Cours de Programmation Impérative: Rappels et Pile d'appel

Julien David

A101 - david@lipn.univ-paris13.fr

Lundi 07 Janvier 2012

Organisation

Cours

- Première partie de semestre : 1h30 de cours.
- Deuxième partie du semestre : cours supplémentaires le vendredi.

TD/TP

- Semaines 1 à 4 : 3h de TD ;
- Semaines 5 à 11 : 1h30 de TD, 1h30 de TP.

Soutenances de projet

- Semaine 12.

Notation

Les différentes notes

- Note de contrôle continu : devoir à rendre à vos chargés de TD **par mail**, des interros.
- Un partiel à la moitié du semestre,
- Un projet à faire en binôme.

Précision

- Les devoirs données en TD/TP ne sont pas facultatifs.
- On apprend à programmer en programmant, pas simplement en révisant son cours.

Plan du cours

1. Variables
   - Locales
     - Scalaires
     - Tableaux
     - Structures de données
   - Globales
2. Boucles : while / for
3. Fonctions
   - Déclaration,
   - Définition,
   - Appel.

Le

Variable : définition

Une variable est une zone de mémoire caractérisée par :

- un nom
- un type
- une valeur
- une adresse

Example

```
int x=4;
```

- Nom : `x`
- Type : entier (`int`)
- Valeur : `4`
- Une adresse : `80` (déterminée à l'exécution du programme)

Variable : définition

Une variable est une zone de mémoire caractérisée par :

- un nom
- un type
- une valeur
- une adresse

Example

```
int x=4;
```

- Nom : `x`
- Type : entier (`int`)
- Valeur : `4`
- Une adresse : `80` (déterminée à l'exécution du programme)

Locale/ Globale

Variable locale

- Déclaration : à **l'intérieur** d'une fonction
- Accès : durée de l'exécution **de la fonction**
- Convention : Le nom d'une variable locale est toujours en **minuscule**.

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

En mémoire

Une variable locale est stockée dans **la pile d'appel**

Locale/ Globale

Variable globale

- Déclaration : à **l'extérieur** d'une fonction
- Accès : durée de l'exécution **du programme**
- Convention : Le nom d'une variable globale est toujours en **majuscule**.

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

En mémoire

Une variable globale est stockée dans la **zone de données**

Variable scalaire

Variable scalaire

Variable destinée à contenir une valeur atomique, une unique “donnée”.

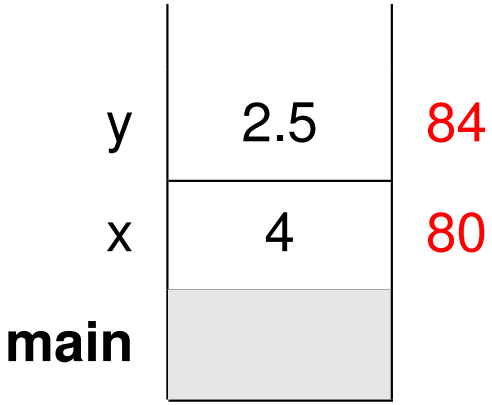
Exemple

- un nombre
  - entier : `int`, `unsigned int`, `long int`, `long long int`
  - réel : `float`, `double`
- une lettre/un caractère : `char`

Variable scalaire locales

```
1 int main(){
2   int x=4;
3   float y=2.5;
4   printf("%d\n",x);
5   return EXIT_SUCCESS;
6 }
```

Exécution du programme



Le programme affiche : "4".

Tableaux

Tableaux

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

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

Exemple

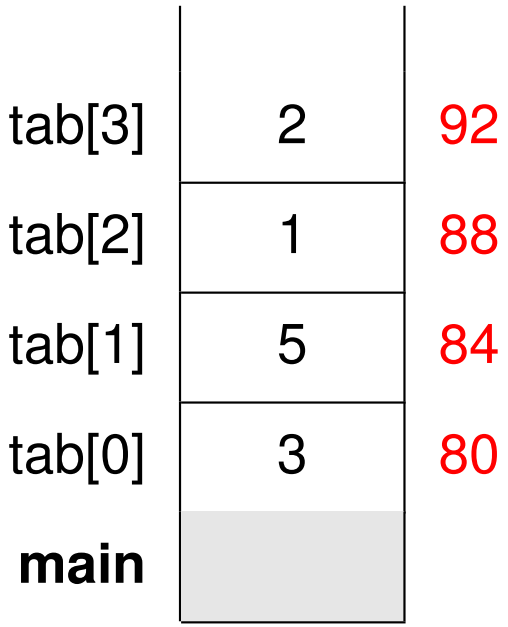
```
int tab[2]={1,2};
```

- Nom : `tab`
- Type : Tableau d'entiers : chaque case est de type `int`.
- Valeur de chaque case : `tab[0]= 1, tab[1]= 2`
- Adresse : l'adresse de `tab` est la même que `tab[0]`.

Tableaux

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

Exécution du programme



Le programme affiche :
"La 4eme case du tableau vaut 2"

Tableaux à plusieurs dimensions

Tableaux à deux dimensions

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

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

Exemple

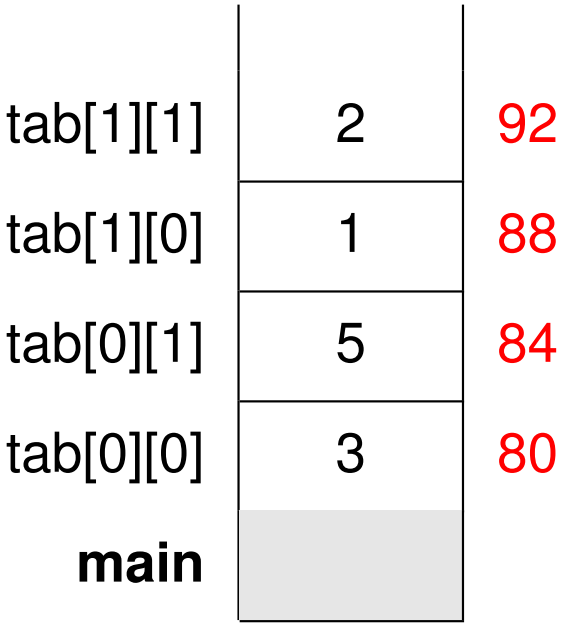
```
int tab[3][2]={ {3,5} , {1,2} , {4,1} };
```

	0	1
0	3	5
1	1	2
2	4	1

Tableaux à plusieurs dimensions

```
1 int main(){
2   int tab[2][2]={ {3,5} , {1,2} };
3   printf("La case situee a la ligne 0 et la colonne 1 vaut %d\n",tab[0][1]);
4   return EXIT_SUCCESS;
5 }
```

Exécution du programme



Le programme affiche :

"La case situee à la ligne 0 et la colonne 1 vaut 5"

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, ...
- la fin d'une chaîne est **obligatoirement** signalée par un $'\backslash 0'$.
- 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 :

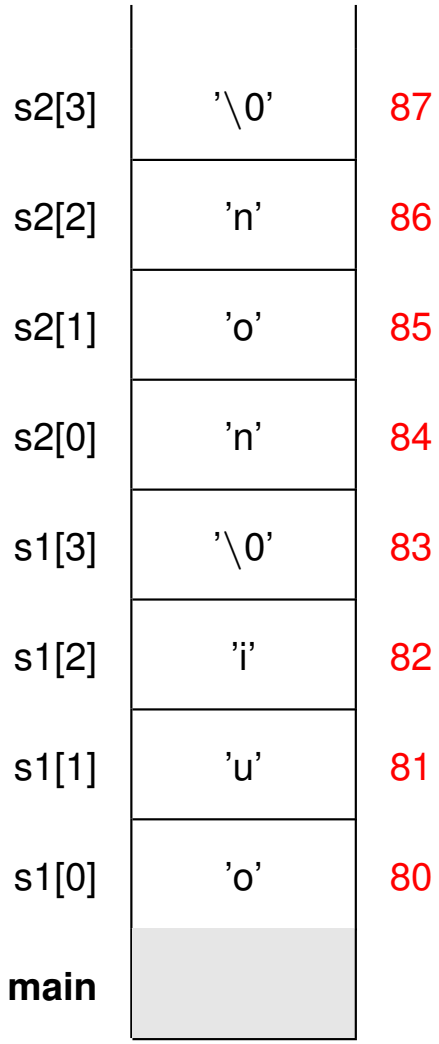
'c'	'o'	'u'	'r'	's'	'\0'

Chaînes de caractères

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

Le programme affiche :
"oui et non"

Exécution du programme



Question

Qu'affiche ce programme ?
(en pratique, la réponse dépends de la machine)

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

Structures

Structures

Séquence de variables pouvant être de **types différents**.

Définition

La définition d'un type structuré permet au compilateur de comprendre ce nouvel objet.

- Convention : on l'écrit dans le fichier `nom_structure.h`

```
1 /* point.h */
2 struct point_s{
3   int x;
4   int y;
5 };
6 typedef struct point_s point_t;
```

Structures

Séquence de variables pouvant être de **types différents**.

Déclaration d'une variable structurée

- Une variable structurée se déclare comme une variable scalaire ou un tableau :
  - au début d'une fonction si c'est une variable locale.
  - en dehors d'une fonction si c'est une variable globale.

```
1 #include"point.h"
2
3 point_t P; /* Variable Globale */
4
5 int main(){
6   point_t c; /* Variable locale */
7   return EXIT_SUCCESS;
8 }
```

Structures

Séquence de variables pouvant être de **types différents**.

Initialisation

Une variable de type structurée peut-être initialisée de plusieurs façons :

- Comme un tableau, au moment de la déclaration.
- Chaque champs séparément, après la déclaration.

```
1 #include"point.h"
2 int main(){
3   point_t p1={1,5}; /* Initialisation comme un tableau*/
4   point_t p2;
5   /* Initialisation champs par champs */
6   p2.x=6;
7   p2.y=3;
8   return EXIT_SUCCESS;
9 }
```

Structures

```
1 /* point.h */
2 struct point_s{
3   int x;
4   int y;
5 };
6 typedef struct point_s point_t;
```

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

- Son centre
- Son rayon

Structures

```c
/* point.h */
struct point_s{
  int x;
  int y;
};
typedef struct point_s point_t;
```

```c
/* cercle.h */
#include"point.h"
struct cercle_s{
  point_t centre;
  double rayon;
};
typedef struct cercle_s cercle_t;
```

Structures

```c
/* point.h */
struct point_s{
  int x;
  int y;
};
typedef struct point_s point_t;
```

```c
/* cercle.h */
#include"point.h"
struct cercle_s{
  point_t centre;
  double rayon;
};
typedef struct cercle_s cercle_t;
```

```c
/* main.c */
#include"cercle.h"
int main(){
  point_t p={1,4};
  cercle_t c={{0,-2},5.3};
  return EXIT_SUCCESS;
}
```

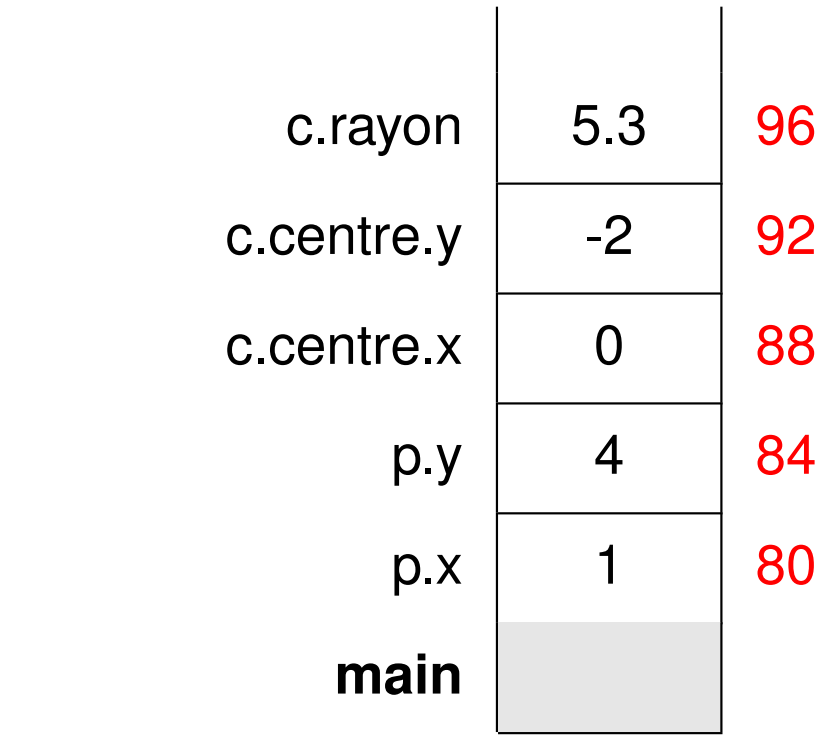
Structures

```c
/* point.h */
struct point_s{
  int x;
  int y;
};
typedef struct point_s point_t;
```

```c
/* cercle.h */
#include"point.h"
struct cercle_s{
  point_t centre;
  double rayon;
};
typedef struct cercle_s cercle_t;
```

```c
/* main.c */
#include"cercle.h"
int main(){
  point_t p={1,4};
  cercle_t c={{0,-2},5.3};
  return EXIT_SUCCESS;
}
```

Exécution du programme



Boucle

- for
- while

Boucles

Les boucles `for` et `while` permettent de répeter plusieurs fois une même séquence d'instructions.
Elles se caractérisent par :

A) une variable de boucle qui doit être initialisée,

B) un test d'arrêt,

C) la mise à jour de la variable de boucle.

Syntaxe du `for`

```
1 for(i=0/*A)*/ ; i<4 /*B)*/; i++/*C)*/){
2   Instructions;
3 }
```

Syntaxe du `while`

```
1 i=0; /* A) */
2 while(i<4 /*B) */ ){
3   Instructions;
4
5   i++; /* C) */
6 }
```

Parcours d'un tableau

```
1 int main(){
2   int tab[3]={5,2,7};
3   int i;
4   for(i=0;i<3;i++){
5     if(tab[i]%2 == 0) /* si le nombre dans la case i est pair*/
6       printf("%d\n",tab[i]);
7   }
8   return EXIT_SUCCESS;
9 }
```

Exécution du programme

i	?	92
tab[2]	7	88
tab[1]	2	84
tab[0]	5	80
main		

i	0,1,2,3	92
tab[2]	7	88
tab[1]	2	84
tab[0]	5	80
main		

Le programme affiche "2"

Fonctions

Fonctions

Fonction

Une fonction est un sous-programme, un ensemble d'instructions.
On distingue trois notions importantes autour des fonctions.

- la déclaration
- la définition
- l'appel

Fonctions

Fonction : déclaration

La déclaration, ou le prototype, d'une fonction permet au compilateur de savoir qu'une fonction existe et quels sont ses paramètres.
La déclaration d'une fonction est caractérisée par :

- ses paramètres d'entrées : les types des arguments,
- som nom,
- son résultat : le type du résultat.

Exemple

On veut créer une fonction qui additionne deux entiers positifs.

```
1 unsigned int addition(unsigned int x,unsigned int y);
```

- som nom : addition,
- ses paramètres : deux entiers positifs x et y (unsigned int),
- son résultat : un entier positif.

Fonction : définition

La définition d'une fonction désigne la description de l'ensemble des instructions qui seront effectuées par la fonction.

- Convention : la première ligne est la même que celle de la déclaration.

```
1 unsigned int addition(unsigned int x,unsigned int y){
2   unsigned int result=x+y;
3   return result;
4 }
```

Fonction : appel

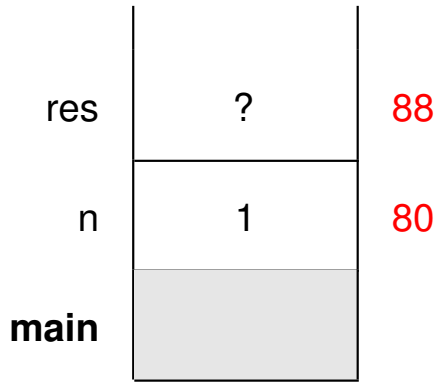
Utilisation de la fonction.

```
1 int main(){
2   unsigned int n=1;
3   unsigned int res=addition(n,3);
4   printf("%d+3 = %d\n",n,res);
5   return EXIT_SUCCESS;
6 }
```

Fonctions

```
1 unsigned int addition(unsigned int x,unsigned int y){
2   unsigned int result=x+y;
3   return result;
4 }
```

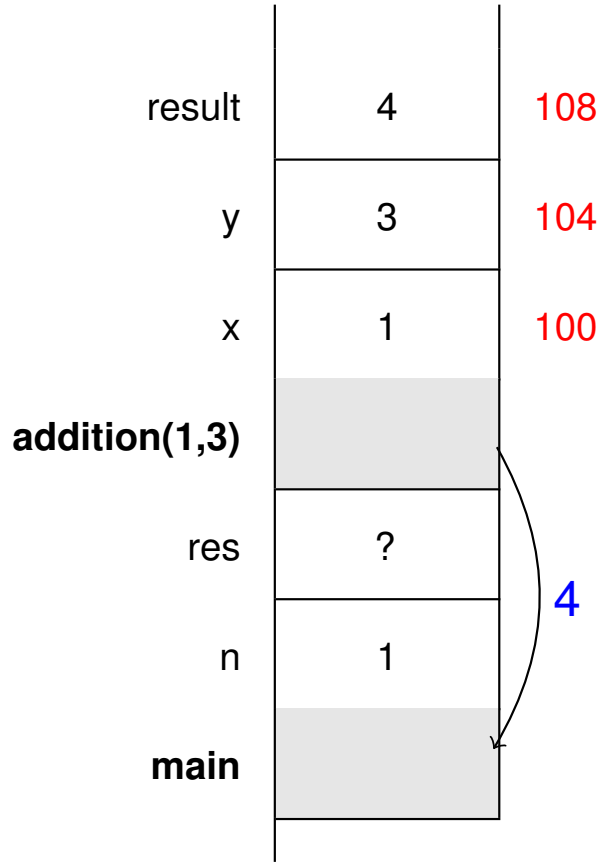
```
1 int main(){
2   unsigned int n=1;
3   unsigned int res=addition(n,3);
4   printf("%d+3 = %d\n",n,res);
5   return EXIT_SUCCESS;
6 }
```



Fonctions

```
1 unsigned int addition(unsigned int x,unsigned int y){
2   unsigned int result=x+y;
3   return result;
4 }
```

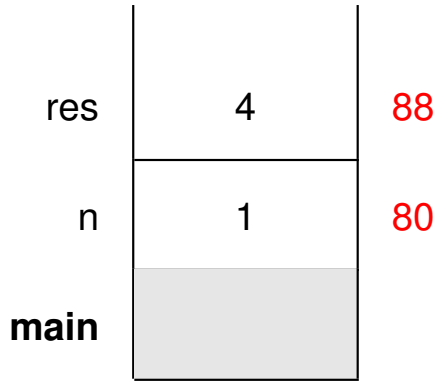
```
1 int main(){
2   unsigned int n=1;
3   unsigned int res=addition(n,3);
4   printf("%d+3 = %d\n",n,res);
5   return EXIT_SUCCESS;
6 }
```



Fonctions

```
1 unsigned int addition(unsigned int x,unsigned int y){
2   unsigned int result=x+y;
3   return result;
4 }
```

```
1 int main(){
2   unsigned int n=1;
3   unsigned int res=addition(n,3);
4   printf("%d+3 = %d\n",n,res);
5   return EXIT_SUCCESS;
6 }
```



Le programme affiche "1+3=4"

Fonctions

Fonction : à savoir

Il est possible d'imbriquer les fonctions, d'utiliser le résultat d'une fonction comme paramètre d'une autre fonction.
Exemple : on souhaite calculer $4 + 3 + 2$.

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

Fonction : erreur classique

Lors d'un appel de fonction, on ne réécrit pas les types des variables.

```
1 unsigned int res=addition(unsigned int n,unsigned int m);
```

Cette instruction ne veut strictement rien dire.

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.

Les fonctions et structures

Bien coder des structures c'est :

Fabriquer une bibliothèque de fonctions pour manipuler la structure.

- Création/Initialisation
- Destruction (si besoin)
- Afficher
- ...

```c
void afficher_point(point_t p){
  printf("%d %d\n",p.x,p.y);
}

point_t creer_point(int x,int y){
  point_t result;
  result.x=x;
  result.y=y;
  return result;
}
```

```c
int main(){
  point_t p[1000];
  int i;
  for(i=0;i<1000;i++){
    p[i]=creer_point(i,i);
    afficher_point(p[i]);
  }
  return EXIT_SUCCESS;
}
```

Fonctions récursives

Définition

Les fonctions récursives sont des fonctions qui, dans leur définition, contiennent un appel à elles-mêmes.

Mais où peut-on en savoir plus ?

En TD, cet après midi ou vendredi prochain.