Les projets en C

ICC-C Cours 10

Paramètres

- Les paramètres d'une fonction sont des "entrées" et sa valeur de retour est une "sortie"
- Fonctions mathématiques
  $f : \mathbb{R} \to \mathbb{R}, \;\; f(x) = x^2$
- On peut calculer f de 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
```

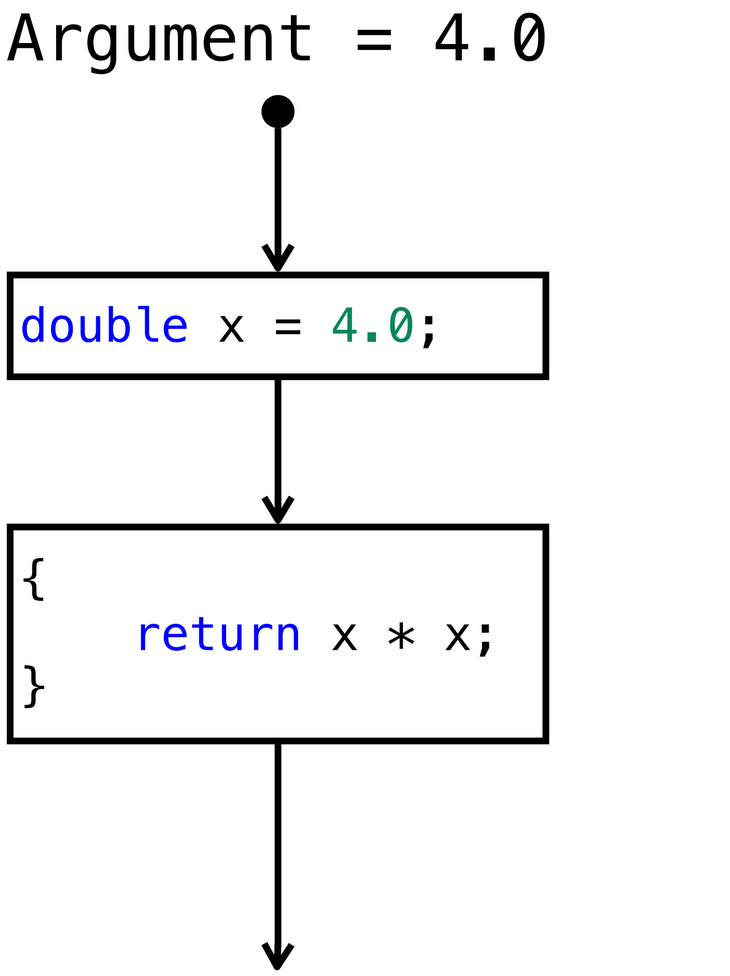
Appel de fonction

```
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”

- La fonction `pop` met à l’adresse contenue dans `valeur` l’élément du haut de la pile
- `pop` modifie l’emplacement où `valeur` pointe et change la liste
- Cette fonction a un effet secondaire

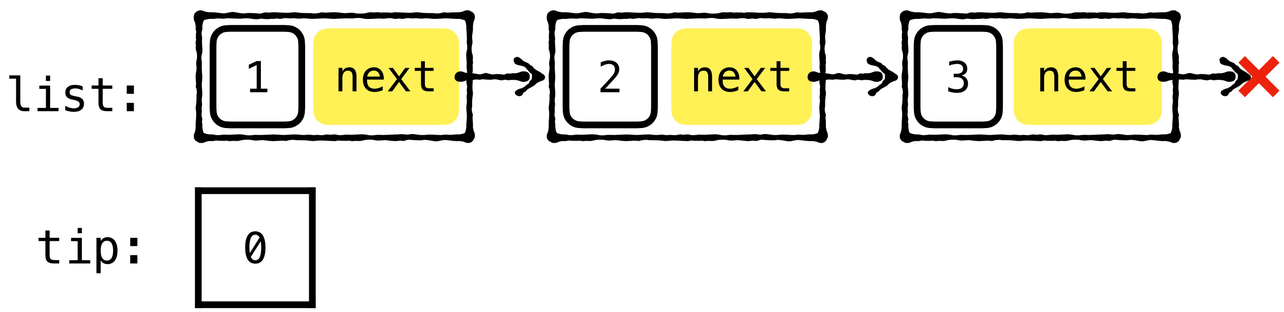
```c
int pop(struct list *plist, int *valeur)
{
    if (plist->head != NULL)
    {
        *valeur = plist->head->contenu;
    }
    return delete_head(plist);
}
```

Appel de fonction

```
struct list list;
...
int tip = 0;
pop(&list, &tip);
```

1. Passage par valeur

2. Le code de la fonction est exécuté



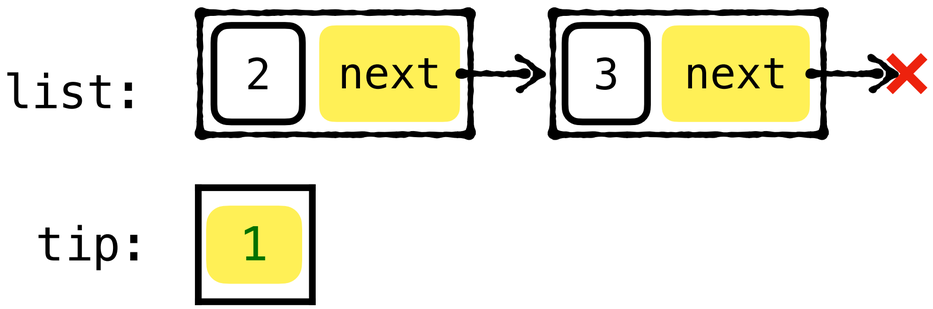
Arguments = &list, &tip

```
struct list *plist = &list;
int *valeur = &tip;
```

```
{
  if (plist->head != NULL)
  {
    *valeur = plist->head->contenu;
  }
  return delete_head(plist);
}
```

Appel de fonction

```
struct list list;
...
int tip = 0;
pop(&list, &tip);
```



1. Passage par valeur

2. Le code de la fonction est exécuté

Arguments = &list, &tip

```
struct list *plist = &list;
int *valeur = &tip;
```

```
{
  if (plist->head != NULL)
  {
    *valeur = plist->head->contenu;
  }
  return delete_head(plist);
}
```

Valeur de retour = 1

Paramètre pointeur vers scalaire ou tableau?

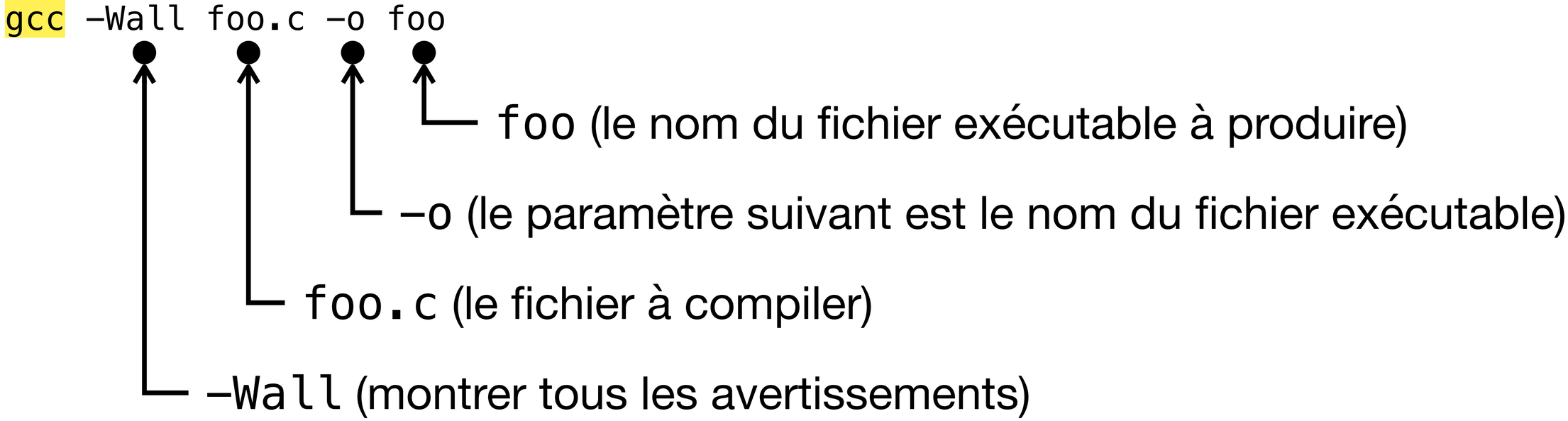
```c
void incrementer(int *param)
{
  (*param)++;
}
```

```c
void incrementer_tableau_3(int *param)
{
  for (int i = 0; i < 3; i++)
    param[i]++;
}
```

- Comment peut-on distinguer `incrementer` et `incrementer_tableau_3` ?
- **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

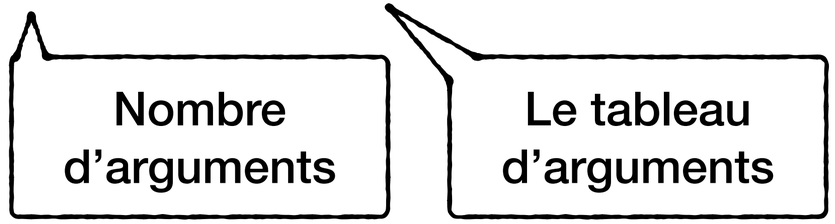
- C’est avec une nouvelle signature de la fonction `main`

D’habitude on écrit:

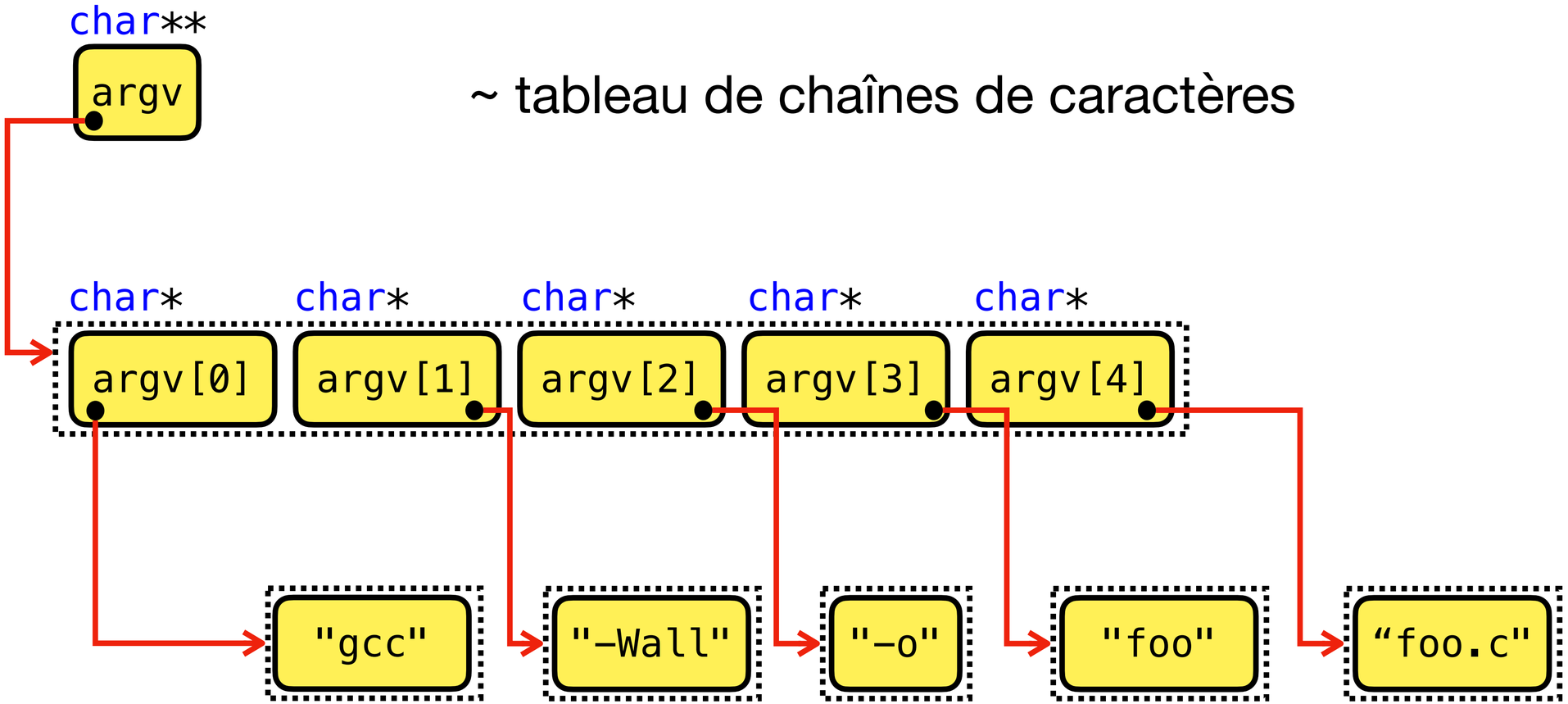
```
int main()
{
  ...
  return 0;
}
```

Mais on peut écrire aussi:

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



Pointeur double vers char



Programme qui affiche ses arguments

- `argc` = nombre d'arguments en ligne de commande
- `argv` = 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;
}
```

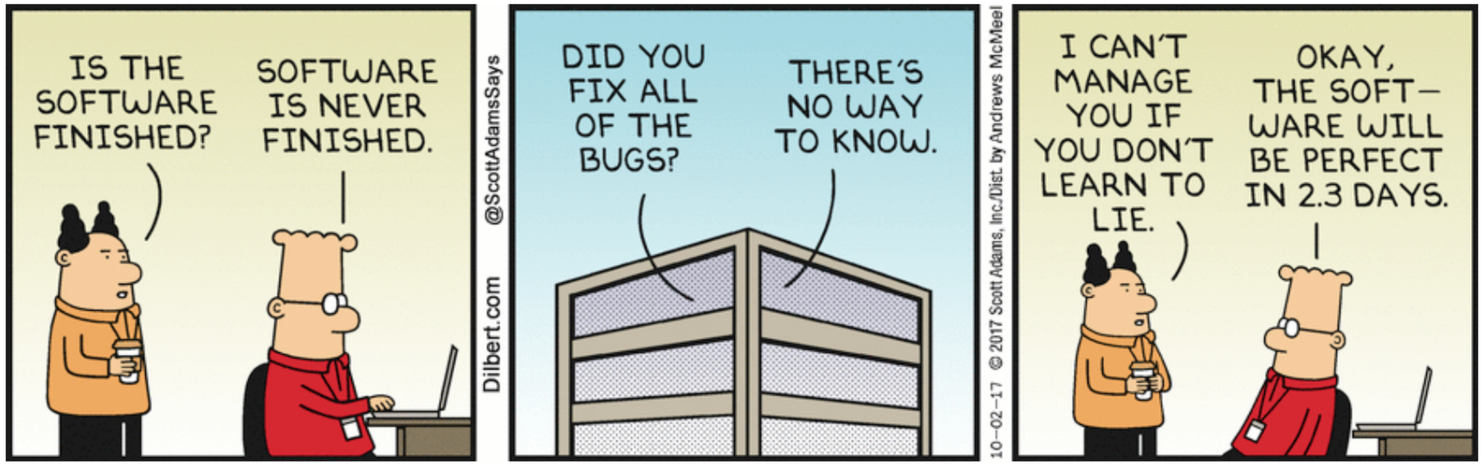
Programme qui affiche ses arguments

```
❯ ./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”…

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

La compilation

- Un processus plutôt complexe
- Plusieurs étapes = passes sur le code source, dont:
  - Preprocessing
  - Compilation proprement-dite
  - Assemblage
  - Création de liens

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"

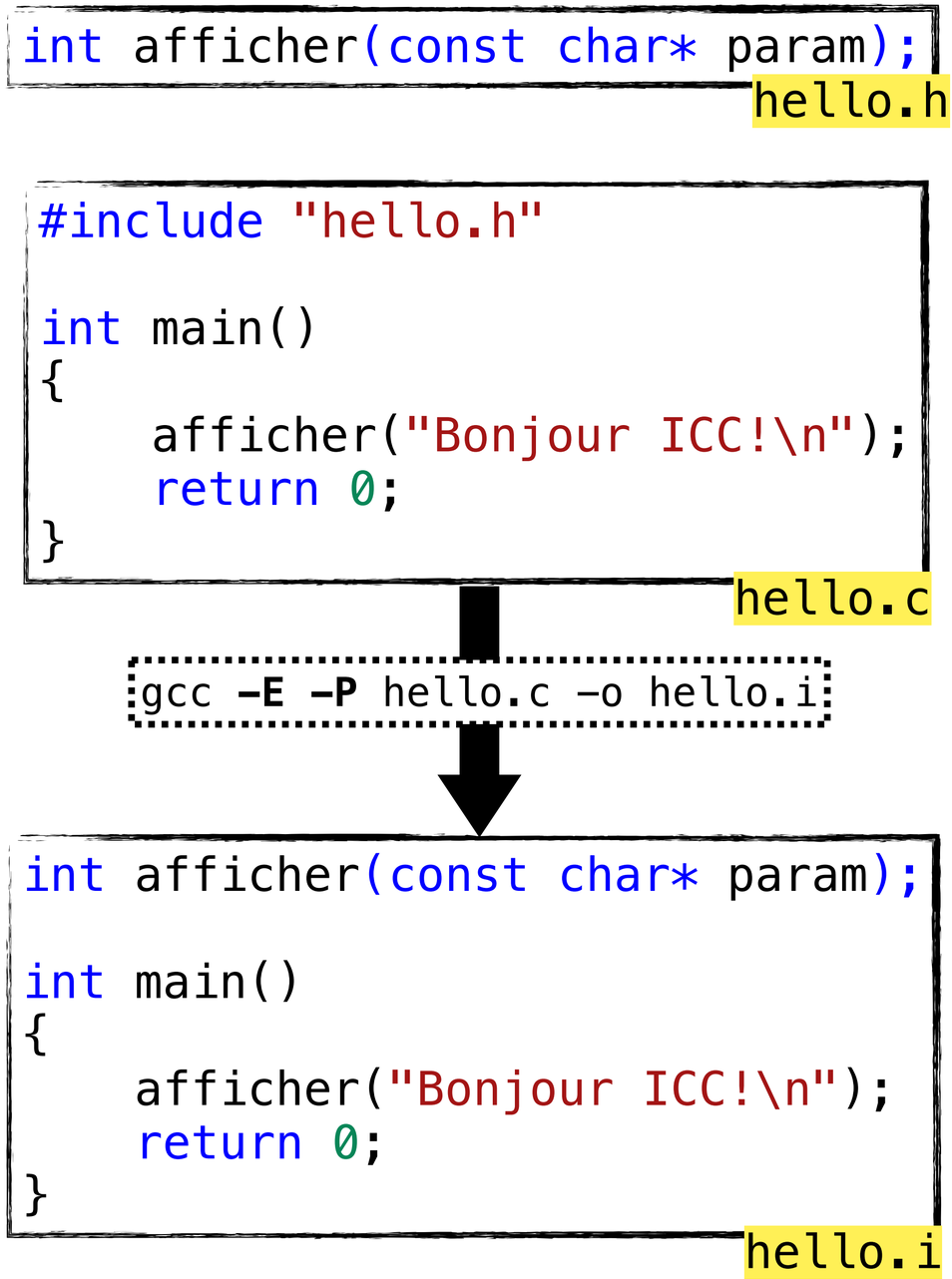
`define`, `undef`, `include`, `if`, `ifdef`, `ifndef`, `else`, `elif`, `elifdef`, `elifndef`(C23), `endif`, `line`, `embed`(C23), `error`, `warning`(C23), `pragma`

```c
#include <stdio.h>

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

#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 des déclarations de fonctions et des constantes
- Les guillemets “`hello.h`" = le fichier est dans le même dossier où on effectue la compilation
  (vs. `<stdio.h>` = répertoire système)



Étape 2: Compilation

- 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
- Pendant la compilation, le parser vérifie que la grammaire est bien respectée
- Exemple:
  - on ne peut pas écrire `return 3+; // error: expected expression`
  - Le préprocesseur ne s'en plaindra pas!

Étape 2: Compilation

- = “Translation”
- 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 function main
  .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 Spill
  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 Spill
  stur  wzr, [x29, #-4]
  adrp  x0, l_.str@PAGE
  add  x0, x0, l_.str@PAGEOFF
  bl  _afficher
  ldr  w0, [sp, #8]                       ; 4-byte Folded Reload
  ldp  x29, x30, [sp, #16]                ; 16-byte Folded Reload
  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 3: Assemblage

- Produit un fichier binaire “objet” `.o`
- C’est la dernière étape avant la génération de l’exécutable
- 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 (1-3):

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

Étape 3: Assemblage

- Définissons quand même notre fonction dans un autre fichier .c
- On peut aussi créer son objet

```
gcc -c afficher.c -o afficher.o
```

```c
int afficher(const char* param);
```

hello.h

```c
#include <stdio.h>
#include "hello.h"

int afficher(const char* param)
{
  return printf(
    "⚠️ %s ⚠️\n",
    param);
}
```

afficher.c

Étape 4: Création des liens

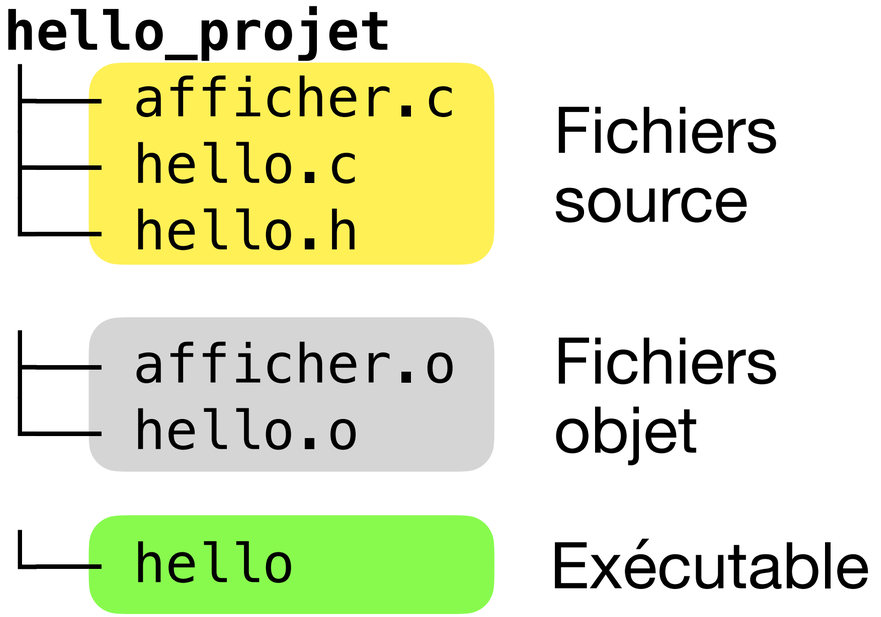
Linking

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

```
gcc hello.o afficher.o -o hello
```

Résumé

- Dans notre projet il y a trois fichiers source: `hello.c`, `hello.h`, `afficher.c`
- Il faut d'abord générer les fichiers objet
  ```
  gcc -c afficher.c -o afficher.o
  gcc -c hello.c -o hello.o
  ```
- Et ensuite l'exécutable
  ```
  gcc hello.o afficher.o -o hello
  ```
- Et si on avait 30 fichiers source? Ou alors 200? 😨



Build tools ⚒️

- Nous avons besoin d’un outil pour gérer tout ça
- L’outil le plus répandu est (était?)

  `make`

- Il faut décrire les dépendances dans un format spécifique:
  - Un fichier `Makefile` (qui s’appelle `Makefile`)
- Il vit dans la “racine” du projet

Makefiles

- Contient des “cibles” (*targets*)
  = des tâches à exécuter
- Une cible a un identifiant suivi du caractère “:”
- Sur les lignes suivantes il y a les commandes à exécuter afin d’accomplir la tâche
- Les commandes sont précédées par le caractère d’indentation “tab”
- On exécute une cible avec la commande **make** suivie de l’identifiant de la cible

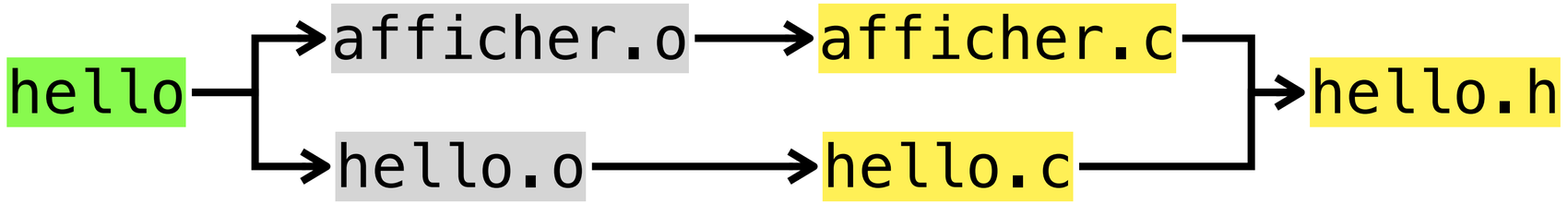
```
salut:
    echo “Comment ça va?”

compile_hello:
    gcc -c hello.c -o hello.o
```

```
make -s salut
Comment ça va?
```

Dépendances

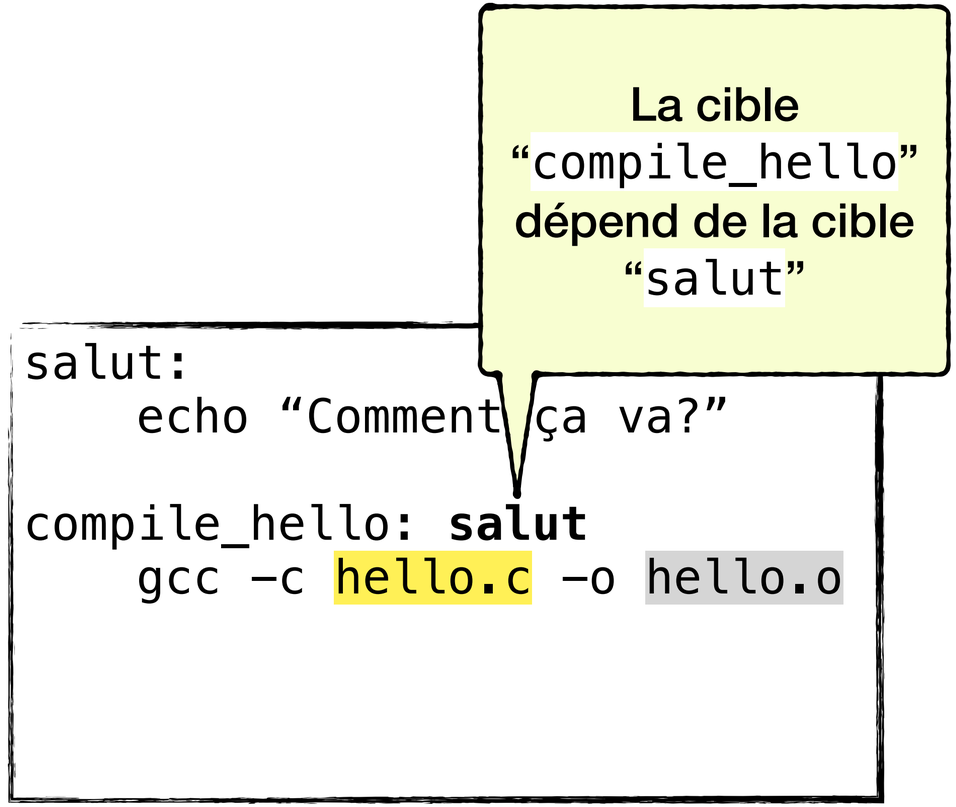
- Si on modifie un fichier source c'est encore plus compliqué…
- Souvenez-vous:
  dans `afficher.c` **et** dans `hello.c` il y a `#include "hello.h"`
- Si on modifie
  - `hello.h` il faudrait tout recommencer
  - `hello.c` il suffit de régénérer `hello.o` et `hello`
  - `afficher.c` il suffit de régénérer `afficher.o` et `hello`



"Dependency graph"

Makefiles
Dépendances

- Une cible A peut dépendre d'une autre cible B
  - Il suffit de l'indiquer sur la même ligne où on définit A (après le ":")
  - Quand on fait
    make A
    la cible B sera exécutée aussi




```
make -s compile_hello
Comment ça va?
gcc -c hello.c -o hello.o
```

Makefiles

- Nous pouvons définir les dépendances dans une Makefile
- Si aucune cible n’est indiquée, la première est exécutée
- Cette version exécute toujours toutes les étapes — mais si on n’a modifié qu’un seul fichier?

```
build: compile_hello compile_afficher
    gcc hello.o afficher.o -o hello

compile_afficher:
    gcc -c afficher.c -o afficher.o

compile_hello:
    gcc -c hello.c -o hello.o
```

make build

ou juste

make

Makefiles

- Une cible peut avoir le même nom qu'un fichier
- Les dépendances aussi
- make s'assure que chaque fichier dépendance existe
- Si un fichier-cible est plus récent que ses fichiers-dépendances, alors on suppose qu'il a déjà été généré

```
hello: hello.o afficher.o
    gcc hello.o afficher.o -o hello

afficher.o: afficher.c hello.h
    gcc -c afficher.c -o afficher.o

hello.o: hello.c hello.h
    gcc -c hello.c -o hello.o
```

```
make hello
make: `hello' is up to date.
```