Programmation C et Structures de données (Prog 2)

Cours 0

Présentation et Rappels

Pierre Fouilhoux & Christophe Tollu pierre.fouilhoux@lipn.fr et ct@lipn.univ-paris13.fr

30 janvier 2024

- Présentation
- Rappel : Types de données
 - Variables
 - Variables scalaires
 - Tableaux
 - Structures
 - Affectation de variables
- Rappel : Fonctions et paramètres
 - Déclarations, définitions et appels
 - Paramètres

Organisation

12 semaines

Attention : les horaires changent suivant les groupes et les semaines ! Regardez les ajustements sur votre emploi du temps !

1 -	Mardi 30/01	СМ	TD	TP
2 -	Mardi 06/02	CM	TD	TP
3 -	Mardi 13/02	CM	TD	TP
4 -	Mardi 27/02	CM	TD	TP
5 -	Mardi 05/03	CM	TD	TP*2
6 -	Mardi 12/03	CM	TD*2	TP
7 -	Mardi 19/03	CM*2	TD	TP
8 -	Mardi 02/04	CM	TD	TP
9 -	Mardi 23/04	CM	TD*2	TP*2
	Jeudi 25/04			TP*2
10 -	Mardi 30/04	CM	TD	TP
	Jeudi 02/05	CM	TD	
11 -	Mardi 07/05	CM	TD	TP
12 -	Mardi 14/05	CM	TD	TP*2
	Jeudi 02/05		TD	TP

Contrôle des connaissances

Déroulé des périodes :

- 3 semaines CM/TD/TP
- 1 semaine de vacances
- 4 semaines CM/TD/TP dont certaines en doubles séances
- Partiel 1 : semaine du 25/03
- 1 semaine CM/TD/TP
- 1 semaine de vacances + 1 semaine sans Prog2
- 4 semaines avec Mardi et Jeudi
- Partiel 2 : semaine du 20/05
- Rattrapage : semaine du 24/06

3 types d'interrogations:

- Des contrôles (QCM, TP) en TD/TP
- Un rendu de mini-projet final
- 2 partiels Pa1 et Pa2

Contrôle des connaissances

Trois notes au final

- EvC : contrôle continu (TP notés + QCM + mini-projet)
- Pa1 : partiel de mi-semestre : semaine du 25/03/24
- Pa2 : partiel de fin de semestre : semaine du 20/05/24

Note finale

Max {(EvC + Pa1 + 2Pa2)/4; Pa}

Avertissement

- Absence à un partiel, à une épreuve de contrôle continu ou TP noté remis hors délai = note 0.
- Absence à toutes les épreuves = défaillance.

Mini-projet

Mini-projet en fin de semestre

- A faire en binôme (pas en trinôme!)
- A rendre avant la dernière séance de TP
- ... Passage potentiel de votre code et votre rapport au détecteur de plagiat...
- A la dernière séance de TP : mini-soutenance avec questions individuelles
 - Questions sur le code : "à quoi sert cette boucle ?".
 - Questions théoriques sur le rapport : "quelle est la complexité de cette fonction? et pourquoi?
 - Questions expérimentales sur les performances de votre code (courbes d'expérimentation, analyse...)
 - Et des questions surprises qui demandent du recul sur le travail rendu!

On détectera facilement si vous n'avez pas fait vous-même le travail rendu...

Contrôle des connaissances (2nde chance)

Une seule note

• Pa : partiel de seconde chance : semaine du 26 juin 2023.

Note finale

Pa

Pour éviter d'en arriver là

- Apprendre le cours au fur et à mesure, participer activement en TD et TP, finir les exercices des feuilles de TD et TP.
- On apprend à programmer en programmant, pas simplement en feuilletant son cours ou les solutions.
- On apprend à faire des exos écrits en refaisant les exos de TD, pas en lisant les corrections.

Programmation 2 sur Moodle USPN

URL

Espace dédié au cours, "Programmation 2 (N2 MATH-INFO)" à l'adresse : https://moodlelms.univ-paris13.fr/course/view.php?id=547

Ressources

Y seront disponibles

- les énoncés des TD et TP;
- les diapos du cours;
- des informations importantes;
- des recommandations (par exemple pour les partiels);
- des zones de dépôt pour les TP notés;
- peut-être quelques corrigés (TP et partiels).
- Vous devez consulter le cours sur Moodle plusieurs fois par semaine
- Vous devez utiliser le forum "Forum Prog. 2" pour poser vos questions sur le contenu du cours et son organisation

Programmation 2 : la suite de Programmation 1

Objectifs du cours

- Progresser dans la programmation :
 - Faire des programmes longs, mini-projets
 - Acquérir de bonnes pratiques de programmation
 - Compilation séparée et Makefile
 - Débogage
- Première notion de gestion de la mémoire et les structures de données
 - Pointeurs
 - Allocation dynamique de mémoire (et gestion des fuites mémoire)
 - Listes (simplement et doublement) chaînées
- Être capable d'implémenter et d'analyser
 - des programmes par l'introduction de la complexité des algorithmes
 - des algorithmes "classiques" de recherche et de tri.
 - des algorithmes "classiques" d'analyse de texte
 - des algorithmes provenant de l'algèbre linéaire

Bibliographie sommaire

La référence

- Brian W. Kernighan and Dennis M. Ritchie, The C programming language Prentice Hall, 1978 (2nd ed. 1988)
- Traduction française : Le langage C. Norme ANSI. 2e édition, Dunod, 2000

Autres ouvrages recommandés

- Claude Delannoy, Le livre du C premier langage, Eyrolles, 2002
- Claude Delannoy, *Programmer en langage C* (5e édition), Eyrolles, 2016

- Présentation
- Rappel : Types de données
 - Variables
 - Variables scalaires
 - Tableaux
 - Structures
 - Affectation de variables
- Rappel : Fonctions et paramètres

Variable : définition

Définition

Une variable est une zone de mémoire caractérisée par :

un nom (ou identificateur)
un type (qui détermine la taille de l'espace mémoire occupé par la variable)
une valeur (codée en mémoire selon le type de la variable)
une adresse

Example

```
int x=4;
```

- Nom:x
- Type : entier positif ou négatif (int)
- Valeur: 4
- Une adresse : 990 (déterminée à l'exécution du programme pour une variable locale)

Variable : définition

Définition

Une variable est une zone de mémoire caractérisée par :

un nom (ou identificateur)
un type (qui détermine la taille de l'espace mémoire occupé par la variable)
une valeur (codée en mémoire selon le type de la variable)
une adresse

Example

```
int x=4;
```

- Nom : x
- Type : entier positif ou négatif (int)
- Valeur: 4
- Une adresse : 990 (déterminée à l'exécution du programme pour une variable locale)

Variable : déclaration et initialisation

Déclaration

La déclaration permet au programme de connaître le nom et le type d'une variable :

int x;

Initialisation

L'initialisation permet d'affecter une valeur à une variable :

$$x = -17;$$

Adressage

L'opérateur unaire & permet de référencer une variable par son adresse :

```
scanf("%d", &x);
```

Locale/ Globale

Variable locale (à une fonction, à un bloc)

- Déclaration : dans une fonction (et au début d'un bloc { · · · } !)
- Portée et accessibilité : la partie du bloc suivant sa déclaration.
- Par convention, le nom d'une variable locale commence par une minuscule.

```
int main(){
/* Declaration de deux variables locales */
int x = 4;
int y = 2*x;
printf("x = %d, y = %d\n", x, y);
return EXIT_SUCCESS;

// }
```

En mémoire

Une variable locale est stockée dans **la pile d'appel** de la fonction et ne peut être reconnue comme variable que dans le bloc de la fonction.

Rappel: la fonction main est une fonction comme les autres!

Locale/ Globale

Variable globale

- Déclaration : en dehors de toute fonction
- Portée (compilateur) : la partie du fichier source suivant la (première) déclaration.
- Accessibilité : par défaut étendue aux autres fichiers sources.
- Par convention, le nom d'une variable globale commence par une majuscule.

```
1 int X = 4; /* Variable globale */
2 int main() {
3   int y = 2*X, x = 2*y;
4   printf("X = %d, x = %d, y = %d\n", X, x, y);
5   return EXIT_SUCCESS;
6 }
```

En mémoire

Une variable globale est stockée dans la **zone de données** : elle est reconnue et accessible dans tous les blocs du programme.

- Présentation
- Rappel : Types de données
 - Variables
 - Variables scalaires
 - Tableaux
 - Structures
 - Affectation de variables
- Rappel : Fonctions et paramètres

Variable scalaire

Variable scalaire

Variable destinée à contenir une valeur atomique, une unique "donnée".

Exemple

- un nombre
 - entier: int, unsigned int, long int, long long int
 - "réel" (en vérité toujours un rationnel) : float, double
- un caractère (affichable ou non) : char

Variable scalaire locale

```
int main() {
    int x=4;
    float y=2.5;
    printf("%d\n",x);
    return EXIT_SUCCESS;
}
```

Exécution du programme : pile d'appel de la fonction main



Conventions

- Dans la zone grisée sont stockées des informations indispensables pour l'exécution de l'appel (adresse de retour, etc.)
- A droite, on indique l'adresse mémoire de la case (on a mis ici 1000 sans réalisme aucun)
- Les adresses sont décroissantes en montant dans la pile

- Présentation
- Rappel : Types de données
 - Variables
 - Variables scalaires
 - Tableaux
 - Structures
 - Affectation de variables
- Rappel : Fonctions et paramètres

Tableaux

Tableau

Séquence de variables d'un même type. Un tableau permet de

- déclarer plusieurs variables en une seule instruction,
- parcourir plus facilement un ensemble de valeurs (ex : boucle)
- ...

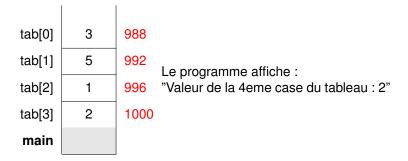
Exemple

```
int tab[4]=\{1,2,3,4\};
```

- Nom: tab
- Type: Tableau d'entiers: chaque case est de type int.
- Valeur de chaque case:int tab[4];
 tab[0]= 1; tab[1]= 2; tab[2]= 3; tab[3]= 4;
- Adresse: l'adresse contenue dans tab est la même que celle de tab [0].

Tableaux

```
int main(){
int tab[4]={3,5,1,2};
printf("Valeur de la 4eme case du tableau : %d\n",tab[3]);
return EXIT_SUCCESS;
}
```



Tableaux à plusieurs dimensions

Tableaux à deux dimensions

Il est possible de créer des tableaux à deux dimensions (ou plus).

 Chaque case d'un tableau à deux dimensions est un tableau à une dimension.

Exemple

int tab[3][2]={
$$\{3,5\}$$
, $\{1,2\}$, $\{4,1\}$ };



Tableaux à plusieurs dimensions

```
int main(){
   int tab[2][2]={ {3,5} , {1,2} };
   printf("Valeur de la case ligne 0 et colonne 1 : %d\n",tab[0][1]);
   return EXIT_SUCCESS;
}
```



Tableaux particuliers

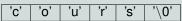
Les chaînes de caractères

Les chaînes de caractères sont des tableaux à une dimension dont chaque case est de type char.

- permet de stocker des mots, des phrases, etc.
- la fin d'une chaîne est obligatoirement signalée par un '\0'.
- si on souhaite stocker un mot de longueur n il faut donc n+1 cases dans le tableau.

Exemple

Pour stocker le mot "cours", on définit le tableau suivant :



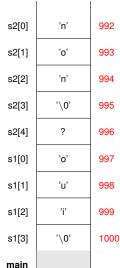
Chaînes de caractères

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

#int main() {
char s1[4]="oui";
char s2[5]= {'n', 'o', 'n', '\0'};
printf("%s et %s\n", s1, s2);
return EXIT_SUCCESS;

}
```

L'exécution du programme affiche : "oui et non"

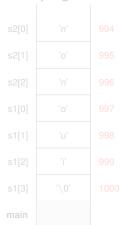


Question : qu'affiche le programme ci-dessous ?

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

int main(){
    char s1[4] = "oui";
    char s2[3] = {'n','o','n'};
    printf("%s et %s\n", s1, s2);
    return EXIT_SUCCESS;
}
```

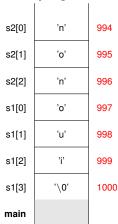
L'exécution du programme affiche : "oui et nonoui"



Question : qu'affiche le programme ci-dessous?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4
5 int main(){
6 char s1[4] = "oui";
7 char s2[3] = {'n', 'o', 'n'};
8 printf("%s et %s\n", s1, s2);
9 return EXIT_SUCCESS;
0 }
```

L'exécution du programme affiche : "oui et nonoui"



- Présentation
- Rappel : Types de données
 - Variables
 - Variables scalaires
 - Tableaux
 - Structures
 - Affectation de variables
- Rappel : Fonctions et paramètres

Structure

Séquence de variables, appelées **champs**, pouvant être de **types différents**.

Définition

- La définition d'un type structuré doit permettre au compilateur
 - de connaître ce nouvel objet.
 - de déterminer la taille de l'espace occupé en mémoire par chaque champ
- On place la définition en haut du programme (pour les programmes courts) ou plutôt dans un fichier headline (.h).

```
/* Dans un fichier point.h */
typedef struct point_s{
  float x;
  float y;
} point_t;
```

Structure

Séquence de variables, appelées champs, pouvant être de types différents.

Déclaration d'une variable structurée

- Une variable structurée se déclare comme une variable scalaire ou un tableau.
- la variable correspond à un emplacement mémoire de la taille totale de tous ses champs.

```
#include "point.h"

int main(){
    point_t p; /* variable locale */
    return EXIT_SUCCESS;
}
```

Accès aux champs

Soit p une variable de type point_t.

- Les deux champs de p sont des variables identifiées par p.x et p.y, respectivement.
- Cet accès vaut pour tout niveau d'imbrication de structures (par exemple une structure contenant un champs qui est une structure).

```
#include "point.h"

int main(){
    point.t p; /* variable locale */
    p.x=2;
    p.y=3;
    p.y=p.y+1;
    return EXIT_SUCCESS;
}
```

Initialisation

Une variable de type structuré peut être initialisée de plusieurs façons :

- totalement ou partiellement à la déclaration
- champ par champ ou par recopie des valeurs des champs d'une structure de même type.

```
#include "point.h"
int main(){
   point_t p1 = {-1.5, 5.0}; /* en respectant l'ordre des champs, sans les nommer *
   point_t p2 = { .x = -2.25 }; /* (partiellement) en nommant certains champs */
   point_t p3;
   p2.y = 6.75; /* initialisation du 2e champ de p2 */
   p3 = p1; /* chaque champ de p3 prend la valeur du champ correspondant de p1 */
   return EXIT_SUCCESS;
}
```

```
1 /* Dans un fichier point.h */
2 typedef struct point_s{
3 float x;
4 float y;
5 } point_t;
```

On veut définir la notion de cercle. Un cercle est défini par

- son centre
- son rayon.

```
/* Dans un fichier point.h */
typedef struct point_s{
  float x;
float y;
} point_t;
```

```
/* Dans un fichier cercle.h */
2 #include"point.h"
3 typedef struct cercle_s{
4 point_t centre;
5 double rayon;
6 } cercle_t;
```

Notez l'imbrication des acces pou accéder aux coordonnées du centre.

```
1 /* Dans un fichier point.h */
2 typedef struct point_s{
3 float x;
float y;
5 } point_t;
```

```
/* Dans un fichier cercle.h */
2 #include"point.h"
3 typedef struct cercle_s{
    point_t centre;
    double rayon;
6 } cercle_t;
```

```
/* Dans un fichier main.c */
2 #include "cercle.h"
3 int main() {
4 point_t p = {1.5, 4.0};
5 cercle_t c = {{0.25, -2.5}, 5.33};
6 printf("Cercle de centre (%.g,%g) et
6 de rayon %g\n",
7 c.centre_x,c.centre_y,c.rayon);
8 return EXIT_SUCCESS;
9 }
```

Notez l'imbrication des accès pour accéder aux coordonnées du centre.

Structures

```
1 /* Dans un fichier point.h */
2 typedef struct point_s{
3 float x;
4 float y;
5 } point_t;
```

```
/* Dans un fichier cercle.h */
2 #include"point.h"
3 typedef struct cercle_s {
    point_t centre;
    double rayon;
6 } cercle_t;
```

```
/* Dans un fichier main.c */
2 #include "cercle.h"
3 int main() {
4 point.t p = {1.5, 4.0};
5 cercle_t c = {{0.25, -2.5}, 5.33};
6 printf("Cercle de centre (%.g,%g) et
7 de rayon %g\n",
8 c.centre.x,c.centre.y,c.rayon);
9 return EXIT_SUCCESS;
0 }
```

Notez l'imbrication des accès pour accéder aux coordonnées du centre.

Exécution du programme

0.25	980
-2.5	984
5.33	988
1.5	996
4.0	1000
	-2.5 5.33 1.5

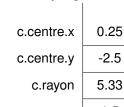
Structures

0

```
1 /* Dans un fichier point.h */
  typedef struct point_s{
    float x:
    float y;
    point_t:
1 /* Dans un fichier cercle.h */
2 #include "point.h"
3 typedef struct cercle_s{
    point_t centre;
    double rayon;
6 } cercle_t:
1 /* Dans un fichier main.c */
2 #include "cercle.h"
3 int main(){
    point_t p = \{1.5, 4.0\};
    cercle_t c = \{\{0.25, -2.5\}, 5.33\};
    printf ("Cercle de centre (%.g,%g) et
6
7
            de rayon %g\n",
8
           c.centre.x.c.centre.y.c.rayon);
9
    return EXIT_SUCCESS:
```

Notez l'imbrication des accès pour accéder aux coordonnées du centre.

Exécution du programme



1.5 p.x 4.0 p.y

main

980

984

988

996

1000

- Présentation
- Rappel : Types de données
 - Variables
 - Variables scalaires
 - Tableaux
 - Structures
 - Affectation de variables
- Rappel : Fonctions et paramètres

Affectations

Affectations

L'affectation entre deux expressions s'écrit :

donne des rôles tout à fait différents au deux expressions

- A gauche var doit être une variable qui soit une lvalue (left-hand value) : c'est-à-dire une variable pouvant être utilisée comme contenant d'une valeur.
- A droite expr est une expression pouvant être évaluée (appelé parfois rvalue): la valeur résultant de l'évaluation est alors affectée à la variable de la gauche de l'affectation.

Remarque: en "pseudo-code", on note cela

 $var \leftarrow expr;$

Exemple d'expression qui n'est pas une lvalue : un appel de fonction.

Affectation entre variables

L'affectation entre deux variables :

est ainsi le fait de mettre la valeur de var2 dans la variable var1.

Affectation entre variables

- Si var1 est de type simple : c'est une opération d'affectation
- Si var1 est une case d'un tableau. e.g. tab [2]=3 : c'est une opération d'affectation d'une variable simple :
- Si var1 est d'un type struct : c'est autant d'opérations d'affectation que de champs dans le type struct

Cas des tableaux

Un tableau n'est pas une Ivalue

• Si **var1** est un tableau : c'est une opération illicite, car un tableau n'est pas une lvalue

```
1 int tab1[10];
2 tab1++;
```

Affichage: error: lvalue required as increment operand

Copie d'un tableau

Pour copier un tableau dans un autre, il faut recopier chaque case une à une : "au prix" d'autant d'affectations qu'il y a de cases

```
1 for (i=0;i<taille;i++)
2 tab2[i]=tab1[i];
```

- Présentation
- Rappel : Types de données
- Rappel: Fonctions et paramètres
 - Déclarations, définitions et appels
 - Paramètres

Les fonctions : pour faire quoi?

Les fonctions : pour faire quoi?

- Éviter la duplication de code.
- En créant des bibliothèques de fonctions, on peut les réutiliser dans différents programmes.
- Un programme proprement découpé en fonctions est plus lisible.
- Prévoir la liste des fonctions nécessaires permet de mieux structurer un programme et facilite la conception.

Fonction

Une fonction est un sous-programme, un ensemble d'instructions. On distingue trois notions importantes associées aux fonctions :

- la déclaration
- la définition
- l'appel

Fonction: déclaration

La **déclaration** donne les informations indispensables pour que le compilateur reconnaisse une fonction et vérifie qu'elle est appelée conformément à son **prototype**, à savoir :

- les types des arguments passés à la fonction : à chaque appel, chacun des paramètres formels d'entrées reçoit une valeur d'un type fixé
- le nom de la fonction
- le type du résultat qu'elle renvoie (ou void si elle ne renvoie pas de résultat).

La déclaration de la fonction doit précéder tout appel à cette fonction dans le code du programme.

Exemple de déclaration

On veut créer une fonction qui additionne deux entiers positifs.

unsigned int addition(unsigned int, unsigned int);

Fonction: définition

La **définition** reprend le prototype de la déclaration (avec le nom des paramètres d'entrées) et ajoute le **bloc**, appelé **corps** de la fonction, constitué des instructions qui seront exécutées lors de chaque appel à la fonction.

- L'exécution d'un appel à une fonction qui renvoie une valeur doit se terminer par l'exécution d'une instruction return (avec comme argument une expression dont la valeur appartient au type du résultat de la fonction.)
- Le corps d'une fonction qui ne renvoie pas de valeur peut contenir des instructions return sans argument.
- L'exécution d'une instruction return met fin à l'exécution de l'appel.

Exemple de définition

```
unsigned int addition(unsigned int x,unsigned int y){
unsigned int result=x+y;
return result;
}
```

Fonction: appel

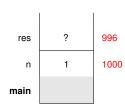
Utilisation de la fonction.

```
int main() {
 unsigned ir
 unsigned ir
 unsigned ir
 printf("%d-
 return EXIT
 }
      unsigned int n = 1;
      unsigned int res = addition(n, 2+n);
      printf("%d+2+%d = %d\n", n, n, res);
      return EXIT_SUCCESS;
```

```
unsigned int result=x+y;
return result;

int main() {
  unsigned int n = 1;
  unsigned int res = addition(n, 2+n);
  printf("%d+2+%d = %d\n", n, n, res);
  return EXIT_SUCCESS;
}
```

unsigned int addition (unsigned int x, unsigned int y) {



```
unsigned int addition(unsigned int x,unsigned int y){
unsigned int result=x+y;
return result;
}

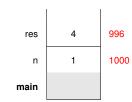
int main() {
unsigned int n = 1;
unsigned int res = addition(n, 2+n);
printf("%d+2+%d = %d\n", n, n, res);
return EXIT_SUCCESS;
}
```

result	4	968
у	3	972
х	1	976
addition(1,2+n)		
res	?	996
n	1	1000
main		

La fonction renvoie 4 et on dépile.

```
unsigned int addition(unsigned int x,unsigned int y){
unsigned int result=x+y;
return result;

int main() {
unsigned int n = 1;
unsigned int res = addition(n, 2+n);
printf("%d+2+%d = %d\n", n, n, res);
return EXIT_SUCCESS;
}
```



Le programme affiche "1+2+1=4"

Fonction: imbrication d'appels

Il est possible d'imbriquer les appels de fonction, d'utiliser le résultat d'un appel comme argument d'un autre appel (à la même fonction ou à une autre). Exemple : on souhaite calculer 4+3+2.

```
int res=addition(4,addition(3,2));
```

- Présentation
- Rappel : Types de données
 - 3 Rappel : Fonctions et paramètres
 - Déclarations, définitions et appels
 - Paramètres

Paramètres formels et variables locales d'une fonction

Paramètres formels d'une fonction

- Ils sont déclarés dans l'entête de la fonction.
- Leur portée est le bloc contenant le corps de la fonction. Ils jouent le rôle de variable locale.
- Par convention, leur nom est toujours en minuscule(s).

En mémoire

Un paramètre de fonction est stocké dans la pile d'appel.

Paramètres formels et variables locales d'une fonction

Il est donc illégal de faire

```
1 int f(int n, int m)
    int res = 3*n:
    int m = 2 ; /** DECLARATION ILLEGALE */
      /** => ERREUR A LA COMPILATION */
    return res + m;
```

Par contre, on "pourrait" faire

```
1 int f(int n, int m)
    int res = 3*n:
```

scanf("%d", &m);

8

int m: /** DECLARATION LEGALE MAIS ??? */

return res + m: 9 Mais c'est très piégeux d'écrire une définition de fonction comme celle-ci (et parfaitement inutile)!

Paramètres effectifs

Paramètres effectifs de l'appel d'une fonction

unsigned int addition (unsigned int x, unsigned int y) {

- Ce sont les valeurs passées à la fonction lors de l'appel.
- La fonction reçoit les valeurs passées lors de l'appel

```
unsigned int result=x+y;
return result;

int main() {
  unsigned int n = 1;
  unsigned int res = addition(n, 2+n);
  printf("%d+2+%d = %d\n", n, n, res);
  return EXIT_SUCCESS;
}
```

Ici lors de l'appel, la fonction addition reçoit les valeurs

- 1 en 1er paramètre : ce qui initialise le paramètre formel x
- 3 en 2ème paramètre : ce qui initialise le paramètre formel y

Passage de paramètres

Passage par copie

- Seules les valeurs sont transmises à une fonction.
- Une variable scalaire utilisée en paramètre effectif n'est donc pas impactée par l'appel.
- On parle de "passage par copie"

Illustration de ce "non passage de paramètre"

"Passage de paramètres" : utilisation du "return"

Utilisation du return

Un moyen de pouvoir récupérer une valeur dans une variable après un appel est d'utiliser le return.

"Passage de paramètres" : utilisation du "return"

Utilisation du return avec un struct

1 /* Dans un fichier point.h */

Et tout ceci fonctionne également avec un struct.

```
typedef struct point_s{
    float x;
    float v;
    point_t:
1 #include "point.h" 2
  point_t pousseAGauche(point_t p){
    p.x = p.x -1;
    return p:
6
8 int main(){
    point_t p = \{5.2, 7.5\};
    printf("p = (%g, %g)\n", p.x, p.y); /* Affiche p = (5.2, 7.5) */
    p = pousseAGauche(p);
    printf("p = (%g, %g)\n",p.x,p.y); /* Affiche p = (4.2, 7.5) */
    return EXIT_SUCCESS:
```

"Passage de paramètres" : utilisation du "return"

Copie d'un struct

Lors

- d'une affectation entre deux struct (de même type)
- d'un passage par copie
- (et donc lors d'une affectation d'une variable après un return)

tous les champs sont recopiés un à un, et avec toutes les imbrications.

Cela entraîne des affectations inutiles et qui consomment du temps!

"Passage de paramètres" : utilisation des variables globales

Utilisation des variables globales

- En utilisant des variables globales, une variable est visible dans toutes les fonctions.
- Il n'y a pas besoin de passer de paramètres!
 Elle peut être modifiée partout!

Attention à la mauvaise pratique

Une variable globale peut être donc modifiée "par erreur" par une fonction "secondaire" oubliée dans un coin!

Portée des variables et bonne pratique

Que se passe-t-il si...

```
int X = 4; /* Variable globale */

void f(){
   int X = 7; /* Variable locale */

int main(){
   f();
   printf("X = %d", X);
   return EXIT_SUCCESS;
}
```

lci le nom de variable X désigne à la fois

- une variable globale
 - une variable locale à la fonction ${\bf f}$ de nom identique a une variable globale

Alors la variale locale **masque** la variable globale pendant l'exécution de la fonction **f**, i.e. la variable globale est inaccessible et **X** désigne la variable locale : c'est très dangereux car **piégeux!**

En mémoire

L'usage des variables globales est déconseillé pour les gros programmes.

"Passage de paramètres" : tableaux

Passage d'un tableau lors d'un appel

Passer un tableau en paramètre effectif lors d'un appel peermet de modifier le contenu des cases d'un tableau!

Ce n'est pourtant pas une contradiction avec le passage par copie lors d'un appel!

"Passage de paramètres" : tableaux

Passage d'un tableau lors d'un appel

- le type int T[] est un autre nom du type int *T
- **T** est un pointeur, c'est-à-dire une variable contenant une adresse mémoire (ici un tableau).
- T[3] est le déférencement *(T+3) qui permet d'accéder au contenu

Voici une traduction "moins lisible" du code précédent, mais qui dévoile l'usage des pointeurs

```
void multiplie(int *T, int taille){
   int i;
   for (i=0; i<taille; i++)
        *(T+i) = *(T+i) * 2;
}

int main(){
   int T[3] = {2, -3, 1};
   printf("T = [ %d %d %d ]\n",*T, *(T+1), *(T+2));
   multiplie(T, 3);
   printf("T = [ %d %d %d ]\n",*T, *(T+1), *(T+2));
   return EXIT_SUCCESS;
}</pre>
```

Nous verrons cela dans les cours suivants!