UNIVERSITÉ PARIS 8
VINCENNES-SAINT-DENIS

Licence informatique & vidéoludisme
Semestre 1

Méthodologie de la programmation

Chapitre 3

La programmation en C (partie 2)

Variables et mémoire

- la mémoire d'un ordinateur est décomposée en octets
- chaque octet a une adresse qui est un entier
- quand on définit une variable, on lui attribue une adresse
- on accède à l'adresse d'une variable grâce à l'opérateur "&"
- le type de la variable définit la taille de la zone mémoire qui lui est attribuée

Types composés

A partir des types prédéfinis du C (caractères, entiers, flottants), on peut créer de nouveaux types, appelés types composés, qui permettent de représenter des ensembles de données organisées.

Types composés - tableaux

un tableau est un ensemble d'éléments **du même type** stockés de manière **contiguë** dans la mémoire

```
type nom_du_tableau[nombre_éléments];
int tab[10]; // alloue 10 x 4 octets dans la mémoire

type nom_du_tableau[N] = {constante_1,constante_2,...,constante_N};
int tab[4] = {1, 2, 3, 4};
int tab[4] = {1, 2, 3}; // le dernier élément est initialisé à 0
```

Attention : le compilateur complète toutes les chaînes de caractères par le caractère nul '\0' $\Rightarrow$ la taille du tableau = la taille de la chaîne + 1

```
char tab[8] = "exemple";
char tab[] = "exemple"; // la taille du tableau vaut 8
```

Types composés - tableaux - `sizeof`

L'opérateur `sizeof` donne la quantité de stockage, **en octets**, obligatoire pour stocker un objet du type de l'opérande.

```
1 char tab1[] = "hello";
2 int tab2[4] = {1, 2, 3, 4};
```

Que valent `sizeof(tab1)` et `sizeof(tab2)` ?

Types composés - tableaux - `sizeof`

L'opérateur `sizeof` donne la quantité de stockage, **en octets**, obligatoire pour stocker un objet du type de l'opérande.

```
1 char tab1[] = "hello";
2 int tab2[4] = {1, 2, 3, 4};
```

Que valent `sizeof(tab1)` et `sizeof(tab2)` ? **6** et **16**

Types composés - tableaux

les indices vont de 0 à nombre-éléments - 1

```
1 #define N 10 // cette directive permet de définir une constante
2 main()
3 {
4   int tab[N];
5   int i;
6   for (i = 0; i < N; i++)
7     printf("tab[%d] = %d\n",i,tab[i]);
8 }
```

Types composés - tableaux

Un tableau est en réalité l'adresse du premier élément, qui est **constante**.
On ne peut donc pas faire

```
1 tab1 = tab2
```

il faut affecter chaque valeur dans le nouveau tableau :

```
1
2 #define N 10
3 main()
4 {
5   int tab1[N], tab2[N];
6   int i;
7   ...
8   for (i = 0; i < N; i++)
9     tab1[i] = tab2[i];
10 }
```

Types composés - tableaux à plusieurs dimensions

```
type nom_du_tableau[nombre_lignes][nombre_colonnes]
```

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

main()
{
  int i, j;
  for (i = 0 ; i < M; i++)
    {
      for (j = 0; j < N; j++)
      {
        printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
      }
    }
}
```

Types composés - struct

Une structure est une suite finie d'objets de types différents.

```
1  // déclaration d'un "modèle" de structure
2  struct modele
3  {type_1 membre_1;
4    type_2 membre_2;
5      ...
6    type_n membre_n;
7  };
8
9  // déclaration d'un objet de type "modèle"
10 struct modele objet;
11
12 // accès aux membres (ou champs)
13 objet.membre_i
```

Types composés - struct

```c
#include <math.h>
struct complexe
{
  double reel;
  double imaginaire;
};
```

Types composés - enum

Un objet de type `enum` est un ensemble de valeurs numériques constantes successives :

```
1 enum modele {constante-1, constante-2,...,constante-n};
2
3 enum truc {
4     CONSTANTE1, // vaut 0 par défaut
5     CONSTANTE2, // vaut 1 par défaut
6     CONSTANTE3, // vaut 2 par défaut
7     NB_CONSTANTES // vaut 3 par défaut
8 };
```

```c
enum animal {
    CHIEN,
    CHAT,
    COCHON,
    NB_ANIMAUX
};
void crier(enum animal gerard) {
    switch(gerard) {
        case CHIEN:
            printf("ouaf ouaf!\n");
            break;
        case CHAT:
            printf("miaou!\n");
            break;
        case COCHON:
            printf("grouik!\n");
            break;
        default:
            printf("Animal inconnu!\n");
    }
}
```

la directive `typedef` permet d'alléger le code :

```
1  enum animal {
2      CHIEN,
3      CHAT,
4      COCHON,
5      NB_ANIMAUX
6  };
7  typedef enum animal = animal
8  void crier(animal gerard) {
9      switch(gerard) {
10         case CHIEN:
11             printf("ouaf ouaf!\n");
12             break;
13         case CHAT:
14             printf("miaou!\n");
15             break;
16     ...
17 }
```

1. Types composés

2. Fonctions d'entrées-sorties I/O

3. Pointeurs

printf : fonction d'impression formatée

```
1 printf("chaîne de contrôle", expression_1, ..., expression_n);
```

la "chaîne de contrôle" contient le texte à afficher et spécifications de format des expressions à remplacer : le format est donné après le signe % selon le tableau :

%d	int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	unsigned int	octale non signée
%lo	unsigned long int	octale non signée
%x	unsigned int	hexadécimale non signée
%lx	unsigned long int	hexadécimale non signée
%f	double	décimale virgule fixe
%lf	long double	décimale virgule fixe
%e	double	décimale notation exponentielle
%le	long double	décimale notation exponentielle
%g	double	décimale, représentation la plus courte parmi %f et %e
%lg	long double	décimale, représentation la plus courte parmi %lf et %le
%c	unsigned char	caractère
%s	char*	chaîne de caractères

printf : fonction d'impression formatée

```c
#include <stdio.h>
void main()
{
  int i = 23674;
  int j = -23674;
  char c = 'A';
  char *chaine = "test";
  /* %d : int (10), %u : unsigned int (10), %x : unsigned int (16) */
  printf("impression de i: \n");
  printf("%d \t %u \t %x",i,i,i); // 23674  23674   5c7a
  printf("\nimpression de j: \n");
  printf("%d \t %u \t %x",j,j,j); // -23674 4294943622  ffffa386
  printf("\nimpression de c: \n");
  printf("%c \t %d",c,c); // A  65
  printf("\nimpression de chaine: \n");
  printf("%s",chaine); // test
  printf("\n");
}
```

`scanf` : fonction de saisie

Saisir des données au clavier et les stocker aux adresses spécifiées :

```
scanf("chaîne de contrôle",argument−1,...,argument−n)
```

la "chaîne de contrôle" ne contient que la spécification du (ou des) format(s) :

```c
#include <stdio.h>
main()
{
  int i;
  printf("entrez un entier sous forme hexadecimale i = ");
  scanf("%x",&i);
  printf("i = %d\n",i);
}
```

scanf : fonction de saisie

```c
#include <stdio.h>
void main()
{
  int i = 23674;
  int j = -23674;
  char c = 'A';
  char *chaine = "test";
  /* %d : int (10), %u : unsigned int (10), %x : unsigned int (16) */
  printf("impression de i: \n");
  printf("%d \t %u \t %x",i,i,i); // 23674  23674   5c7a
  printf("\nimpression de j: \n");
  printf("%d \t %u \t %x",j,j,j); // -23674 4294943622  ffffa386
  printf("\nimpression de c: \n");
  printf("%c \t %d",c,c); // A  65
  printf("\nimpression de chaine: \n");
  printf("%s",chaine); // test
  printf("\n");
}
```

1. Types composés

2. Fonctions d'entrées-sorties I/O

3. Pointeurs

Pointeurs : besoin

Toute variable est stockée dans la mémoire, à une adresse précise. On désigne (très) souvent ces variables par des identificateurs, plus lisibles que les adresses.

Parfois, il est pourtant utile de manipuler les adresses directement.

```
1 int i, j;
2 i = 3;
3 j = i;
```

conduit à la situation :

objet	adresse	valeur
i	4831836000	3
j	4831836004	3

L'affectation opère sur les *valeurs* des variables, &i n'est pas modifiée.

Pointeurs : définition

Un objet de type pointeur est défini par :

- une adresse
- un type : utile pour indiquer combien de bits lire lorsqu'on souhaite manipuler l'objet pointé

```
1 type *p; // déclaration d'un pointeur p
```

une fois p défini :

- l'opérateur * permet d'accéder à la valeur pointée, c'est le déréférencement ;
- l'opérateur & permet de récupérer l'adresse mémoire d'une variable ;
- le format %p de `printf` permet d'afficher une adresse mémoire en hexadécimal.

Pointeur sans type

on peut définir un pointeur sans type :

```
1 void *ptr;
```

Dans ce cas, `ptr` ne sert qu'à stocker une adresse mémoire, mais on ne pourra pas utiliser l'opérateur * puisqu'on n'indique pas au compilateur sur combien de bits est codé l'objet pointé.

Pointeurs : bonne pratique

Il est recommandé de toujours initialiser un pointeur, même à NULL, afin de pouvoir tester sa valeur avant de l'utiliser :

```
1 int32_t i = -12;
2 int32_t *p_init = &i;    /* pointe sur l'adresse de i */
3 int32_t *p_ok = NULL;    /* vaut 0x0 */
4 int32_t *p_danger;       /* a une valeur indéterminée ! */
```

Si `p_ok == NULL`, alors on ne cherche pas à l'utiliser !

Pointeurs : un peu de gymnastique

À votre avis, que fait :

```
1 main()
2 {
3   int i = 3, j = 6;
4   int *p1, *p2;
5   p1 = &i;
6   p2 = &j;
7   *p1 = *p2;
8 }
```

Pointeurs : un peu de gymnastique

À votre avis, que fait :

```
1 main()
2 {
3   int i = 3, j = 6;
4   int *p1, *p2;
5   p1 = &i;
6   p2 = &j;
7   p1 = p2; // c'est ici que ça a changé !
8 }
```

Le passage de paramètres **par valeur**

Exemple de chose à ne pas faire :

```
1 void echange_faux(uint32_t a, uint32_t b)
2 {
3     uint32_t t = a;
4     a = b;
5     b = t;
6 }
```

Le passage de paramètres **par valeur**

Exemple de chose à ne pas faire :

```
1 void echange_faux(uint32_t a, uint32_t b){
2
3         uint32_t t = a; // t reçoit la valeur de la copie de a
4         a = b; // la copie de a reçoit la valeur de la copie de b
5         b = t; // la copie de b reçoit la valeur de t
6
7 }
```

À l'appel de la fonction, de nouveaux espaces mémoire sont alloués pour les paramètres.
On y stocke les valeurs des paramètres mais ce sont des COPIES.

Le passage de paramètres **par valeur**

Exemple de chose à ne pas faire :

```
1 void echange_faux(uint32_t a, uint32_t b){
2
3         uint32_t t = a; // t reçoit la valeur de la copie de a
4         a = b; // la copie de a reçoit la valeur de la copie de b
5         b = t; // la copie de b reçoit la valeur de t
6
7 }
```

À l'appel de la fonction, de nouveaux espaces mémoire sont alloués pour les paramètres.
On y stocke les valeurs des paramètres mais ce sont des COPIES.

Comment s'en assurer (`echange_faux.c`) ?

Le passage de paramètres **par valeur**

Exemple de chose **à faire** :

```
1 void echange(uint32_t *a, uint32_t *b)
2 {
3
4     uint32_t t = *a; // t reçoit la valeur pointée par la copie de a
5     *a = *b; // la valeur pointée par la copie de a reçoit la valeur
          pointée par la copie de b
6     *b = t; // la valeur pointée par la copie de a reçoit t
7
8 }
```

Attention aux confusions :

- on déclare un pointeur en utilisant la notation $*p$
- on accède à l'adresse correspondante grâce à p
- on accède à la valeur correspondance grâce à $*p$

Sources

- Cours de Pablo Rauzy (lien)
- Cours de Jean-Pascal Palus (lien)
- Cours d'Anne Canteaut