INFORMATIQUE
Sciences
Université Paris Cité

Langage C –Cours 2

Lélia Blin

lelia.blin@irif.fr

2023 - 2024

- **Rappels et Fin présentation basique**
- **Tableaux (de taille fixe)**
- **Structures**
- **Énumérations**

Signed vs unsigned

Types signés

- Les types de données signés permettent de représenter à la fois des **nombres positifs et négatifs**.
- Ils réservent généralement le bit le plus significatif (le bit le plus à gauche) pour représenter le signe du nombre (0 pour positif, 1 pour négatif).
- Par exemple, un entier signé de 8 bits peut stocker des valeurs de -128 à 127.

Types non signés :

- Les types de données non signés ne peuvent représenter que des nombres positifs ou nuls.
- Ils utilisent tous les bits pour représenter la valeur numérique, ce qui signifie qu'ils peuvent stocker des nombres plus grands que leurs homologues signés.
- Par exemple, un entier non signé de 8 bits peut stocker des valeurs de 0 à 255.

Le choix entre les types signés et non signés dépend de la nature des données que vous prévoyez de manipuler dans votre programme

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

int main(){
        int a=-1;
        unsigned b=1;
        if(a>b){
                puts("a < b");
        }else{
                puts("a >= b");
        }
}
```

test-egalite.c

<u>Remarque: </u> `unsigned` est équivalant a `unsigned int`

- Lorsque on compare `a` et `b` dans l'expression `a > b`, la **promotion** des types est effectuée.
  - Dans ce cas, `a` est converti en `unsigned int` pour que les deux valeurs puissent être comparées.
- La valeur de `a` est `-1` en tant qu'entier signé.
  - Lorsqu'elle est convertie en `unsigned int`, elle devient une grande valeur positive (car les bits sont interprétés différemment lorsqu'ils sont interprétés comme des entiers non signés).
- après la conversion, `a` devient une valeur beaucoup plus grande que `b`, ce qui fait que l'expression `a > b` est évaluée comme vraie, et par conséquent, "a < b" est affiché.
- La valeur de a en unsigned int est 4294967295

Pour éviter un comportement inattendu, assurez-vous de comparer des variables de même type signé ou non signé

printf et puts

- Servent à écrire sur la sortie standard
- **Syntaxe :**
  - **puts(chaine)**
  - **printf(chaine,expression-1,expression-2,...,expression-n)**
- puts rajoute un saut à la ligne à l'affichage et ne permet pas d'afficher des expressions
- La chaine donnée à printf contient des spécifications de format pour chaque expression

Formats pour printf :

- int :%d (décimale signé)
- unsigned :%u (décimale non signé),%x (hexadécimale non signé)
- long : %ld (décimale signé)
- unsigned long :%lu (décimale non signé)
- double :%lf (décimale virgule fixe),%le (décimale notation exponentielle)
- char : %c

Affichage de printf

- On peut maîtriser un peu plus l'affichage du format. Par exemple :
  - %10u : au minimal 10 caractères sont réservés pour imprimer l'entier et la donnée est cadrée à droite (%-10u veut dire que la donnée est cadrée à gauche)
  - %.12lf : le flottant est imprimé avec 12 chiffres après la virgule
  - %10.2lf : on réserve au minimal 12 caractères pour imprimer le flottant (en comptant la virgule) et les deux derniers sont pour les chiffres après la virgule

```c
#include <stdio.h>
int main(void){
unsigned x=123;
unsigned y=123456789;
double a=12.3456;
double b=12345.6789;
double c=0.1;
printf("***%6u***\n",x);
printf("***%-6u***\n",x);
printf("***%6u***\n",y);
printf("***%-6u***\n",y);
printf("***%.1lf***\n",a);
printf("***%.6lf***\n",a);
printf("***%4.6lf***\n",b);
printf("***%4.1lf***\n",c);
printf("***%-4.1lf***\n",c);
}
```

printf-format.c

scanf (1)

- Sert à récupérer des données de l'entrée standard
- **Syntaxe :**
  - **scanf(chaine,argument-1,argument-2,...,argument-n)**
- La chaine donnée à scanf contient des spécifications de format pour chaque expression que l'on attend
- Les formats sont les mêmes que ceux pour printf

scanf (2)

- Les arguments sont quant à eux des pointeurs vers des données du type de format :
  - on verra plus tard ce qu'est un pointeur
  - à retenir pour l'instant, vous faîtes précéder votre nom de variable où stocker la donnée lue par &
- Les espaces dans la chaine compte comme des espaces ou des retours à la ligne (à éviter)
- La fonction renvoie 0 si il y a eu un problème et le nombre de données lues correctement sinon
- **ATTENTION : Limitez vous à un usage très basique de cette fonction**

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

int main () {
  int x=0;
  int y=0;
  int r=scanf("HELLO %d BYE",&y);
  printf("%d donnees : %d %d \n",r,x,y);
  return EXIT_SUCCESS;
}
```

test-scanf.c

Fonctions récursives

- Les fonctions C peuvent être récursives
- Une fonction peut donc s'appeler elle-même
- Il faut faire attention (comme d'habitude) :
  - à la terminaison de la récursion
  - aux cas de base

```c
#include <stdio.h>
#include <stdlib.h>
unsigned puissance(unsigned,unsigned);
int main () {
  int r=puissance(3,4);
  printf("Test : 3 puissance 4 = %u\n",r);
  return EXIT_SUCCESS;
}
/*Calcule a a la puissance n*/
unsigned puissance(unsigned a, unsigned n){
  if(n==0){
    return 1;
  }else{
    return(a*puissance(a,n-1));
  }
}
```

recur-puissance.c

define

- `#define` permet de définir :
  - des constantes
  - des **macros** avec paramètres
- Le pré-processeur remplacera à la compilation chaque macros par sa définition

```
#define N 5
#define SUM(a,b) (a+b)
#define PRODUIT(a,b) (a*b)
```

- Pour une macro avec paramètre on a des arguments entres parenthèses juste après la parenthèse
- **Attention** au remplacement textuel `PRODUIT(x+y,u+t)` sera remplacé par `x+y*u+t`

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

#define PRODUIT(a,b) a*b
int main () {
  int x=2;

  int y=3;
  int t=0;
  int u=5;
  int z=PRODUIT(x+y,t+u);
  int z2=PRODUIT((x+y),(t+u));
  printf("%d \n",z); //affiche 7
  printf("%d \n",z2); //affiche 25
  return EXIT_SUCCESS;

}
```

test-macros.c

Branchement inconditionnel

- Il y a deux instructions de branchement inconditionnel que l'on peut utiliser dans une boucle :
- **break :** arrête l'exécution de la boucle (sans que le test de la boucle ne soit réalisé)
- **continue** : arrête l'exécution courante de la boucle et revient à la boucle au moment où le test est réalisé
- **goto label ;** -> permet de 'sauter' au morceau de code étiqueté par l'étiquette label
- Ces instructions sont à utiliser avec parcimonie et on peut souvent s'en passer (en particulier le goto **à éviter**)

```c
#include <stdlib.h>
#include <stdbool.h>

bool estPremier(unsigned);

int main () {
  puts("Donnez un entier positif.");
  unsigned x=0;
  int r=scanf("%u",&x);
  if(r>0){
    if(estPremier(x)){
      printf("%u est premier\n",x);
    }else{
      printf("%u n'est pas premier\n",x);
    }
  }else{
    puts("Mauvais argument");
  }
  return EXIT_SUCCESS;
}

bool estPremier(unsigned n){
  bool b=true;
  if(n==0 || n==1){
    return false;
  }
  for(unsigned i=2;i<n;++i){
    if(n%i==0){
      b=false;
      break; //On arrete la boucle for des que l'on trouve un diviseur
    }
  }
  return b;
}
```

test-break.c

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

int main () {
  for(unsigned i=0;i<12;++i){
    printf("Valeur de i :%u\n",i);
    if(i%2==0){
      continue;
      printf("Jamais atteint\n");
    }else{
      for(unsigned j=0;j<i;++j){
        printf("*");
      }
      printf("\n");
    }
  }
  return EXIT_SUCCESS;
}
```

test-continue.c

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

int main () {
        unsigned x = 120;
        goto OH;
        printf("%u\n", x);

        OH:
        puts("OHOH");
        puts("Suite du code");
        return EXIT_SUCCESS;

}
```

test-goto2.c

Tableaux de taille fixe

- Déclaration de tableaux de taille fixe
- en C, la taille d'un tableau doit être une constante au moment de la compilation.
    - Cela signifie que vous ne pouvez pas utiliser une variable pour spécifier la taille d'un tableau lors de sa déclaration

Déclaration de la taille d'un tableau

- Pour les déclarer : **type nom-du-tableau[nombre d'elements] ;**
- Par exemple un tableau de 5 entiers :

```c
int tab[5];
```

- Parfois on encodera la taille de ces tableaux dans une constante donnée par une macro

```c
#define N 5
int main() {
        int tableau[N];
        ...
```

Indices

- On accède ensuite aux éléments du tableau en utilisant des entiers numérotés de 0 (pour la première case) à nombre d'elements-1 (pour la dernière case)

Parcours de tableau

- utilisation de `size-t` pour les indices du tableau
  - `size_t` est un type entier non signé utilisé pour représenter la taille des objets en mémoire
  - type est principalement utilisé pour les opérations liées à la gestion de la mémoire, telles que :
    - l'indexation des tableaux,
    - le calcul de la taille des données ou
    - la manipulation de pointeurs.
  - librairie `<stddef.h>` d'entiers non signés utilisé pour stocker des tailles
  - Le format `%zu` est utilisé pour afficher une valeur de type `size_t`.

Taille du tableau

- **Attention à ne pas dépasser la taille du tableau**
- En toute généralité, on ne peut pas connaître la taille d'un tableau
  - en particulier si il est donné en argument d'une fonction
  - On peut utiliser `sizeof(tab)` pour obtenir la taille totale en octets du tableau
    - autrement dit la taille de chaque élément multipliée par le nombre total d'éléments dans le tableau
  - mais **mieux vaut connaître toujours la taille des tableaux manipulés**

```c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h> //pour utiliser size_t

#define N 5

int main () {
  unsigned tab[5];
  for(size_t i=0;i<5;++i){
    tab[i]=2*i;
  }
  for(size_t i=0;i<5;++i){
    printf("tab[%zu] vaut %u\n",i,tab[i]);
        }
}
```

test-tab1.c

Initialisation (1)

- Pour initialiser un tableau, **au moment de sa déclaration** :
  - **type nom-du-tableau[N] ={val1, … valN };**
- On donne les N valeurs du tableaux
- Si on donne moins de valeurs, par exemple $k < N$, cela initialisera les k premières ‘cases’ du tableau et les autres cases vaudront 0

```
#define N 5
int main () {
        int tab[N]={1,2}
        for(size_t i=0;i<N;++i){
                printf("tab[%zu]=%u,",i,tab[i]);
        }
}//tab[0]=1,tab[1]=2,tab[2]=0,tab[3]=0,tab[4]=0
```

Initialisation (2)

- On peut aussi spécifier la valeur que de certaines cases :
  - **type nom-du-tableau[N] ={ [i1]=v1,[i2]=v2,...,[ik]=vk };**
  - Dans ce cas, $i1, \ldots, ik$ sont des indices corrects du tableau (dont la valeur est comprise entre 0 et N-1.

```
#define N 6
int main () {
        int tab[N]={[2]=1,[4]=2};
        for(size_t i=0;i<N;++i){
        printf("tab[%zu]=%u,",i,tab[i]);
        }
}//tab[0]=0,tab[1]=0,tab[2]=1,tab[3]=0,tab[4]=2,tab[5]=0
```

Remarque

- I**MPORTANT :** on ne peut faire ces initialisations qu'à la déclaration
- **On ne peut pas faire :**

```
int tab[5]={1,2,3,4,5};
tab={0,3,-10,12,13} ;
```

Mais vous pouvez changer les valeurs une à une

```
tab[0] = 0;
tab[1] = 3;
tab[2] = -10;
...
```

Règles et bonnes pratiques

- Si tab et tab2 sont des tableaux, **on ne peut pas faire :**
  - `tab1=tab2`
  - pour dire que tab1 reçoit le tableau tab2
- **On ne peut pas comparer par égalité deux tableaux**, en faisant :
  - `tab1==tab2`
- si on veut tester si ils contiennent les mêmes données, il faut les parcourir

Règles et bonnes pratiques

- Une fonction peut prendre en argument des tableaux (**et leur taille!!!!!**)
  - Exemple une fonction qui fait la somme des éléments d'un tableau d'entiers non signés :

```
unsigned somme(unsigned t[],size_t taille)
```

- Une fonction ne retournera pas de tableaux (dans ce cours)
  - On verra qu'une fonction peut modifier le contenu d'un tableau

```c
#define N 10
unsigned somme(unsigned[],size_t);
int main () {
  unsigned tab[N];
  for(size_t i=0;i<N;++i){
    tab[i]=i+1;
  }
  unsigned s=somme(tab,N);
  printf("%u \n",s);
}
unsigned somme(unsigned t[],size_t taille){
  unsigned r=0;
  for(size_t i=0;i<taille;++i){
    r+=t[i];
    }
return r;
}
```

test-taille.c

Tableaux comme arguments de fonctions

- En C, le passage des arguments aux fonctions se fait **par valeur**
  - Si l'on a unsigned f(unsigned x,unsigned y) {....}
  - Et que l'on faut l'appel f(a,b) (où a et b sont des variables)
  - Alors la valeur de a est 'copiée' dans x et celle de b est 'copiée' dans b
  - L'exécution de la fonction n'a (normalement) pas d'incidence sur le contenu des variables a et b
- Quand on passe un tableau en arguments d'une fonction, ce que l'on passe en réalité c'est un **pointeur** vers le tableau (pour faire court, l'adresse du tableau), donc la fonction peut modifier l'intérieur du tableau.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

unsigned f(unsigned);

int main () {
  unsigned a=2;
  unsigned b=f(a);
  printf("a vaut %u et b vaut %u\n",a,b);
}

unsigned f(unsigned x){
  x+=10;
  return x;
}//a vaut 2 et b vaut 12
```

test-valeur.c

```c
#include <stdio.h>

unsigned f(unsigned[]);

int main () {
  unsigned a[1]={2};
  unsigned b=f(a);
  printf("a[0] vaut %u et b vaut %u\n",a[0],b); //a[0] vaut 12 et b vaut 12
}

unsigned f(unsigned x[]){
  x[0]+=10;
  return x[0];
}
```

test-tabarg.c

Fonctions remplissant des tableaux

- Dans ce cours, on évitera de faire des fonctions qui créent des tableaux
- Mais on peut créer le tableau avant de faire l'appel à la fonction et la fonction ne fait que le remplir
- **Il ne faut toujours pas oublier de fournir la taille des tableaux manipulés aux fonctions**

```c
#include <stdlib.h>
#include <stddef.h>

#define N 10

void copie(unsigned[],unsigned[],size_t);
void affiche(unsigned[],size_t);

int main(){
  unsigned tab1[N];
  unsigned tab2[N];
  tab1[0]=1;
  for(size_t i=1;i<N;++i){
    tab1[i]=tab1[i-1]+i;
  }
  copie(tab1,tab2,N);
  affiche(tab2,N);
  return EXIT_SUCCESS;
}

void copie(unsigned t1[],unsigned t2[],size_t taille){
  for(size_t i=0;i<taille;++i){
    t2[i]=t1[i];
  }
}

void affiche(unsigned t[],size_t taille){
  for(size_t i=0;i<taille;++i){
    printf("Indice %zu valeur %u\n",i,t[i]);
  }
}
```

test-copie.c

Tableaux à plusieurs dimensions

- On peut également faire des tableaux à plusieurs dimensions

```
int tab[2][3]={{1,2,3},{4,5,6}};
```

- On crée un tableau contenant deux tableaux chacun de taille 3
- `tab[0][1]` permet d'accéder au deuxième élément du premier tableau (ici vaut 2)
- Ce n'est pas comme en java, on ne va pas pouvoir créer un tableau contenant des tableaux de différentes tailles (du moins pas de cette façon)
- Autre initialisation possible :

```
int tab[2][3]={[0][0]=1,[0][1]=2,[0][2]=3,[1][0]=4,[1][1]=5,[1][2]=6};
```

Les structures

- Une structure est un moyen de regrouper des données de types différents sous un même nom. Syntaxe:

```
struct modele {
        type1 membre1 ;
        type2 membre2 ;
        ...
        typen membre n ;
} ;
```

- Une telle structure a n membres
- Pour définir un objet de ce nouveau type :
    - **struct modele obj ;**

Exemple d'une structure pour un point avec deux coordonnées :

```
struct point{
        int x;
        int y;
        };
```

- On peut ensuite initialiser de la façon suivante **à la déclaration** :

```
struct point p1={.x=-2,.y=3};
```

Exemple suite

On n'est pas obligé de remplir tous les champs qui seront mis à 0.

- On accède aux différentes champs de la structure que l'on peut lire ou écrire en faisant par exemple :

```
p1.x=12;
printf("%d\n",p1.u);
```

Structures vs tableaux

- Les tableaux contiennent des données de même type
- On ne peut pas affecter un tableau à un autre tableau, on peut le faire avec une structure

Structures vs tableaux

```
struct point p1={.x=-2,.y=3};
struct point p2=p1 ;
```

- **Attention : la structure dans ce cas est copiée !!!**
- **On ne peut pas tester l'égalité ou la différence** de structures
- On peut passer des structures en arguments de fonctions, mais dans ce cas **le passage se fait par valeur, la structure est copiée !**
- **Une fonction peut renvoyer une structure**

```c
struct point{
          int x;
         int y; };
struct point f(struct point q){
         q.x=100;
         q.y=100;
         return q; }
int main(void){
  struct point p1={.x=1,.y=1,};
  struct point p2=p1;
  p2.x=10;
  p2.y=10;
  struct point p3=f(p1);
  printf("p1.x :%d, p2.x: %d, p3.x: %d\n",p1.x,p2.x,p3.x);
  return EXIT_SUCCESS;
}
```

test-point.c

Structures plus complexes

- On peut mettre des tableaux (de taille fixe) dans des structures

```
struct pointbis{
  int num;
  int coord[2];
};
```

- Dans ce cas, partout où la structure est copiée (affectation, passage en arguments) le tableau est copié aussi
- On peut faire des tableaux de structures

```
struct point tab[2]={[0]={.x=1,.y=3},[1]={.x=10,.y=11}};
```

Structure dans les structures

- On peut aussi mettre des structures dans des structures

```
struct ligne {
  struct point p1;
  struct point p2;
};
```

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

struct pointbis{
  int num;
  int coord[2];
};
struct pointbis f(struct pointbis q){
  q.coord[0]=1000;
  q.coord[1]=2000;
  return q;
}
int main(void){
  struct pointbis p1={.num=0,.coord[0]=10,.coord[1]=20};
  //ou : struct pointbis p1={.num=0,.coord={10,20}};
  struct pointbis p2=p1;
  p2.coord[0]=100;
  p2.coord[1]=200;
  struct pointbis p3=f(p1);
  printf("p1.coord[0] :%d, p2.coord[0] :%d, p3.coord[0] :%d\n",

         p1.coord[0],p2.coord[0],p3.coord[0]);
  return EXIT_SUCCESS;
}
```

test-pointbis.c

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

struct point{
  int x;
        int y; };
struct ligne{
  struct point p1;
  struct point p2;
};
int main(void){
  struct ligne l1={.p1={.x=1,.y=2},.p2={.x=10,.y=20}};
  struct point p={.x=10,.y=20};
  struct point q={.x=100,.y=200};
  struct ligne l2={.p1=p,.p2=q};
  printf("%d %d\n",l1.p1.y,l2.p2.x);
  return EXIT_SUCCESS;

}
```

test-ligne.c

Création d'alias: typedef

- On peut simplifier les écritures en utilisant un alias de type.
- Exemple : Crée un alias de type "entier" pour le type de données "int"

```c
typedef int entier;
entier x = 5;
```

Alias

- Autre exemple

```
struct point{
  int x;
        int y;
};
typedef struct point point;
```

- Ici point sera un alias de struct point
- On aurait pu mettre n'importe quoi à la place de point pour renommer struct point, mais usage de mettre le même nom.

Alias

- En plus court (mais plus sujet à erreur):

```c
typedef struct point{
  int x;
  int y;
} point ;
```

```c
struct point{
  int x;
int y; };
typedef struct point point;

point f(point q){
  q.x=100;

q.y=100;

return q; }

int main(void){
  point p1={.x=1,.y=1,};
  point p2=p1;
  p2.x=10;
  p2.y=10;
  point p3=f(p1);
  printf("p1.x :%d, p2.x: %d, p3.x: %d\n",p1.x,p2.x,p3.x);
  return EXIT_SUCCESS;

}
```

test-typedef.c

Les énumérations

- Permettent de définir un type en listant ses valeur possibles
- Syntaxe: **enum modele {constante 1,constante 2, ..., constante n} ;**
- Les constantes sont en fait des alias pour des entiers signés et qu'on utilise de 0 à n-1

```
enum couleur {ROUGE, BLEU, JAUNE, VERT}
```

- ROUGE vaut 0, BLEU vaut 1, JAUNE vaut 2 VERT vaut 3

Exemple :

```
enum couleur {ROUGE, BLEU, JAUNE, VERT}
```

- ROUGE vaut 0, BLEU vaut 1, JAUNE vaut 2 VERT vaut 3
- On peut ensuite les utiliser comme type (enum couleur) et ces noms comme constante
- Comme ces entiers, on peut les comparer et aussi faire des switch dessus
- On peut imposer leur valeur :

```
enum couleur {ROUGE=10, BLEU=2, JAUNE, VERT} ;
```

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

enum couleur {ROUGE, BLEU, JAUNE, VERT};

int main(void){
  enum couleur r=ROUGE;
  enum couleur j=JAUNE;
  printf("%d %d\n",r,j);//0 2
  return EXIT_SUCCESS;

}
```

test-enum.c

Remarque: la valeur associé' est 1 de plus que la valeur précédente

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

enum couleur {ROUGE=10, BLEU=2, JAUNE, VERT};

int main(void){
  enum couleur r=ROUGE;
  enum couleur j=JAUNE;
  printf("%d %d\n",r,j);//10 3
  return EXIT_SUCCESS;

}
```

test-enum2.c