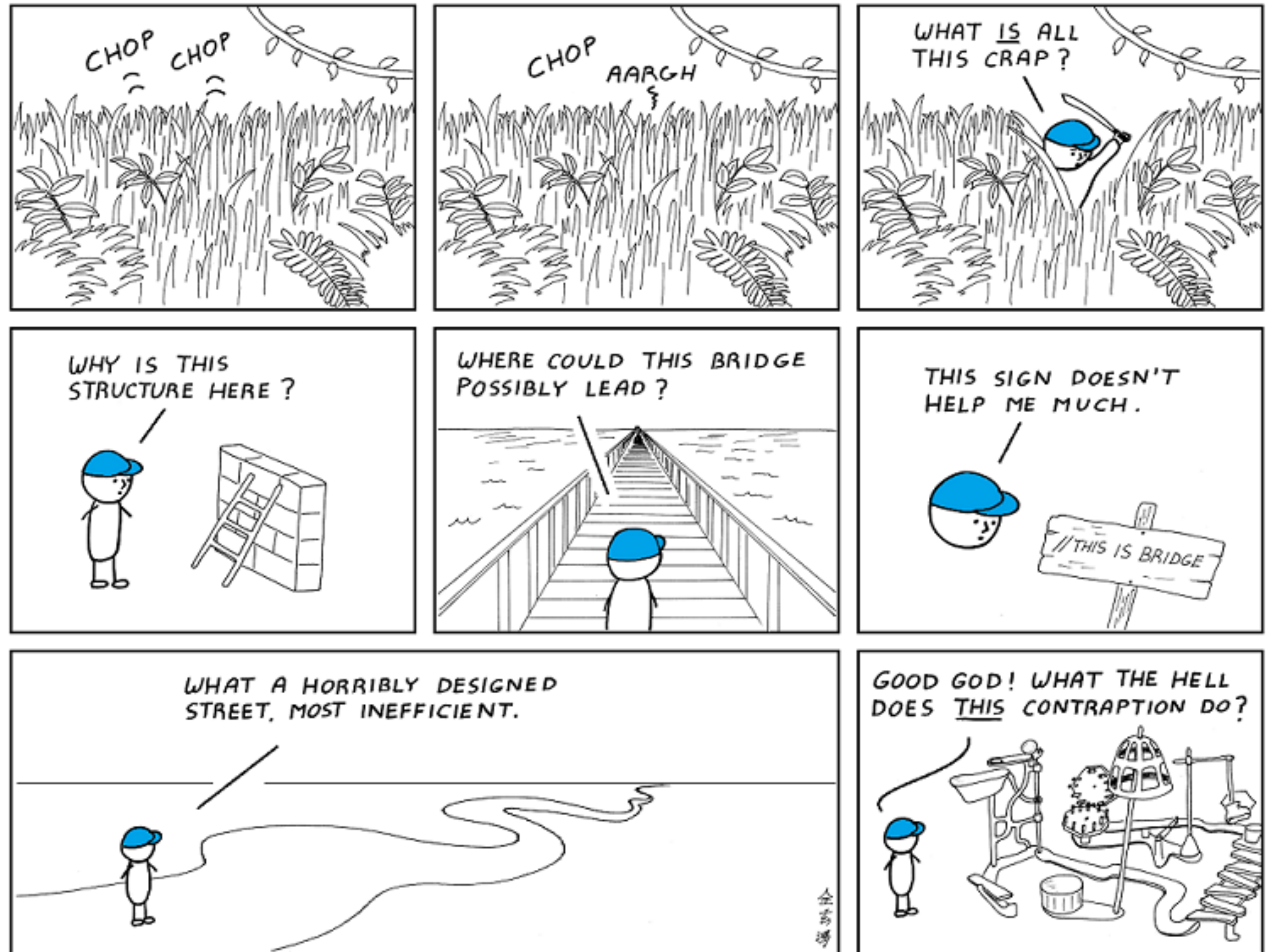


Fonctions



L'instruction for

Initialisation —
Exécuté une seule fois

- Lire un tableau d'entiers

La condition pour continuer

L'expression pour itérer

```
int indice = 0;
printf("Entrez %d valeurs : \n", nuits);
while (indice < nuits)
{
    scanf("%f", &heures[indice]);
    indice++;
}
```

```
int indice;
printf("Entrez %d valeurs : \n", nuits);
for (indice=0; indice < nuits; indice++)
{
    scanf("%f", &heures[indice]);
}
```

L'instruction do-while

- `do` instruction `while` (expression)
- Comme `while`, mais la condition est vérifiée à la fin de la boucle

```
while (n != 0)
{
    printf("Entrez un nombre : ");
    scanf("%d", &n);
    printf("Vous avez entré : %d\n", n);
}
```

```
do
{
    printf("Entrez un nombre : ");
    scanf("%d", &n);
    printf("Vous avez entré : %d\n", n);
} while (n != 0);
```

Boucles imbriquées

Nested loops

- On peut mettre une boucle dans une boucle!
- Par exemple, afficher toutes les paires

```
int vec[] = {1, 2, 3};
int size = 3;
int i, j;

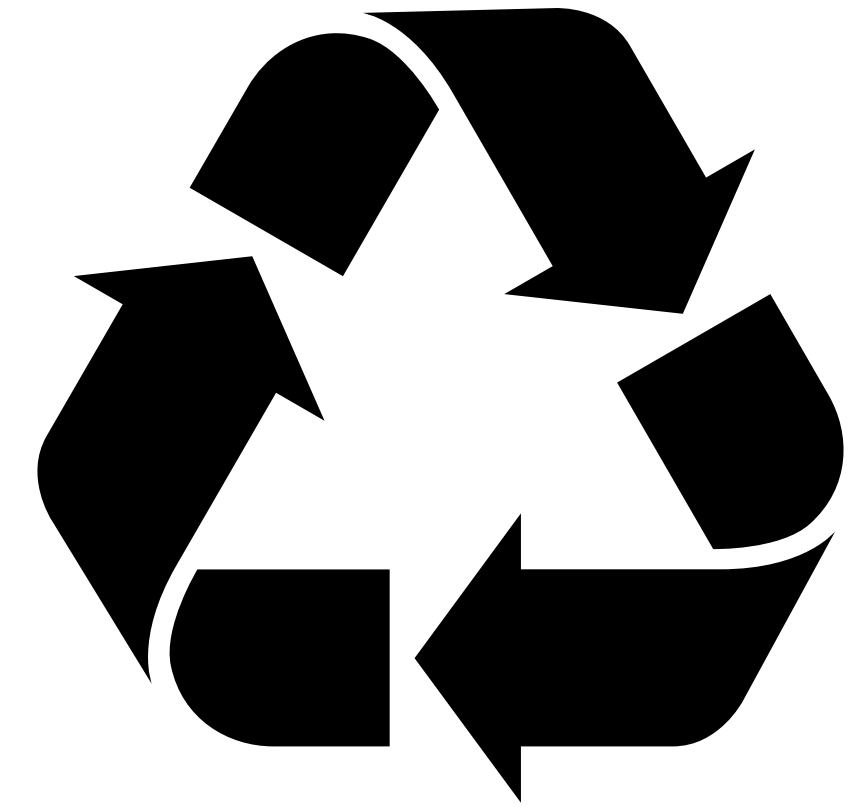
for (i = 0; i < size; i++) {
    for (j = 0; j < size; j++) {
        printf("(%d, %d)\n", vec[i], vec[j]);
    }
}
```

```
// Affiche :
// (1, 1)
// (1, 2)
// (1, 3)
// (2, 1)
// (2, 2)
// (2, 3)
// (3, 1)
// (3, 2)
// (3, 3)
```

Fonctions

Pourquoi?

- Il y a des bouts de code qu'on aimerait réutiliser
 - Lire un tableau
 - Calculer la longueur d'une chaîne de caractères
 - Calculer la racine carrée, sinus, cosinus, ...
 - Utiliser la carte graphique pour afficher des formes 3d
 - Entraîner un réseau de neurones
 - Etc.



Fonctions

Où les trouve-t-on?

- Chaque programme a une **fonction main**
- Il existe des collections de **fonctions** qu'on peut utiliser
 - Des **bibliothèques** (*libraries*) qu'on peut installer!
 - On réutilise le code écrit par d'autres développeurs
- On ~~peut~~ doit écrire nos propres fonctions

Fonctions

- Une fonction mathématique a un domaine et un co-domaine

$$f: \mathbb{R} \times \mathbb{Z} \rightarrow \mathbb{R}$$

- Une **fonction** C aussi!

```
double f(double x, int y); // f: double, int -> double
```

- C'est la déclaration de la fonction

Déclaration d'une fonction

`f` retourne une valeur
de type `double`

`x` et `y` sont les
paramètres de `f`

```
double f(double x, int y);
```

- Utile si la définition est ailleurs
- Devrait apparaître *avant* l'endroit où on appelle la fonction
- Souvent se trouve dans un fichier entête (*header file*) `.h`
 - Exemples: `stdio.h`, `math.h`, `stdlib.h`, etc.

Définition d'une fonction

`f` retourne une valeur
de type `int`

`a` et `b` sont les
paramètres de `somme`

Le corps

```
int somme(int a, int b)
{
    return a + b;
}
```

- Le corps (*body*) = un bloc d'instructions qui utilise les paramètres
- La fonction doit "retourner" une valeur du type indiqué
- Quand elle retourne, elle termine son exécution

Appels de fonctions

Function calls

- On appelle une fonction (*calling a function*)
- Les valeurs transmises à la fonction sont des arguments

```
int main()  
{  
    int s = 0;  
    s = somme(1, 2); // Arguments 1 et 2  
    printf("%d\n", s);  
    // Affiche: 3  
}
```

Appels de fonctions

Function calls

- L'appel de fonction est un **opérateur**
- L'**expression** qui en résulte a le type donné par le type de retour de la fonction
- **Conversion implicite** des arguments vers le type des paramètres

```
int somme(int a, int b);
```

```
...
```

```
int s = 0;
```

```
s = somme(1.3, 2.1); // Arguments convertis en int 1 et 2
```

```
printf("%d\n", s);
```

```
// Affiche: 3
```

Une pile

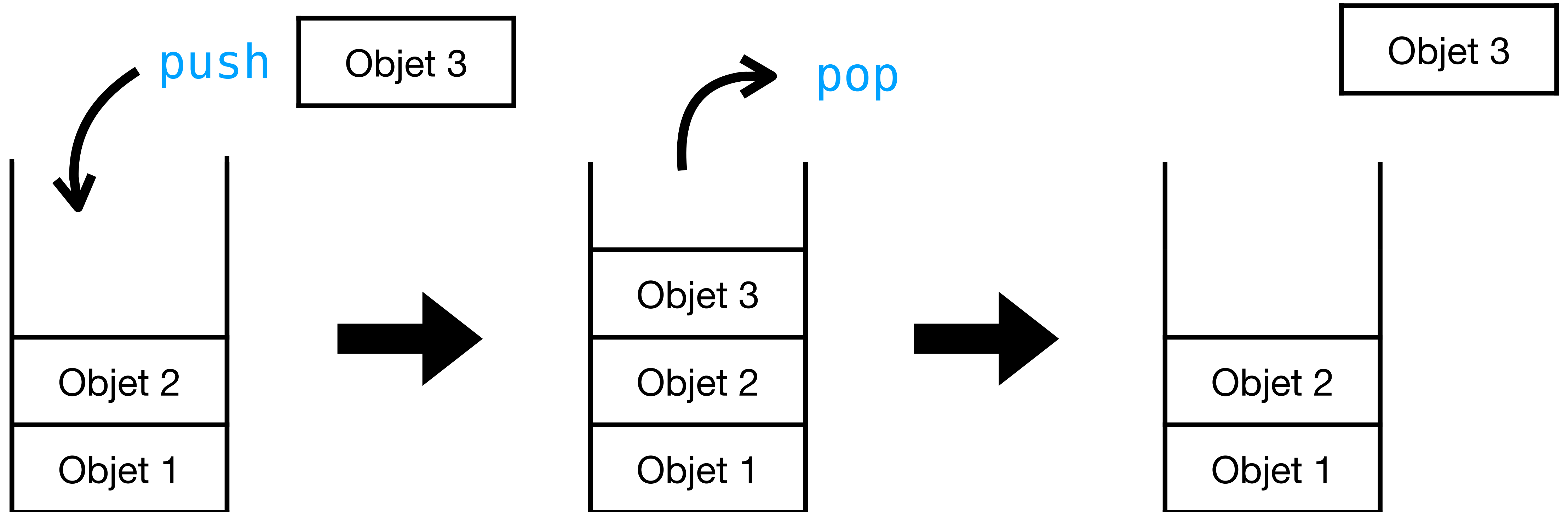
A stack

- [Structure de données](#) qui permet de stocker des objets
- Deux opérations:
 - **push** = on met qqch sur la pile
 - **pop** = on enlève qqch du haut de la pile



Une pile

A stack



La pile d'exécution

Call stack

- [Le contexte](#) des fonctions est stocké dans la pile d'exécution

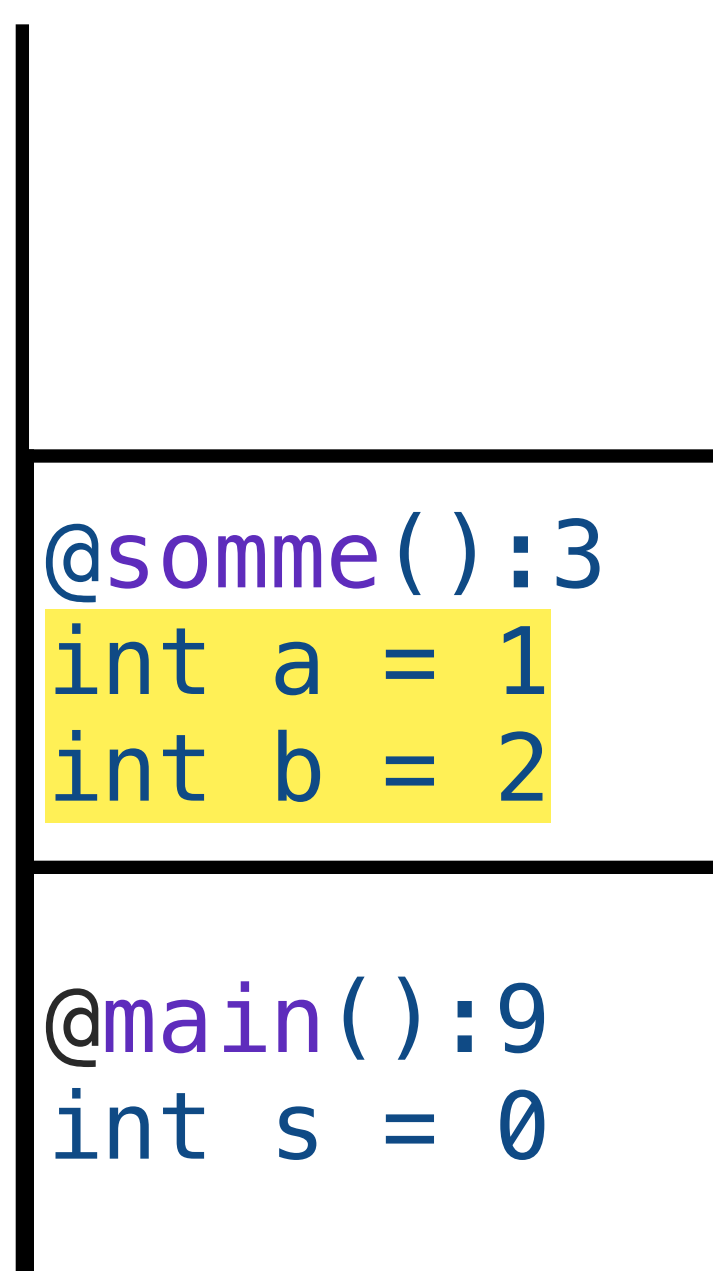
```
@main():9  
int s = 0
```

```
1 int somme(int a, int b)  
2 {  
3     return a + b;  
4 }  
5  
6 int main()  
7 {  
8     int s = 0;  
9     s = somme(1, 2); // Arguments 1 et 2  
10    printf("%d\n", s);  
11    // Affiche: 3  
12 }
```

La pile d'exécution

Call stack

- Quand on appelle une fonction, on **push** son nouveau **contexte** dans la pile
- Les paramètres prennent les valeurs des arguments

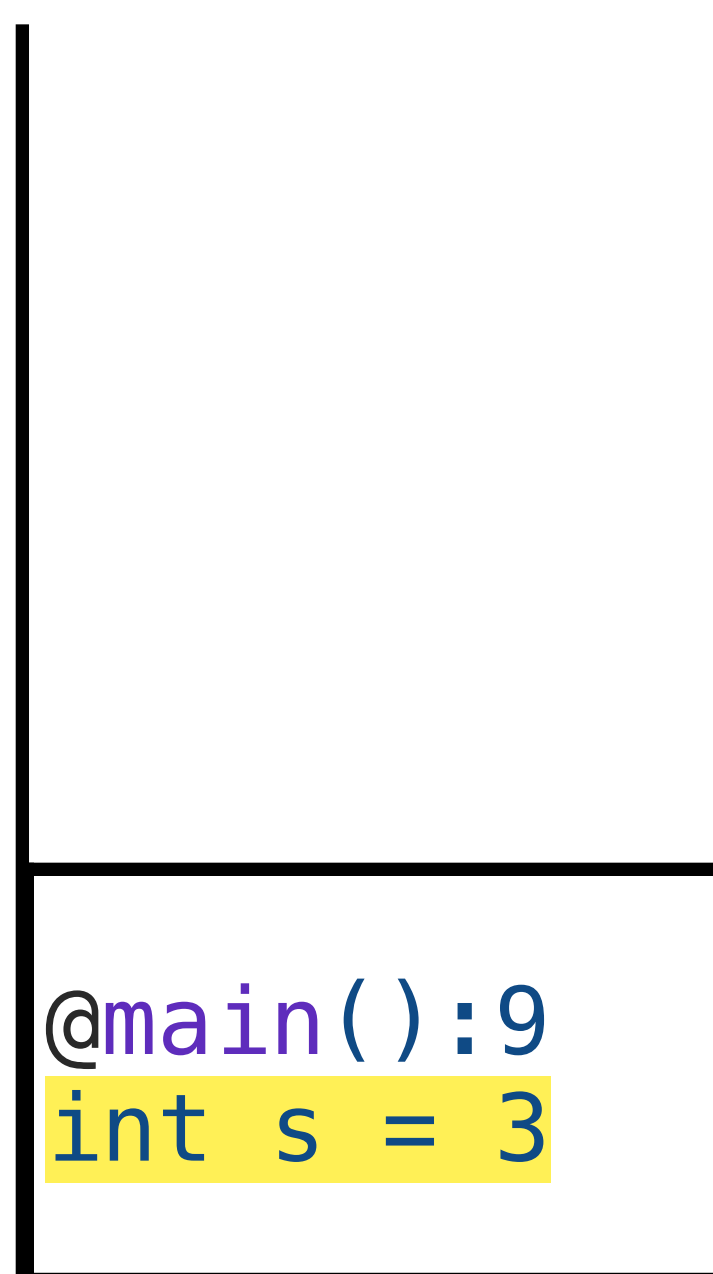


```
1 int somme(int a, int b)
2 {
3     return a + b;
4 }
5
6 int main()
7 {
8     int s = 0;
9     s = somme(1, 2); // Arguments 1 et 2
10    printf("%d\n", s);
11    // Affiche: 3
12 }
```

La pile d'exécution

Call stack

- **return** valeur = on quitte la fonction & **pop** son **contexte** de la pile
- La valeur se retrouve dans l'expression où l'appel a été effectué



```
1 int somme(int a, int b)
2 {
3     return a + b;
4 }
5
6 int main()
7 {
8     int s = 0;
9     s = somme(1, 2); // Arguments 1 et 2
10    printf("%d\n", s);
11    // Affiche: 3
12 }
```


L'instruction `return`

- Termine l'exécution de la fonction
- Le programme continue à partir de l'endroit où la fonction a été appelée
- L'expression de l'appel de fonction prendra la valeur donnée dans le `return`
- Si on utilise `return` dans `main()`, alors le programme est **terminé**

Une fonction qui ne retourne rien

Le type `void`

le type "rien"...

```
void affiche_pair(int test)
{
    if (test % 2)
        return; // on sort
    printf("%d est pair\n", test);
}
```

Pourquoi main a un type de retour?

... et pourquoi on ne retourne rien?

- Quand un programme termine, il “retourne” une valeur au système d’exploitation
- Cette valeur est interprétée comme un “code d’erreur”
- **Convention**: 0 = pas d’erreur
- Si on n’écrit pas de “return” dans main, le compilateur rajoute automatiquement à la fin

```
return 0;
```

```
#include <stdio.h>

int main()
{
    printf("Hello, World!\n");
    printf("Bonjour, ICC!\n");
}
```

Variables locales

Contexte local

- Le [contexte](#) dans la pile d'exécution contient:
 - toutes les variables "actives" dans la fonction
 - la référence de l'instruction qui est en train d'être exécutée
- Les variables définies dans une fonction sont des [variables locales](#)
- Elles sont visibles **uniquement** depuis la fonction où elles sont définies
- On peut réutiliser le nom des variables dans une autre fonction sans impact

Portée des variables locales

- Qu'affiche ce code ?

main avant: a = 1

```
@main()  
int a = 1  
int b = 0
```

```
1 #include <stdio.h>  
2  
3 int incrementer(int a)  
4 {  
5     a = a + 1;  
6     printf("incrémenter: a = %d\n", a);  
7     return a;  
8 }  
9  
10 int main()  
11 {  
12     int a = 1, b = 0;  
13     printf("main avant: a = %d\n", a);  
14     b = incrementer(a);  
15     printf("main après: a = %d\n", a);  
16     printf("main b = %d\n", a);  
17  
18     return 0;  
19 }
```

Portée des variables locales

- Qu'affiche ce code ?

main avant: a = 1
incrémenter: a = 2

Une autre variable a
qui vit dans la fonction
incrémenter

```
@incrémenter  
int a = 2
```

```
@main():14  
int a = 1  
int b = 0
```

```
1 #include <stdio.h>  
2  
3 int incrémenter(int a)  
4 {  
5     a = a + 1;  
6     printf("incrémenter: a = %d\n", a);  
7     return a;  
8 }  
9  
10 int main()  
11 {  
12     int a = 1, b = 0;  
13     printf("main avant: a = %d\n", a);  
14     b = incrémenter(a);  
15     printf("main après: a = %d\n", a);  
16     printf("main b = %d\n", a);  
17  
18     return 0;  
19 }
```

Portée des variables locales


- Qu'affiche ce code ?

```
main avant: a = 1
incrémenter: a = 2
main après: a = 1
main b = 2
```

```
@main()
int a = 1
int b = 2
```

```
1 #include <stdio.h>
2
3 int incrementer(int a)
4 {
5     a = a + 1;
6     printf("incrémenter: a = %d\n", a);
7     return a;
8 }
9
10 int main()
11 {
12     int a = 1, b = 0;
13     printf("main avant: a = %d\n", a);
14     b = incrementer(a);
15     printf("main après: a = %d\n", a);
16     printf("main b = %d\n", b);
17
18     return 0;
19 }
```

Variables globales

- Définies dans le module/fichier .c directement
- Sont visibles partout, sauf quand une variable locale a le même nom
 - Comme des variables aux **portées imbriquées** (*nested scope*)
-  Il est **déconseillé** d'utiliser des variables globales (surtout pour les gros projets)
 - N'importe quelle fonction peut les modifier

```
#include <stdio.h>

int n_appels = 0;

void f()
{
    printf("Appel %d\n",
           n_appels++);
}

int main()
{
    f(); // Appel 0
    f(); // Appel 1
    f(); // Appel 2
    printf("Appel suivant = %d\n",
           n_appels);
    // Appel suivant = 3
}
```


Le passage des arguments

- Dans une fonction, les paramètres sont équivalents à des **variables locales**
- En C il se fait par valeur (*call by value*)
- Lors d'un appel, les paramètres sont initialisés avec les valeurs des arguments
- Autrement dit, **on copie** les arguments dans les paramètres

Paramètres de type tableau

```
#include <stdio.h>

void incrementer(int tableau[], int taille)
{
    for (int i = 0; i < taille; i++)
        tableau[i]++;
}

int main()
{
    int tableau[] = {1, 2, 3, 4, 5};
    int taille = 5;
    incrementer(tableau, taille);
    afficher(tableau, taille);
    return 0;
}
```

```
void afficher(int tableau[],
              int taille)
{
    for (int i = 0; i < taille; i++)
        printf("%d ", tableau[i]);
    printf("\n");
}
```

```
// Affiche 2 3 4 5 6
```

???



Pourquoi?

- Nous verrons en détail comment on représente une variable tableau
- Pour éviter que le tableau soit modifié par la fonction, il faut ajouter **const**

```
void incrementer(const int tableau[], int taille)
{
    for (int i = 0; i < taille; i++)
        tableau[i]++; // Ce n'est plus permis!
}
```

```
fun_test.c:6:15: error: read-only variable is
not assignable
    tableau[i]++;
    ~~~~~^
1 error generated.
```



Slide philosophique

- Les langages de programmation nous donnent beaucoup de liberté
- “Il faut savoir s’en servir” 🙄 mais il n’existe pas de code sans bug 🐛
- Il faut plutôt mettre en place des **mécanismes de protection**
- `const` est un de ces mécanismes
 - le code marche aussi bien *avec* ou *sans*
 - si on sait qu’un paramètre ne doit jamais changer sa valeur, ***il vaut mieux l’indiquer*** au compilateur