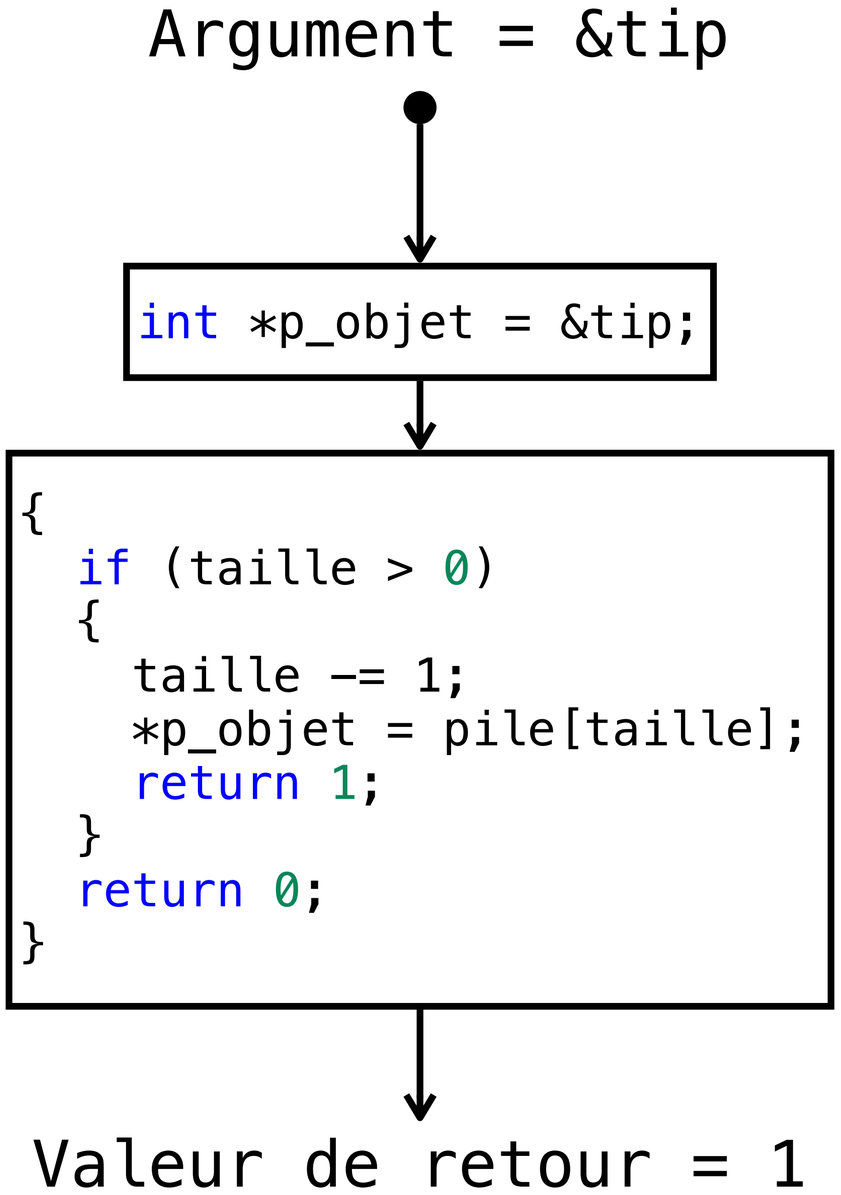
```
int pile[100];
int taille, taille_max;

int pop(int *p_objet)
{
    ...
}
```

```
int tip = 0;
pop(&tip);
```



1. passage par valeur

2. “Le corps de la fonction est exécuté”



Paramètre pointeur vers scalaire ou tableau?

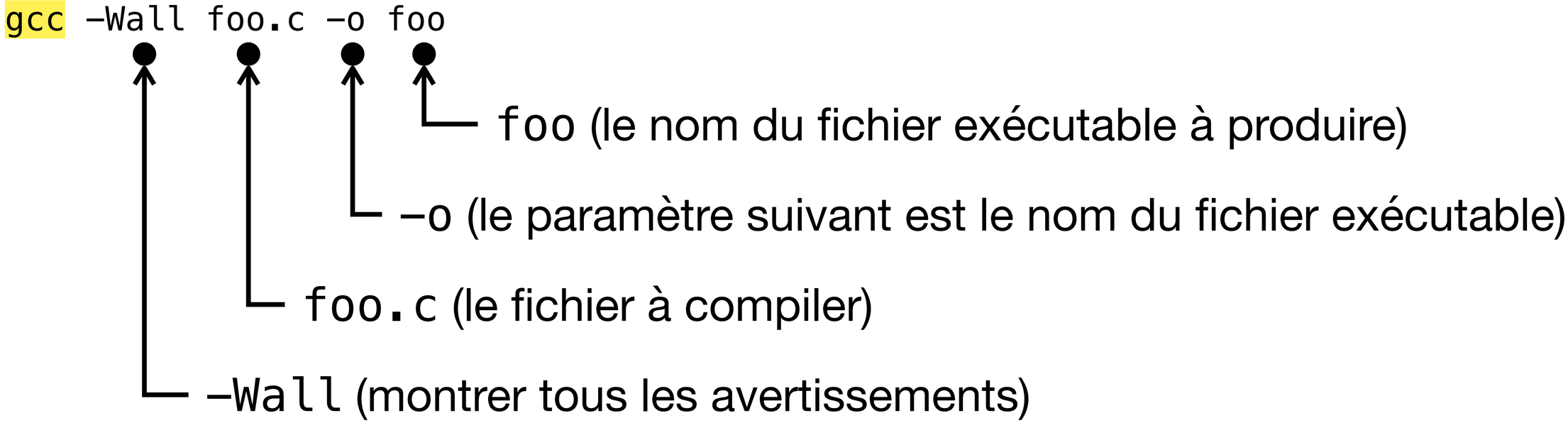
```c
void incrementer(int *param)
{
  (*param)++;
}
```

```c
void incrementer3(int *param)
{
  for (int i = 0; i < 3; i++)
    param[i]++;
}
```

- Comment peut-on distinguer `incrementer` et `incrementer3` ?
- **Réponse**: ⚠️ on ne peut pas!
- La déclaration d'une fonction avec un paramètre pointeur **n'indique pas** si elle modifie un élément ou plusieurs!
- L'accès direct à la mémoire est très (trop?) puissant
- Peut mener à des bugs, failles de sécurité, etc.

Paramètres d’un programme

- Nous avons vu comment lire depuis stdin
- Certains programmes reçoivent des arguments au lancement!



Lire les paramètres

- C’est avec une nouvelle signature de la fonction `main`

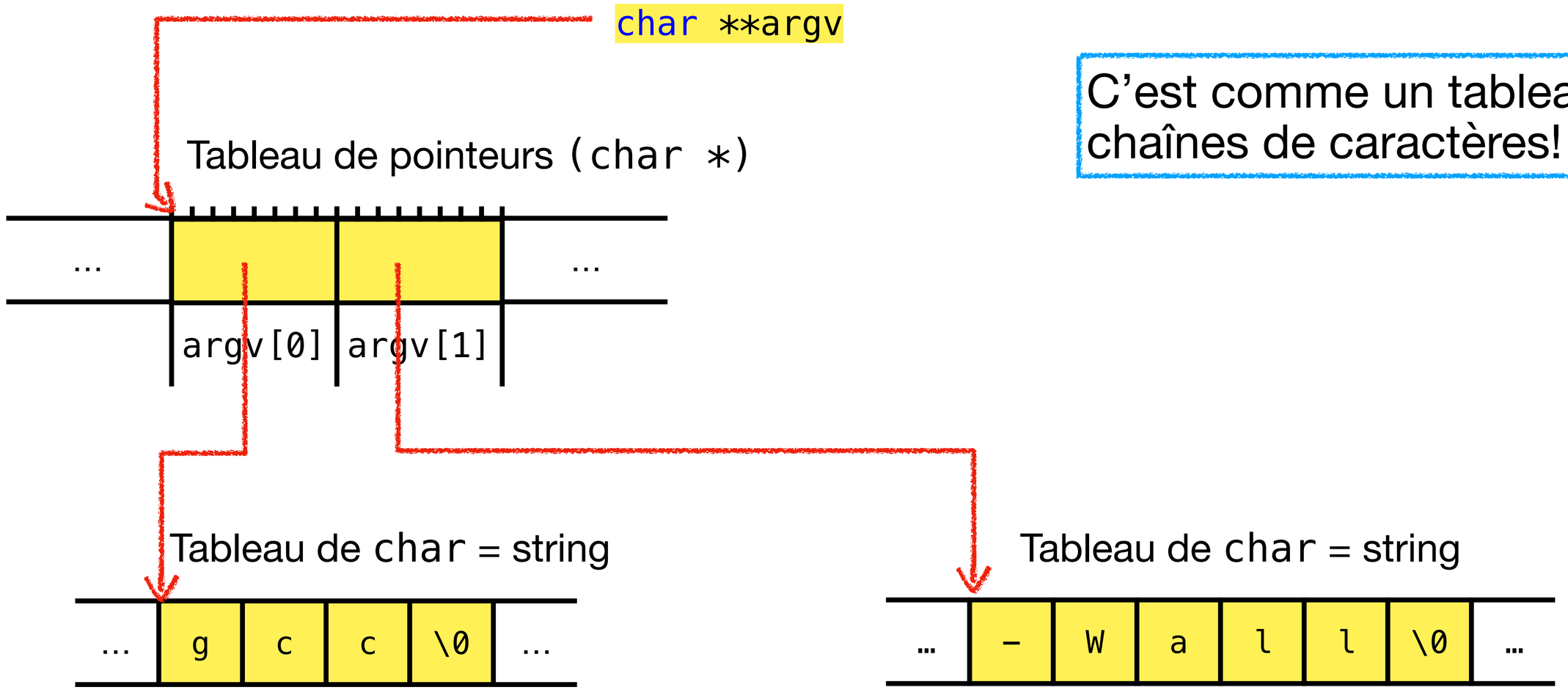
D’habitude on écrit:

```
int main()
{
  ...
  return 0;
}
```

Mais on peut écrire aussi:


```
int main(int argc, char **argv)
{
  ...
  return 0;
}
```

Pointeur double vers char



C’est comme un tableau de chaînes de caractères!

Programme qui affiche ses arguments

- Premier argument = nombre d'arguments en ligne de commande
- Deuxième argument = tableau de strings = les arguments fournis dans l'ordre
- `argv[0]` = la commande utilisée pour lancer le programme

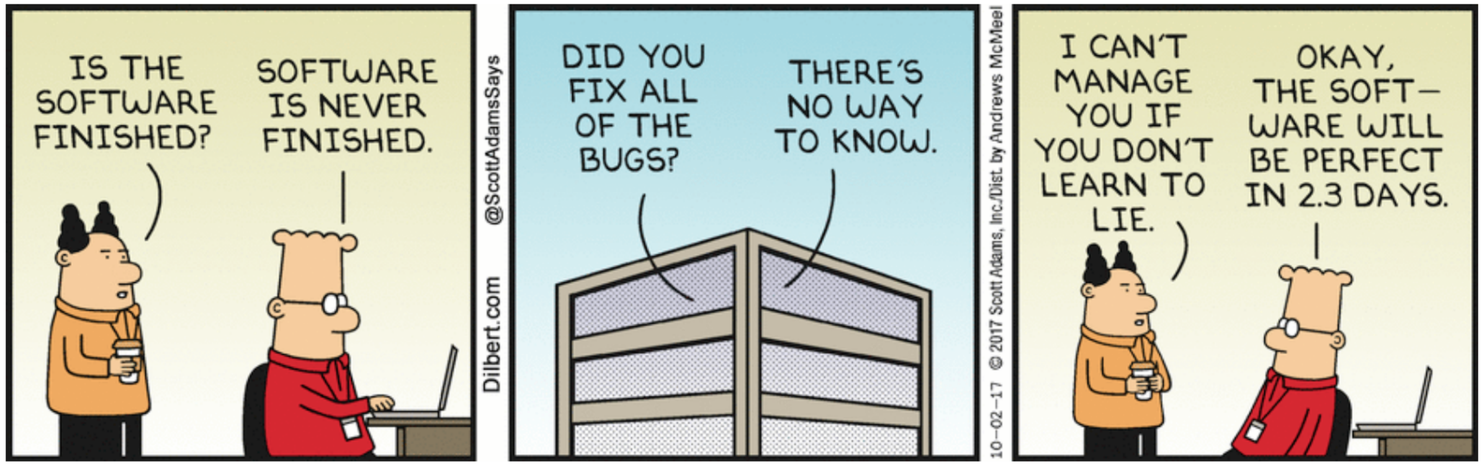
```c
#include <stdio.h>

int main(int argc, char **argv)
{
    for (int i=0; i<argc; i++)
    {
        printf("Argument %d: %s\n”,
                i, argv[i]);
    }
    return 0;
}
```

Programme qui affiche ses arguments

```
❯ ./args des arguments sans espace "sauf lui"
Argument 0: ./args
Argument 1: des
Argument 2: arguments
Argument 3: sans
Argument 4: espace
Argument 5: sauf lui
```

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    for (int i=0; i<argc; i++)
    {
        printf("Argument %d: %s\n”,
                i, argv[i]);
    }
    return 0;
}
```

Projets en C


IS THE SOFTWARE FINISHED?
SOFTWARE IS NEVER FINISHED.
Dilbert.com @ScottAdamsSays
DID YOU FIX ALL OF THE BUGS?
THERE'S NO WAY TO KNOW.
10-02-17 © 2017 Scott Adams, Inc./Dist. by Andrews McMeel
I CAN'T MANAGE YOU IF YOU DON'T LEARN TO LIE.
OKAY, THE SOFT-WARE WILL BE PERFECT IN 2.3 DAYS.

Structure d’un projet

- Des fichiers `.c` (code) et `.h` (header = entête)
- Un fichier `.h` souvent sert de “contrat” pour les fonctions définies dans le fichier `.c` du même nom
- On ne peut pas tout simplement tout refiler à gcc
- Il faut comprendre plus en détail la “compilation”…

```
crush
├── Makefile
├── const.h
├── crush.c
├── display.c
├── display.h
├── engine.c
├── engine.h
├── types.h
├── util.c
└── util.h
```

La compilation

- Un processus plutôt complexe
- Plusieurs étapes = passes sur le code source, dont:
  - Preprocessing
  - Analyse syntaxique (*parsing*)
  - Compilation proprement-dite
  - Assemblage
  - Linking

Étape 1: Préprocesseur

- Que veut dire `#include` ?
- Les lignes qui commencent par # sont traitées par le préprocesseur
- Les directives du préprocesseur s'appellent des "macroinstructions" ou "macros"

`define`, `undef`, `include`, `if`, `ifdef`, `ifndef`, `else`, `elif`, `el ifdef`, `elifndef`(C23), `endif`, `line`, `embed`(C23), `error`, `war ning`(C23), `pragma`

```c
#include <stdio.h>

int main()
{
    printf("Bonjour, ICC!\n”);
    return 0;
}
```

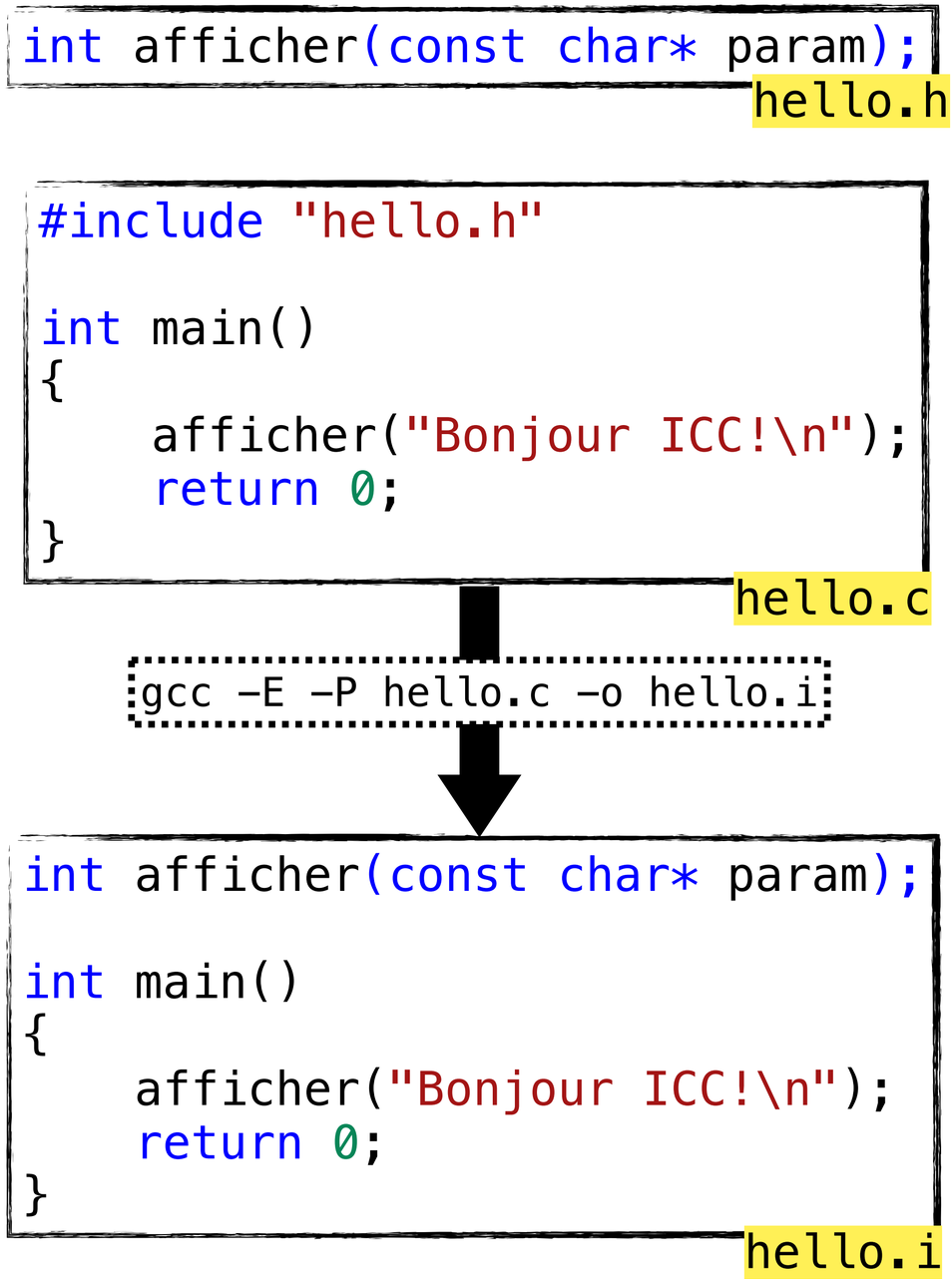
Appeler uniquement le préprocesseur

```
gcc –E –P hello.c –o hello.i
```

- “Exécute” les macros
- Produit un nouveau fichier C sans directives #
- Une seule passe sur le fichier .c

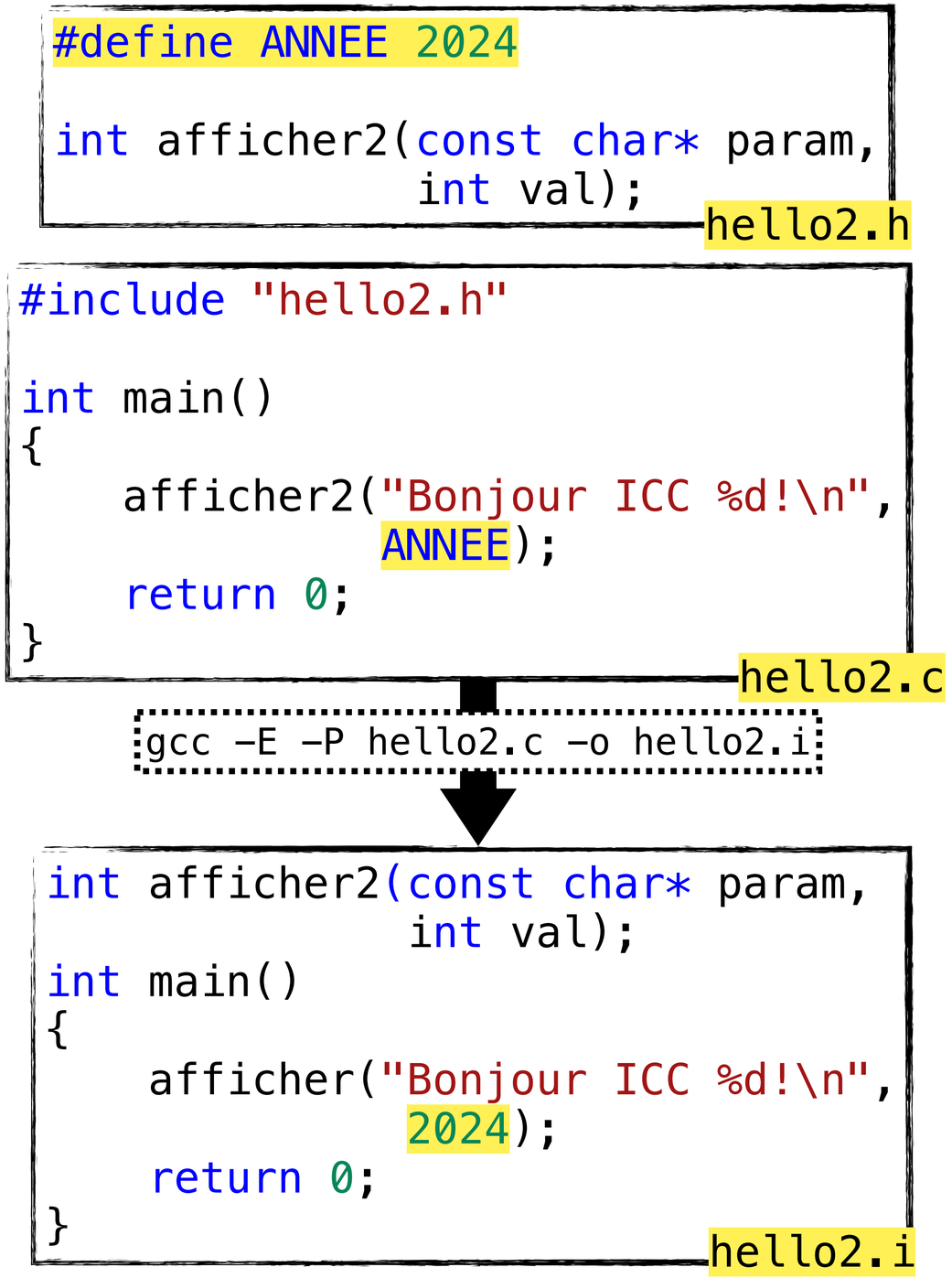
#include

- **Paramètre:** le fichier à inclure, e.g., `hello.h`
- **Action:** comme si on avait copié-collé le contenu de `hello.h` à cet endroit
- Un fichier `.h` contient souvent:
  - Des déclarations de fonctions
  - Des constantes
  - ???



#define

- **Paramètres:** Une macro-constante et sa valeur
- **Action:** Remplacer la macro-constante par sa valeur partout dans le fichier
- Ce n’est pas une “vraie” constante, c’est comme si on écrivait la valeur partout



#define

- **Paramètres:** Une macro-fonction, ses paramètres, et sa définition
- **Action:** Remplacer partout dans le code le faux-appel de macro-fonction par sa définition
- Ce n’est pas une vraie fonction!

```c
#define ANNEE 2024

#define afficher3(str, val) printf(str, val)
```

hello3.h

```c
#include "hello3.h"

int main()
{
    afficher3(“Bonjour ICC %d!\n",
              ANNEE);
    return 0;
}
```

hello3.c

```
gcc -E -P hello3.c -o hello3.i
```

```c
int main()
{
    printf("Bonjour ICC %d!\n",
           2024);
    return 0;
}
```

hello3.i

Exemples de macro-fonctions “acceptables”

```
#define max(x, y) ((x) >= (y) ? (x) : (y))

#define abs(x) ((x) < 0 ? -(x) : (x))
```

- C’est mieux de mettre des parenthèses autour des faux-paramètres, car ils peuvent représenter des expressions compliquées

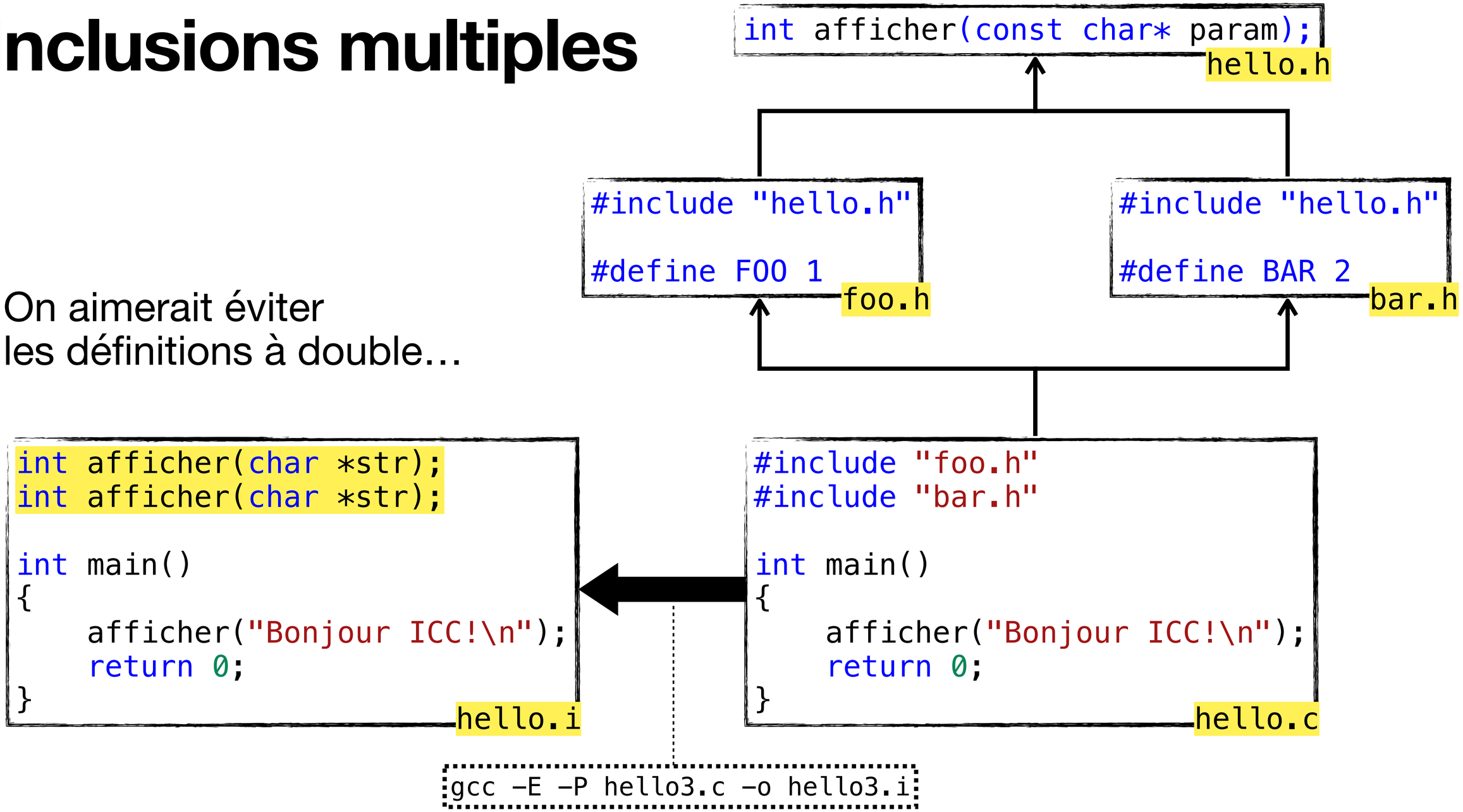
`abs(<expr_compliquée>)` se traduit en

`(<expr_compliquée>) < 0 ? -(<expr_compliquée>) : (<expr_compliquée>)`

- ⚠️ En faisant ce raccourci on va évaluer `<expr_compliquée>` **deux** fois:
  - Une fois pour tester le signe,
  - Une deuxième fois pour calculer la valeur.

Inclusions multiples

On aimerait éviter
les définitions à double…



#ifndef–#endif

- **Paramètres:** Un macro-symbole
- **Action:** Le code compris entre `#ifndef` et `#endif` est pris en compte si le symbole n'est pas défini (avec `#define`)
- Utilisé afin d'empêcher des inclusions multiples
- Dans cet exemple, la *première* inclusion définira `HELLO_H` et la déclaration de `afficher()` sera prise en compte
- Pour la *deuxième* inclusion, `HELLO_H` sera déjà défini, donc la déclaration ne sera pas répétée!

```
#ifndef HELLO_H
#define HELLO_H
int afficher(const char* param);
#endif
```

hello.h

Étape 2: Analyse syntaxique

Parsing

- Un langage de programmation a lui aussi une grammaire
- La grammaire décrit la syntaxe du langage
- La grammaire du C: https://www.quut.com/c/ANSI-C-grammar-y.html
- Le parser vérifie que la grammaire soit respectée
- Exemple:
  - on ne peut pas écrire `return 3+; // error: expected expression`
  - Le préprocesseur ne s'en plaindra pas!

Étape 3: Compilation

- Traduit le code C en assembleur
- Si on veut voir le résultat de cette étape:

```
gcc -S hello.c -o hello.S
```

```
  .section  __TEXT,__text,regular,pure_instructions
  .build_version macos, 14, 0  sdk_version 14, 4
  .globl  _main                                   ; -- Begin func
  .p2align  2
_main:                                            ; @main
  .cfi_startproc
; %bb.0:
  sub  sp, sp, #32
  .cfi_def_cfa_offset 32
  stp  x29, x30, [sp, #16]                        ; 16-byte Folded S
  add  x29, sp, #16
  .cfi_def_cfa w29, 16
  .cfi_offset w30, -8
  .cfi_offset w29, -16
  mov  w8, #0
  str  w8, [sp, #8]                               ; 4-byte Folded Sp
  stur  wzr, [x29, #-4]
  adrp  x0, l_.str@PAGE
  add  x0, x0, l_.str@PAGEOFF
  bl  _afficher
  ldr  w0, [sp, #8]                               ; 4-byte Folded Re
  ldp  x29, x30, [sp, #16]                        ; 16-byte Folded R
  add  sp, sp, #32
  ret
  .cfi_endproc
                                                  ; -- End function
  .section  __TEXT,__cstring,cstring_literals
l_.str:                                           ; @.str
  .asciz  "Bonjour ICC!\n"

.subsections_via_symbols
```

Étape 4: Assemblage

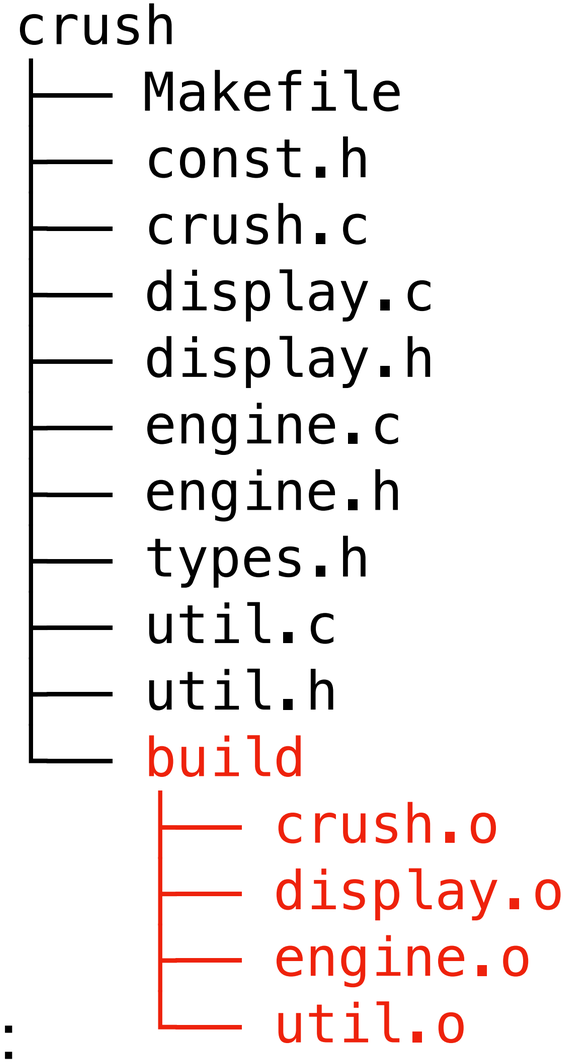
- Produit un fichier objet `.o`
- C’est la dernière étape avant la génération de l’exécutable
- Noter que les fonctions/symboles non définis ne gênent pas
- La fonction `afficher` n’a pas été définie, mais la compilation fonctionne
- Pour arriver directement à cette étape:

```
gcc -c hello.c -o hello.o
```

Étape 5: Création des liens

Linking

- Chaque fichier `.c` a été compilé en un fichier `.o`
- Pour notre exemple nous avons quatre fichiers
- On doit tous les “lier” ensemble avec certaines bibliothèques requises pour créer l’exécutable
- On spécifie les bibliothèques avec `-l`: `-lncurses`, `-lm`, etc.
- **Tous les symboles doivent être définis!**
- Le programme de *link* s’appelle `ld`, mais `gcc` fonctionne aussi:




```
gcc build/engine.o build/display.o build/util.o build/crush.o -lncurses -lm -o crush
```

Build systems

- TL;DR: Il faut générer les fichiers objet (4) + linking pour créer l'executable (5)
- On n'a pas envie de tout taper à la main
- On définit les commandes pour construire des cibles dans un `Makefile`
- L'utilitaire `make` comprendra ce fichier et saura exécuter les bonnes commandes
- Il suffit ensuite de taper `make`

```
CC=gcc -c
LD=gcc

all: build_dir crush

build_dir:
	mkdir -p build

engine.o: engine.c engine.h const.h types.h
	${CC} -o build/engine.o engine.c

display.o: display.c display.h const.h types.h
	${CC} -o build/display.o display.c

util.o: util.c util.h const.h types.h
	${CC} -o build/util.o util.c

crush.o: crush.c engine.h display.h util.h
const.h types.h
	${CC} -o build/crush.o crush.c

crush: engine.o display.o util.o crush.o
	${LD} build/engine.o build/display.o
build/util.o build/crush.o -lncurses -lm -o crush

pack:
	mkdir -p crush
	cp *.c crush
	cp *.h crush
	cp Makefile crush
	zip -r crush.zip crush/
	rm -rf crush

clean:
	rm -f build/* crush
	rmdir build
```

Supplément

Exemple de grammaire pour les parenthèses

Le langage des parenthèses ( )

- À chaque parenthèse ouverte correspond une parenthèse fermée
- La suite de parenthèses comprise entre ces deux parenthèses est elle-même bien formée
- Langage algébrique (*context-free language*)
- Description constructive — a partir d'une suite **bien formée** *S* (qui est dans ce langage) comment peut-on générer d'autres suites?
- La grammaire nous donne des règles de production (*production rules*)
  - Comment aller de *S* à une autre suite valide

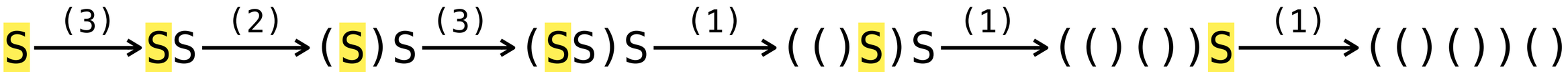
Supplément

Exemple de grammaire pour les parenthèses

Règles de production

1. $S \to ()$
2. $S \to (S)$
3. $S \to SS$

- On veut prouver que (( ) ( )) ( ) est dans le langage
- On commence par un symbole non-terminal S
- On applique une des trois règles à chaque étape



QED