

# Algorithmique et programmation structurée en C

Cours n°4



**ECE** PARIS · LYON  
ÉCOLE D'INGÉNIEURS

## Langage C Sous-programmes et multi-fichiers .c

---

Antoine Hintzy

# Plan du cours

1. Sous-programmes : fonctions/procédures
2. Passage de paramètre par valeur et par adresse
3. Multi-fichiers .c

# Programme principal

Le point d'entrée d'un programme écrit en C est la **fonction main ()**, présente dans le fichier **main.c**. On appellera cette fonction **programme principal**.

```
int main (void) { // DÉBUT  
    // ...  
    // code de notre programme ici  
    // ...  
  
    return 0; // FIN  
}
```

# Sous-programmes

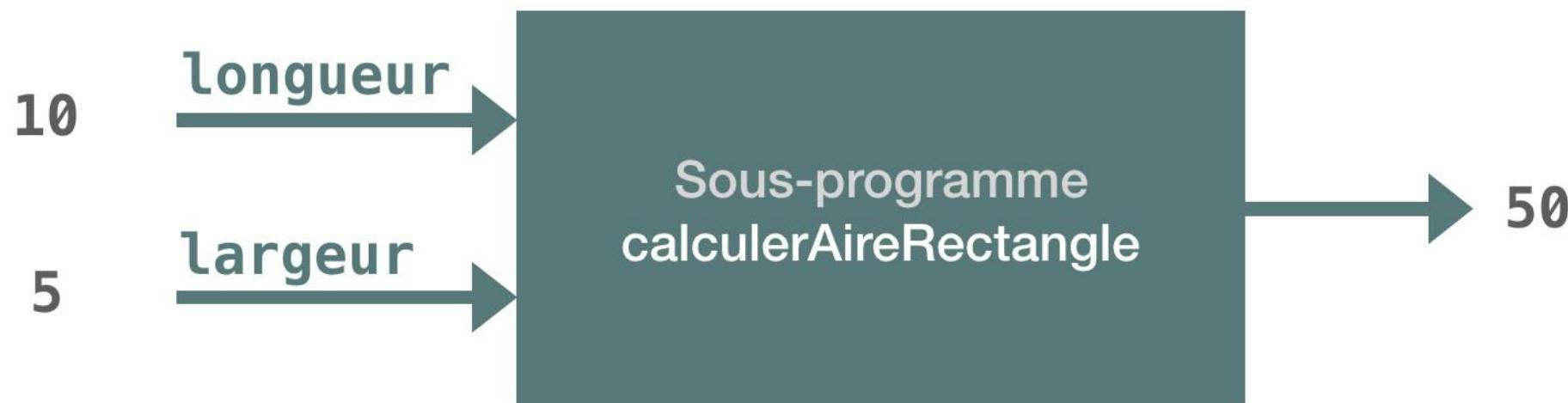
Si nous écrivons tout notre programme dans la **fonction main ()**, le fichier **main.c** deviendra vite :

- très long : beaucoup de lignes de code
- peu maintenable : beaucoup de lignes dupliquées

Nous allons plutôt **décomposer** notre programme **en plusieurs sousprogrammes réutilisables**.

# Qu'est-ce qu'un programme/sous-programme

Un (sous-)programme est comme une **boite noire**, qui regroupe un ensemble d'instructions permettant de **remplir une tâche** (*un sousproblème du cahier des charges identifié lors de l'Analyse Chronologique Descendante*).



# Qu'est-ce qu'un programme/sous-programme

Les (sous-)programmes peuvent prendre zéro, une ou plusieurs valeurs en **entrée** (nécessaire·s dans la réalisation de leur tâche), que l'on appelle **paramètre·s** ou **argument·s**, et peuvent **retourner** ou non un résultat en sortie (*une valeur*).



**NB :** Nous verrons plus tard comment avoir plusieurs sortie.

# Programme vs sous-programme

## ◎ Programme

Le programme répond au cahier des charges. Il est généralement décomposé en plusieurs sous-programmes.

En C, il s'agit de la *fonction main ()* du fichier `main.c`, point d'entrée de notre application.

## ◎ Sous-programme

Les sous-programmes sont des fragments de code, utilisés par un ou plusieurs (sous-)programmes, et répondant à un sous-problème du cahier des charges (ACD).

# Pourquoi des sous-programmes ?

- **Rendre le code plus clair** (mettre une phrase à la place d'un bloc d'instructions).  
Ex : `res = calculateAire(lon, lar)` est plus clair que `res = lon*lar;`.
- **Éviter la duplication de code**, c'est-à-dire les copier/coller de blocs d'instructions avec ajustements.
- **Faciliter l'évolution** : Supposons qu'un jour, l'aire d'un rectangle se calcule différemment, il n'y aurait qu'un seul endroit à modifier avec un sous-programme.
- Pouvoir décomposer un programme long et complexe en plusieurs **sous-programmes simples et réutilisables**.
- Pouvoir **mieux structurer** son code source en le découplant en **plusieurs fichiers**.

# Exemple - Utilité des sous-programmes

## Sans sous-programme

```
int main() {  
    int largeurRectangle1 = 0, longueurRectangle1 = 0,  
        largeurRectangle2 = 0, longueurRectangle2 = 0,  
        perimetreRectangle1 = 0, perimetreRectangle2 = 0;  
    // ...instructions demandant largeur/longueur des 2 rectangles...  
  
    perimetreRectangle1 = (largeurRectangle1 + longueurRectangle1) * 2;  
    perimetreRectangle2 = (largeurRectangle2 + longueurRectangle2) * 2  
    return 0;  
}
```

Ici, le calcul du périmètre est dupliqué pour les deux rectangles, ce qui n'est pas réellement dérangeant car c'est un calcul que tout le monde connaît, et il ne devrait jamais changer...

Mais c'est ce qu'on appelle de la **duplication de code** et les bons développeurs n'en font pas.  
Transformons ce calcul en sous-programme...

# Exemple - Utilité des sous-programmes

## Avec sous-programme

```
int calculerPerimetreRectangle(int lon, int lar) {
    return (largeurRectangle1 + longueurRectangle1) * 2;
}

int main() {
    int largeurRectangle1 = 0, longueurRectangle1 = 0,
        largeurRectangle2 = 0, longueurRectangle2 = 0,
        perimetreRectangle1 = 0, perimetreRectangle2 = 0;
    // ...instructions demandant largeur/longueur des 2 rectangles...

    perimetreRectangle1 = calculerPerimetreRectangle(longueurRectangle1, largeurRectangle1);
    perimetreRectangle2 = calculerPerimetreRectangle(longueurRectangle2, largeurRectangle2);
    return 0;
}
```

Maintenant, le calcul du périmètre d'un rectangle n'est écrit qu'une seule fois dans notre code source. S'il venait à changer, nous n'aurions pas à essayer de l'identifier dans 3000 lignes de code. De plus, le programme principal (main) ne contient plus que des phrases faciles à lire.

# Vocabulaire

- On dit qu'on **appelle** un (sous-)programme quand on demande son exécution :

```
printf("Bonjour !"); // appel de la fonction printf avec un paramètre : la chaîne de caractères "Bonjour !"
```

- On appelle **paramètre** ou **argument** une valeur transmise à un (sous-)programme, au moment de son **appel** (entre les parenthèses).

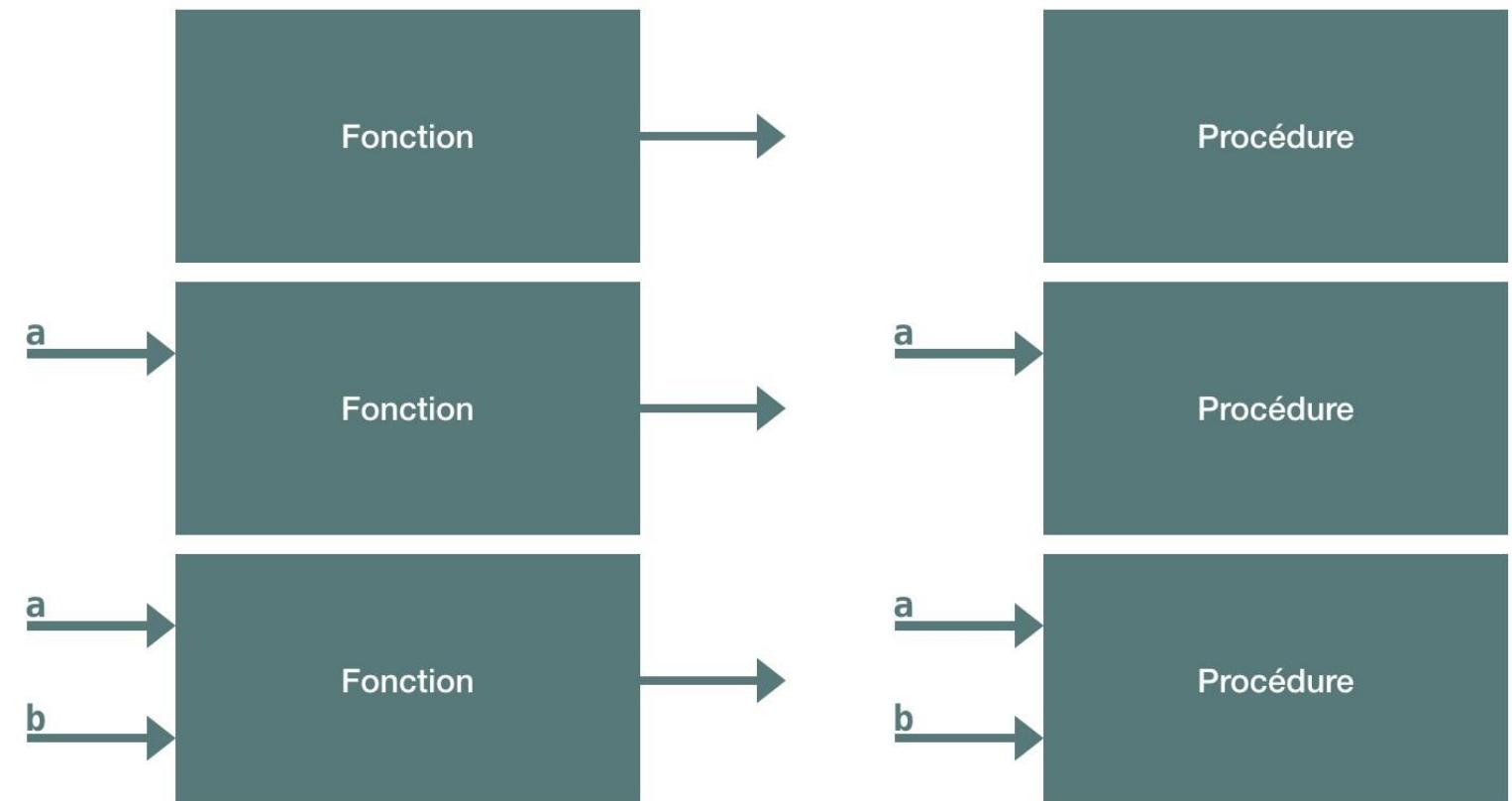
- Un (sous-)programme peut en appeler un autre, on parle alors de **programme appelant/appelé**.

## Fonctions vs procédures

Les (sous-)programmes sont des **fonctions** ou des **procédures**.

Une **procédure** est une **fonction qui ne retourne aucun résultat** (fonction sans sortie).

Le terme fonction peut être utilisé pour parler d'une procédure (une procédure est une fonction particulière qui ne retourne aucun résultat).



**NB :** Ce n'est pas parce qu'une procédure ne retourne rien qu'elle ne peut pas modifier les données qu'elle reçoit, ni transmettre des informations à son (sous-)programme appelant. Nous verrons comment faire cela en manipulant directement des espaces mémoire.

# Nous utilisons déjà des fonctions/procédures...

- **main (...)** : programme principal.
- **printf (...)** : fonction de la bibliothèque standard **stdio.h** permettant d'écrire dans la console.
  - prend en **paramètres** les données à afficher
  - les affiche
  - retourne le nombre de caractères affichés, ou une valeur négative en cas d'erreur.

- **scanf (...)** : fonction de la bibliothèque standard **stdio.h** permettant de lire au clavier.
  - prend en **paramètres** le format des données à lire au clavier et où les stocker - retourne le nombre de caractères lus, ou une valeur négative en cas d'erreur.

# Création d'une fonction/procédure - Prototype

Pour créer une fonction ou une procédure, on commence par **déclarer** son **prototype**. Le prototype indique comment fonctionne la fonction ou la procédure : déclaration du **nom**, et des éventuelles **entrées/sortie** :

```
typeDeRetour nomDuSousProgramme(typeParam1 nomParam1, ..., typeParamN nomParamN)
```

- **typeDeRetour** : type de ce que va retourner la fonction. **void** pour les procédures.
- **nomDuSousProgramme** : nom du (sous-)programme, le plus **explicite** et lisible possible.
- Éventuels paramètres entre parenthèses (séparés par une virgule) :
  - **typeParam1** : type du paramètre **nomParam1**.
  - **nomParam1** : nom du premier paramètre (n'existera que dans la fonction).
- S'il n'y a aucun paramètre, soit on écrit **void** entre les parenthèses, soit on les laisse vides.

# Création d'une fonction/procédure - Prototype

Pour pouvoir appeler une fonction/procédure, il faut que son prototype ait été déclaré/soit connu.

Il existe deux façons de déclarer un prototype :

- **Déclaration simple** (avant tout code appelant cette fonction) : on ajoute un point-virgule au prototype et on pourra définir le corps de la fonction/procédure plus loin (même après du code appelant !) ou dans un autre fichier :

```
typeDeRetour nomDeLaFonction(typeParam1 nomParam1, ..., typeParamN nomParamN); // Déclaration de la fonction
```

# Création d'une fonction/procédure - Prototype

- **Déclaration + définition directe** : on **définit** directement le corps de la fonction/procédure (son code) au moment de sa déclaration, entre accolades (avant tout code appelant cette fonction) :

```
typeDeRetour nomDeLaFonction(typeParam1 nomParam1, ..., typeParamN nomParamN) { // Déclaration de la fonction
    // Définition de la fonction :
    // Corps de la fonction (bloc d'instructions), suivi par d'un return (sauf pour les procédures) :
    return uneValeurDuMemeTypeQueTypeDeRetour; // valeur transmise au (sous-)programme appelant
}
// ! le prototype doit être déclaré AVANT tout code appelant
int nombreAuCarre(int nb);

int main(void) {

    // on peut appeler nombreAuCarre() car son prototype est déclaré avant (il est connu) :
    int cinqAuCarre = nombreAuCarre(5);
    return 0;
}
```

# Création d'une fonction/procédure - Prototype

```
int nombreAuCarre(int nb) { // définition du corps de la fonction initialement déclarée
    return nb * nb;
}
```

**NB :** il est important de déclarer le prototype de la fonction **nombreAuCarre** **avant** notre fonction **main** afin que cette dernière (qui l'appelle) la connaisse. La séparation de la déclaration du prototype et de la définition de la fonction nous permettra de découper notre code source en plusieurs fichiers, on aurait pu définir la fonction (= écrire son code source) en même temps que sa déclaration (en haut du **main**) -> slide suivante.

```
// déclaration et définition de la fonction
int nombreAuCarre(int nb) {
    return nb * nb;
}

int main(void) {
    int cinqAuCarre = nombreAuCarre(5); // on peut appeler la fonction car elle a été déclarée avant
    return 0;
}
```

# Instruction `return`

Dans une fonction, l'instruction `return` précise la valeur sortante, c'est-à-dire la valeur transmise au (sous-)programme appelant.

- L'instruction `return` met automatiquement fin au (sous-)programme en cours d'exécution.  
Tout ce qui se trouve après dans cette fonction ne sera donc pas exécuté.
- Le type de la valeur renvoyée doit être identique à celui spécifié dans le prototype.

```
int maFonction(int a) {  
    return a * 5; // FIN du sous-programme maFonction  
    printf("Ce printf ne sera jamais exécuté..."); }
```

**NB :** Ce n'est pas parce qu'une fonction retourne un résultat que l'on doit forcément l'utiliser. Par exemple, nous n'utilisons jamais la valeur renvoyée par `printf`.

## Instruction `return` - fonction `main()`

Vous avez dû remarquer, le programme principal (`main`) retourne toujours 0.

La fonction `main` est un peu particulière car ce n'est pas nous qui l'appelons, ça peut être un terminal, le système d'exploitation...

Par convention, le programme (`main`) retourne 0 lorsqu'il se termine avec succès, ou un autre chiffre lorsqu'une erreur se produit. Cela aide au déboggage.

**NB :** un `return` dans le main met automatiquement fin à cette fonction et donc au programme. Si on veut mettre fin au programme depuis un sous-programme, on peut utiliser la procédure `void exit(int status) ;` (0 si succès, 1 si échec).

# Création d'une procédure - Exemple

`void` en type de retour indique que la fonction ne retourne rien, il s'agit donc d'une procédure. Il n'y a donc pas de `return`.

```
void direAge (int age) { // procédure avec un paramètre, qui ne retourne rien
    printf("Vous avez %d ans.\n", age);
}

int main (void) {
    direAge(21); // nous n'avons aucun résultat à récupérer,
                  // int age = direAge(21); n'aurait donc aucun sens
    return 0; }
```

**NB :** Il est possible d'ajouter une instruction `return;` (sans valeur) afin de mettre fin à l'exécution d'une procédure, on revient alors directement au (sous-)programme appelant.

# Portée des variables

Les paramètres d'une fonction ou d'une procédure, tout comme les variables qui sont déclarées dans son **bloc** (= entre ses {accolades}), n'existent et ne sont accessibles que dans son bloc.

On dit qu'elles ont une **portée locale** à la fonction/procédure dans laquelle elles sont déclarées. C'est-à-dire qu'on ne peut pas y accéder depuis l'extérieur :

```
int maFonction(int a) {
    // les variables a et b n'existent que dans ce bloc (entre les accolades)
    int b = 0;
    printf("Saisissez un entier :\n");
    scanf("%d", &b);
    return b * a;
}

int main() {
    // le main n'a accès ni à la variable a, ni à la variable b de maFonction
    printf("%d\n", maFonction(4)); // 24
    return 0;
}
```

# Portée des variables

On peut avoir plusieurs variables qui portent le même nom à condition qu'elles ne soient pas déclarées dans le même bloc.

```
int maFonction(int a) {
    // les variables a et b n'existent que dans ce bloc (entre les accolades)
    int b = 0;
    printf("Saisissez un entier :\n");
    scanf("%d", &b);
    return b * a;
}

int main() {
    int a = 4;
    // le main n'a accès ni à la variable a de maFonction, ni à sa variable b
    // mais il a accès à sa propre variable a
    printf("%d\n", maFonction(a)); // 24
    return 0;
}
```

Pourquoi ? Car à chaque déclaration, on réserve un nouvel espace mémoire.

Donc la variable **a** du main a une adresse mémoire différente de la variable **a** de la fonction **maFonction**.

# Appel de fonction/procédure

Soit la fonction suivante :

```
int calculerAire(int longueur, int largeur) {  
    return longueur * largeur;  
}
```

Pour l'utiliser, il faut l'**appeler** en écrivant son nom et en remplaçant chaque paramètre par une valeur.

Nous allons voir dans les slides suivantes différentes façons de l'appeler.

# Appel de fonction/procédure - Exemple d'appel 3

```
int calculerAire(int longueur, int largeur) { X
    return longueur * largeur;
}

int main() {
int aire = 0;
    calculerAire(3, 2); // X on ne récupère pas le résultat (autant ne rien faire)
    aire = calculerAire(3, 2); // ✓
    return 0;
}
```

Dans cette version, nous donnons directement des valeurs (3 et 2) au moment de l'appel. Ces valeurs sont copiées respectivement dans les paramètres **longueur** et **largeur** de la fonction.

# Appel de fonction/procédure - Exemple d'appel 3

```
int calculerAire(int longueur, int largeur) {  
    return longueur * largeur;  
}  
  
int main() {  
    int longueur = 5, aire = 0;  
    aire = calculerAire(longueur, 2);  
    return 0;  
}
```

Dans cette version, nous faisons un mélange entre donner la valeur contenue dans une variable (⚠ 5) et donner directement une valeur (2). La valeur de la variable **longueur** (5) du **main** sera copiée dans la variable **longueur** de la fonction **calculerAire** (⚠ ce sont deux variables différentes bien qu'elles aient le même nom) et la valeur 2 sera copiée dans la variable **largeur** de la fonction.

**NB:** Les variables pourraient tout à fait s'appeler différemment.

# Appel de fonction/procédure - Exemple d'appel 3

```
int calculerAire(int longueur, int largeur) {  
    return longueur * largeur;  
}  
  
int main() {  
    int aire = 0;  
    aire = calculerAire(getEntier(), getEntier());  
    return 0;  
}
```

Cette version est beaucoup plus difficile à comprendre : on appelle une fonction à la place d'un paramètre.

Nous supposerons que le prototype de cette fonction est le suivant : `int getEntier(void)`, et que cette fonction demande un entier au clavier à l'utilisateur.rice et le retourne à la fonction appelante.

Ceci est possible car `getEntier` retourne un `int` et la fonction `calculerAire` veut des `int` en paramètres.

Dans ce cas, l'ordinateur exécute d'abord les deux appels de `getEntier()`, puis injecte respectivement leur résultat dans les paramètres de la fonction `calculerAire`.

# Passage par valeur

Lorsqu'on appelle une fonction, le fait de copier une valeur depuis le (sous-)programme appelant vers le sous-programme appelé s'appelle **passage par valeur**. Tous les exemples précédents sont des passages par valeur.

Nous verrons plus tard le passage par adresse, qui consiste à copier non pas une valeur mais une adresse dans la mémoire centrale.

Les sous-programmes appelés travaillent donc sur des **copies** locales des valeurs qu'elles reçoivent en paramètres. Ainsi, toute modification faite sur un paramètre dans la fonction appelée n'affecte pas la valeur copiée dans le (sous-)programme appelant.

Le passage par adresse nous permettra de modifier les variables du (sous-)programme appelant.

```
void direAgeSuivant (int age) { // on récupère la valeur 21 que l'on copie dans la variable locale age
    age += 1; // on ajoute 1 à la copie locale
    printf("Vous avez %d ans.\n", age); // Affiche "Vous avez 22 ans."
}

int main (void) {
    int age = 21;
    direAgeSuivant(age); // on donne la valeur de la variable age (21) à copier
    printf("Vous avez %d ans.\n", age); // Affiche "Vous avez 21 ans."
    return 0;
}
```

# Erreur fréquente Identification des entrées

```
✗ void direBonjourXFoisMauvais(int x, int i) { // i n'est pas une donnée utile à cette procédure
    for(i = 0; i < x; i++) {
        printf("Bonjour !\n");
    }
}

✓ void direBonjourXFoisBon(int x) {
    int i = 0;                                // si ce n'est pas une donnée utile en entrée, ce n'est pas un paramètre
    for(i = 0; i < x; i++) {
        printf("Bonjour !\n");
    }
}

int main() {
    direBonjourXFoisMauvais(5, 0); // ✗ on pourrait mettre 3, 6... à la place du 0,
                                    // ça ne changerait rien au fonctionnement de la procédure,
                                    // ça n'a donc rien à faire là !
    direBonjourXFoisBon(5);      // ✓ on envoie la valeur 5, une donnée utile à la procédure
    return 0;
}
```

# Passage par adresse

- Imaginons que l'on crée une version simplifiée de **scanf** sans utiliser d'adresse.

```
void scanfMaison(/* ... *, int variableAModifier) { // On reçoit une copie de la
                                                        // valeur 0 dans variableAModifier
    // code compliqué
    variableAModifier = 5; // On suppose que l'utilisateur a tapé 5, la copie est modifiée
}
int main(void) {

    int a = 0;                // a vaut 0
    scanfMaison("%d", a);
    printf("%d", a);          // a vaut toujours 0 (c'est une copie qui a pris la valeur 5), affiche 0
    return 0;
}
```

- Dans le main, pourquoi a ne vaut pas 5 après l'appel de **scanfMaison** ? Car a est passé par valeur.

On copie sa valeur dans la variable de la fonction appelée (**scanfMaison**).

- ◎ Le sous-programme **scanfMaison** travaille donc sur une copie de la valeur de la variable **a** du **main** (une copie de la valeur 0, qui s'appelle **variableAModifier** à l'intérieur de **scanfMaison**).

Pour que la procédure **scanfMaison** puisse modifier directement la valeur de la variable **a** du **main**, ce dernier ne va non plus lui transmettre une copie de la valeur de sa variable **a** (0) mais une **copie de son adresse en mémoire** (ex : **0x3ba1**). Ainsi, la procédure **scanfMaison** pourra modifier directement l'espace mémoire de la variable **a** du **main** :

Sans ce mécanisme, une fonction ne peut modifier que ses propres variables (celles déclarées dans son **bloc** (*ou de façon globale (pas bien)*)).

## Pourquoi utiliser une adresse en passage de paramètre ?

Et cette étoile (\*) à côté du **int**, c'est quoi ? C'est un **pointeur**.

```
void scanfMaison(/* ... */, int* adrVarAModifier) { // adrVarAModifier reçoit comme valeur l'adresse 0x3ba1
                                                        // code compliqué
    *variableAModifier = 5; // On suppose que l'utilisateur a tapé 5,
                           // on écrit la valeur 5 à l'adresse contenue dans adrVarAModifier (en 0x3ba1)
                           // (l'adresse de la variable a du main)
}
int main(void) {
    int a = 0;           // a vaut 0, on suppose qu'elle est à l'adresse 0x3ba1
    scanfMaison("%d", &a); // Nous passons 0x3ba1 en paramètre
    printf("%d", a);    // a vaut maintenant 5 le printf affiche 5
    return 0;
}
```

Le **main** lui fournit bien l'adresse de sa variable **a** (et non pas sa valeur), afin que le **scanf** puisse modifier directement sa valeur.

# Découpe en plusieurs fichiers

# Librairies / Bibliothèques

Où sont écrit les fonctions **printf** et **scanf** ?

- Elles appartiennent à la **bibliothèque** ( *library*) **standard** : **stdio.h**.
- Les librairies fournissent des fonctions et procédures prêtes à être utilisées.

# Librairies standards

Les bibliothèques standards sont connues des compilateurs et sont directement utilisables. Il suffit de **les inclure dans nos fichiers .c** lorsqu'on en a besoin (tout en haut du fichier) :

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

Les bibliothèques standard s'incluent entre chevrons **#include <std...>**.

Il est possible de créer nos propres fichiers/bibliothèques( *libraries*), puis de les inclure dans un autre fichier en utilisant des guillemets :

```
#include "mesfonctions.h" // chemin vers le fichier entre guillemets
```

# Fichiers Headers

Avec **#include**, on inclue toujours les **fichiers d'en-tête** (🇬🇧 headers).

Ils se terminent par l'extension **.h** et contiennent — entre-autres — les **déclarations des prototypes de fonctions et procédures**, dont le corps est défini dans un fichier **.c** du même nom.

On n'écrit donc jamais **✗ #include "maBiblio.c"**

On écrit : **✓ #include "maBiblio.h"**

## - mesfonctions.h

```
#ifndef ECE_COURS_SEANCE3_MESFONCTIONS_H
#define ECE_COURS_SEANCE3_MESFONCTIONS_H

void direAge (int age); void
direAgeSuivant (int age); int
nombreAuCarre(int nb);

#endif
```

**NB :** des instructions de préprocesseurs **#ifndef/#endif** évitent les inclusions infinies (ex : un fichier A inclue un fichier B et inversement), à l'aide d'un identifiant unique : ici **ECE\_COURS\_SEANCE3\_MESFONCTIONS\_H**.

## - mesfonctions.c

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

void direAge (int age) {
    printf("Vous avez %d ans.\n", age);
}

void direAgeSuivant (int age) {
    direAge(age+1); // réutilisation astucieuse d'une autre procédure
}

int nombreAuCarre(int nb) {
return nb * nb;
}
```

## - main.c

```
#include "mesfonctions.h"

int main(void) {
    direAgeSuivant(21);
    return 0;
}
```