Introduction au Langage C

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence ESSIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus
E-mail : vincent.vidal@univ-lyon1.fr

Supports de cours et TPs : `http://claroline connect.univ-lyon1.fr` espace d'activités "M1102 M1103 C - Introduction au langage C"

46H prévues $\approx$ 42H de cours+TPs, 2H - interros, et 2H - examen final

Évaluation : Contrôle continu + examen final + Bonus/Malus TP

Plan

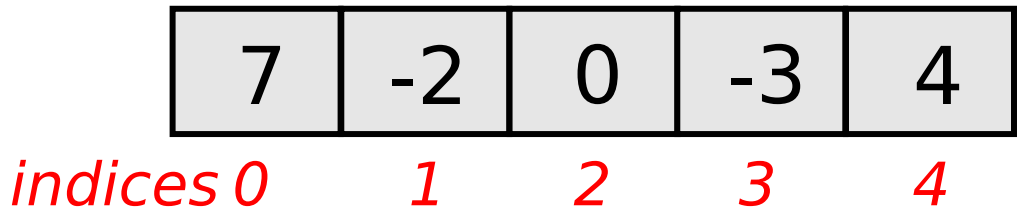
Plan

Définition

Tableau

Un tableau est une structure de données (type structuré) qui désigne un **ensemble fini et contiguë d'éléments de même type** ; chaque élément est repéré par un indice précisant sa position au sein de l'ensemble.

tableau tab de 5 entiers



tab[1] vaut -2

Intérêts des tableaux (1/2)

- **Accès en temps constant à chaque élément** du tableau.
- **Déclaration compacte** : si on veut "traiter" un nombre relativement important de données de même type ($n > 3$), on ne va pas déclarer à la main n variables scalaires différentes !
- **Traiter "à la chaîne" des données de même type** à l'aide d'une boucle qui itère sur les éléments du tableau.
- **Traiter un nombre variable de données** : si la taille du tableau peut changer, il suffit d'adapter la taille du tableau et de fixer une fois pour toute une taille maximale qui ne sera jamais dépassée.

Intérêts des tableaux (1/2)

- **Accès en temps constant à chaque élément** du tableau.
- **Déclaration compacte** : si on veut "traiter" un nombre relativement important de données de même type ($n > 3$), on ne va pas déclarer à la main n variables scalaires différentes !
- **Traiter "à la chaîne" des données de même type** à l'aide d'une boucle qui itère sur les éléments du tableau.
- **Traiter un nombre variable de données** : si la taille du tableau peut changer, il suffit d'adapter la taille du tableau et de fixer une fois pour toute une taille maximale qui ne sera jamais dépassée.

Intérêts des tableaux (1/2)

- **Accès en temps constant à chaque élément** du tableau.
- **Déclaration compacte** : si on veut "traiter" un nombre relativement important de données de même type ($n > 3$), on ne va pas déclarer à la main n variables scalaires différentes !
- **Traiter "à la chaîne" des données de même type** à l'aide d'une boucle qui itère sur les éléments du tableau.
- **Traiter un nombre variable de données** : si la taille du tableau peut changer, il suffit d'adapter la taille du tableau et de fixer une fois pour toute une taille maximale qui ne sera jamais dépassée.

Intérêts des tableaux (1/2)

- **Accès en temps constant à chaque élément** du tableau.
- **Déclaration compacte** : si on veut "traiter" un nombre relativement important de données de même type ($n > 3$), on ne va pas déclarer à la main n variables scalaires différentes !
- **Traiter "à la chaîne" des données de même type** à l'aide d'une boucle qui itère sur les éléments du tableau.
- **Traiter un nombre variable de données** : si la taille du tableau peut changer, il suffit d'adapter la taille du tableau et de fixer une fois pour toute une taille maximale qui ne sera jamais dépassée.

Intérêts des tableaux (2/2)

Applications : saisie de plusieurs valeurs, recherche du min, du max, du nombre de valeurs vérifiant certaines propriétés, tri par ordre croissant ou décroissant, représentation d'un vecteur nD sur un tableau 1D à n cases, représentation d'une matrice mxn sur un tableau 2D à m lignes et n colonnes...

Déclaration

- **Syntaxe *sans initialisation*** :
  <type> <identificateur> [*<exp-const>*] ;
- **Syntaxe *avec initialisation*** :
  <type> <identificateur> [*<exp-const>*] = {*<exp1>*,...,*<expn>*} ;
  <type> <identificateur> [] = {*<exp1>*,...,*<expn>*} ;

type est différent de **void**. *exp-const* est une **expression constante entière** ≥ 0 **calculable par le compilateur**.
Les expressions *expi* sont des expressions (constantes calculables par compilateur) de type *type*. **Il peut y avoir un nombre** n **d'expression *expi* plus petit que la taille du tableau définie par *exp-const*** : dans ce cas en C ANSI, seules les n premières cases du tableau sont initialisées (sauf si statique).

Déclaration

- **Syntaxe *sans initialisation*** :
  <type> <identificateur> [*<exp-const>*] ;
- **Syntaxe *avec initialisation*** :
  <type> <identificateur> [*<exp-const>*] = {*<exp1>*,...,*<expn>*} ;
  <type> <identificateur> [] = {*<exp1>*,...,*<expn>*} ;

type est différent de **void**. *exp-const* est une **expression constante entière** ≥ 0 **calculable par le compilateur**.

Les expressions *expi* sont des expressions (constantes calculables par compilateur) de type *type*. **Il peut y avoir un nombre** n **d'expression *expi* plus petit que la taille du tableau définie par *exp-const*** : dans ce cas en C ANSI, seules les n premières cases du tableau sont initialisées (sauf si statique).

Déclaration

- **Syntaxe *sans initialisation*** :
  <type> <identificateur> [*<exp-const>*] ;
- **Syntaxe *avec initialisation*** :
  <type> <identificateur> [*<exp-const>*] = {*<exp1>*,...,*<expn>*} ;
  <type> <identificateur> [] = {*<exp1>*,...,*<expn>*} ;

type est différent de **void**. *exp-const* est une **expression constante entière** ≥ 0 **calculable par le compilateur**.
Les expressions *expi* sont des expressions (constantes calculables par compilateur) de type *type*. **Il peut y avoir un nombre** n **d'expression *expi* plus petit que la taille du tableau définie par *exp-const*** : dans ce cas en C ANSI, seules les n premières cases du tableau sont initialisées (sauf si statique).

Expression constante calculable par le compilateur ?

```
#define NB 5
int main(void) {
   const int M = 8;
   char tab0[7] ;        /* OK : 7 est une expression constante */
   double tab1[NB] ;     /* OK : NB est une expression constante
                            calculable par le compilateur */
   int tab2[2*NB - 3] ; /* OK : 2*NB - 3 est une expression constante
                            calculable par le compilateur */
   float tab3[M] ;       /* ERREUR : M n'est pas une expression
                            calculable par le compilateur,
                            car elle fait référence à une
                            variable */

   ...
   return 0;
}
```

Quelques règles pour initialiser les tableaux 1D (1/2)

```
#define NB_CASE 5
int main(void) {
   float t[3] = {1.0, 2.0, 3.0};
   int tab[NB_CASE]; /* aucun elmt du tableau n'est init */
   float tab1 [] = {1.3f, -4.6f}; /* la taille de tab1 est 2 */
   const char voyelles[] = {'a', 'e', 'i', 'o', 'u', 'y'}; /* tableau constant de taille 6 :
                                                              la valeur des cases ne peut plus
                                                              être modifiée */

   static int tab2[NB_CASE]; /* les elmts sont init à 0 */
   int tab3[NB_CASE] = {0};  /* C99 : les elmts sont init à 0 */
   int tab4[NB_CASE] = {8};  /* C99 : le 1er à 8, les autres à 0 */
   double tab5[NB_CASE] = {8.0,7.0,5.0}; /* C99 : 8,7,5, puis que des 0 */
   int tab6[NB_CASE] = {[0 ... NB_CASE-1] = 12}; /* tous les elmts de tab6 sont init à 12 (GCC)*/

   return 0;
}
```

Quelques règles pour initialiser les tableaux 1D (2/2)

Une règle issue de la norme C99

Si les éléments entre accolades ne sont pas assez nombreux pour remplir tout le type agrégé (champs d'une structure, cases d'un tableau), alors les champs restants sont initialisés comme si les variables étaient statiques et sont donc à NULL (0 pour des entiers ou des flottants).

Accès aux éléments d'un tableau

Syntaxe : <*identificateur*> [<*indice*>]

- *indice* est une expression arithmétique entière, ainsi $tab[2 * i - j]$ est valide sous réserve que i et j soient des entiers (e.g. variables de type int) et que $0 \leq 2 * i - j$ <taille du tableau.
- Si n représente la taille d'un tableau tab, alors les indices du tableau s'étendent de 0 à $n - 1$. Son 1^{er} élément est $tab[0]$, et son dernier élément est $tab[n - 1]$.
  Attention, $tab[entier < 0]$ ou $tab[entier >= n]$ engendreront des erreurs de débordement de tableau !
- Plus généralement, le i^{eme} élément d'un tableau tab est $tab[i - 1]$.

Accès aux éléments d'un tableau

Syntaxe : <*identificateur*> [<*indice*>]

- *indice* est une expression arithmétique entière, ainsi $tab[2 * i - j]$ est valide sous réserve que i et j soient des entiers (e.g. variables de type int) et que $0 \leq 2 * i - j$ <taille du tableau.
- Si n représente la taille d'un tableau tab, alors les indices du tableau s'étendent de 0 à $n - 1$. Son 1^{er} élément est $tab[0]$, et son dernier élément est $tab[n - 1]$.
  Attention, $tab[entier < 0]$ ou $tab[entier >= n]$ engendreront des erreurs de débordement de tableau !
- Plus généralement, le i^{eme} élément d'un tableau tab est $tab[i - 1]$.

Accès aux éléments d'un tableau

Syntaxe : <*identificateur*> [<*indice*>]

- *indice* est une expression arithmétique entière, ainsi $tab[2 * i - j]$ est valide sous réserve que i et j soient des entiers (e.g. variables de type int) et que $0 \leq 2 * i - j$ <taille du tableau.
- Si n représente la taille d'un tableau tab, alors les indices du tableau s'étendent de 0 à $n - 1$. Son 1^{er} élément est $tab[0]$, et son dernier élément est $tab[n - 1]$.
  Attention, $tab[entier < 0]$ ou $tab[entier >= n]$ engendreront des erreurs de débordement de tableau !
- Plus généralement, le i^{eme} élément d'un tableau tab est $tab[i-1]$.

Exemple

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

int main(void) {
   double notes[5] ; /* tableau notes : 5 éléments contiguës de type double sont réservés */
   int i;
   double moyenne=0.0;
   unsigned int cpt_rattrapage=0;
   notes[0]=0.0; /* modification de la valeur de la 1ère case du tableau notes */
   for(i=0; i<5; i++)
   {
      printf("\nDonner la note de l'etudiant %d :", i);
      scanf("%lf", &notes[i]); /* modification de la (i+1)ème case du tableau notes */
      if( notes[i] < 10 ) cpt_rattrapage++;
   }
   for(i=0; i<5; i++) moyenne=(i*moyenne+notes[i])/(i+1);
   printf("La moyenne de la classe est %f\n", moyenne);
   printf("Le rattrapage concerne %d etudiant(s)\n", cpt_rattrapage);
   system("PAUSE");
   return 0;
}
```

Exemple utilisant une définition de symbole

```c
#include <stdio.h>
#include <stdlib.h>
#define NB_CASE 5 /* Seulement cette ligne à modifier si la taille du tableau change */
int main(void) {
   double notes[NB_CASE] ; /* tableau notes : NB_CASE éléments contiguës de type double sont
                              réservés */
   int i;
   double moyenne=0.0;
   unsigned int cpt_rattrapage=0;
   for(i=0; i<NB_CASE; i++)
   {
      printf("\nDonner la note de l'etudiant %d :", i);
      scanf("%lf", &notes[i]);
      if( notes[i] < 10 ) cpt_rattrapage++;
   }
   for(i=0; i<NB_CASE; i++) moyenne = (i*moyenne+notes[i])/(i+1);
   printf("La moyenne de la classe est %f\n", moyenne);
   printf("Le rattrapage concerne %d etudiant(s)\n", cpt_rattrapage);
   system("PAUSE");
   return 0;
}
```

Quelques règles

<*identificateur*> [<*indice*>] a un comportement similaire à celui d'une variable (*lvalue*) :

- Un élément d'un tableau peut être modifié par affectation, $tab[i] = 5$ mettra la valeur 5 dans la $(i+1)^{eme}$ case du tableau.
- Pour un tableau tab d'entiers, on peut utiliser les opérateurs d'incrémentation et de décrémentation sur un élément du tableau : $tab[i]++$, $++tab[i]$, $tab[i]--$, et $--tab[i]$.

Quelques règles

<*identificateur*> [<*indice*>] a un comportement similaire à celui d'une variable (*lvalue*) :

- Un élément d'un tableau peut être modifié par affectation, $tab[i] = 5$ mettra la valeur 5 dans la $(i+1)^{eme}$ case du tableau.
- Pour un tableau tab d'entiers, on peut utiliser les opérateurs d'incrémentation et de décrémentation sur un élément du tableau : $tab[i]++$, $++tab[i]$, $tab[i]--$, et $--tab[i]$.

Copier un tableau dans un autre

Problème : On a deux tableaux 1D, $t1$ de taille n et $t2$ de taille m tels que $n <= m$. On souhaite copier $t1$ au début de $t2$. Comment faire ?

Copier un tableau dans un autre

Problème : On a deux tableaux 1D, $t1$ de taille n et $t2$ de taille m tels que $n <= m$. On souhaite copier $t1$ au début de $t2$. Comment faire ?
Surtout pas de t2=t1 (même si n==m).

Copier un tableau dans un autre

Problème : On a deux tableaux 1D, $t1$ de taille n et $t2$ de taille m tels que $n <= m$. On souhaite copier $t1$ au début de $t2$. Comment faire ?
Surtout pas de t2=t1 (même si n==m). Et à la fin de $t2$ ($t2[m - 1] = t[n - 1]$ etc.) ?

Déclaration d'un tableau nD

Le C autorise les tableaux à plusieurs indices ou plusieurs dimensions. La déclaration suit la même logique en nD qu'en 1D.

Syntaxe sans initialisation :

<type> <identificateur> [*<exp-const1>*]...[*<exp-constn>*] ;

Les éléments d'un tableau nD sont rangés en mémoire de manière contiguë *en faisant varier l'indice le plus à droite en 1er*.

`int` tab[2][3] ; déclare un tableau 2D de $2 \times 3 = 6$ cases ; en mémoire cela donne "tab[0][0] tab[0][1] tab[0][2] tab[1][0] tab[1][1] tab[1][2]". Ainsi un accès à la case tab2D[i][j] est équivalent en "agencement mémoire" à la case tab1D[i*NB_C+j].

Déclaration d'un tableau nD

Le C autorise les tableaux à plusieurs indices ou plusieurs dimensions. La déclaration suit la même logique en nD qu'en 1D.
Syntaxe sans initialisation :
<*type*> <*identificateur*> [<*exp-const1*>]...[<*exp-constn*>] ;

Les éléments d'un tableau nD sont rangés en mémoire de manière contiguë *en faisant varier l'indice le plus à droite en 1er*.

`int tab[2][3];` déclare un tableau 2D de $2 \times 3 = 6$ cases ; en mémoire cela donne "tab[0][0] tab[0][1] tab[0][2] tab[1][0] tab[1][1] tab[1][2]". Ainsi un accès à la case tab2D[i][j] est équivalent en "agencement mémoire" à la case tab1D[i*NB_C+j].

Déclaration d'un tableau nD

Le C autorise les tableaux à plusieurs indices ou plusieurs dimensions. La déclaration suit la même logique en nD qu'en 1D.
Syntaxe sans initialisation :
<type> <identificateur> [*<exp-const1>*]...[*<exp-constn>*] ;

Les éléments d'un tableau nD sont rangés en mémoire de manière contiguë *en faisant varier l'indice le plus à droite en 1er*.
int tab[2][3] ; déclare un tableau 2D de $2 \times 3 = 6$ cases ; en mémoire cela donne "tab[0][0] tab[0][1] tab[0][2] tab[1][0] tab[1][1] tab[1][2]". Ainsi un accès à la case tab2D[i][j] est équivalent en "agencement mémoire" à la case tab1D[i*NB_C+j].

Quelques règles pour initialiser les tableaux nD

```c
#define NB_L 2 /* nombre de ligne(s) */
#define NB_C 3 /* nombre de colonne(s) */
int main(void) {
   double t[NB_L][NB_C];         /* aucune initialisation */
   static float t_[NB_L][NB_C]; /* tous les elmts init à 0 */

   /* un tableau 2D 2x3 = 2 tableaux 1D de 3 cases */
   int tab[NB_L][NB_C] = { {1,2,3} , {4,5,6} };
   int tab_[NB_L][NB_C] = { {1} , {4,5} }; /* valeurs omises et complétées par des 0 en C99 */

   /* init fonction de l'agencement mémoire */
   int tab2[NB_L][NB_C] = {1, 2, 3, 4, 5, 6};
   int tab2_[NB_L][NB_C] = {1, 2, 3, 4};   /* valeurs omises et complétées par des 0 en C99 */

   return 0;
}
```

A vous de jouer (1/2)

- Déclarer et initialiser un tableau 2D 4x2 de int.
- Déclarer et initialiser un tableau 3D 3x4x2 de int.

A vous de jouer (2/2)

```
int tab[3][4][2] = { { {1,2},{3,4},{5,6},{7,8} },
                     { {9,10},{11,12},{13,14},{15,16} },
                     { {17,18},{19,20},{21,22},{23,24} }
    };
```

Parcours d'un tableau 2D via 2 for

```c
#define NB_L 2 /* nombre de ligne(s) */
#define NB_C 3 /* nombre de colonne(s) */
int main(void) {
   int tab2D[NB_L][NB_C] ;
   int l, c;

   for(l=0; l<NB_L; l++)
      for(c=0; c<NB_C; c++)
         tab2D[l][c] = 6 ;

   return 0;
}
```

Parcours d'un tableau 2D via 2 while

```c
#define NB_L 2 /* nombre de ligne(s) */
#define NB_C 3 /* nombre de colonne(s) */
int main(void) {
   int tab2D[NB_L][NB_C] ;
   int l, c;

   l=0;
   while(l < NB_L)
   {
      c=0;
      while(c < NB_C)
      {
         tab2D[l][c] = 6 ;

         c++ ;
      }

      l++;
   }
   return 0;
}
```

Plan

Définitions

Adresse

Une adresse est un entier positif spécifiant l'indice d'un octet dans la mémoire centrale de l'ordinateur.

Pointeur

Un pointeur est une variable destinée à contenir des *adresses* (mémoire) d'autres objets (variables, fonctions...). Il sera possible d'accéder à la valeur de la variable "pointée" voire même de la modifier.

Définitions

Adresse

Une adresse est un entier positif spécifiant l'indice d'un octet dans la mémoire centrale de l'ordinateur.

Pointeur

Un pointeur est une variable destinée à contenir des *adresses* (mémoire) d'autres objets (variables, fonctions...). Il sera possible d'accéder à la valeur de la variable "pointée" voire même de la modifier.

Déclaration

Syntaxe :

- **Pointeur normal** : <*type*> * <*identificateur[= adresse_init]*> ;
- **Pointeur générique** : `void` * <*identificateur[= adresse_init]*> ;

type est un type quelconque différent de `void`, en particulier *type* peut être un type pointeur.
`void` * (C ANSI) désigne un pointeur sur un objet de type quelconque, et il n'a pas de type associé.

```
int * pt; /* pointeur vers une lvalue de type "int" */
double ** pt2; /* pointeur vers une lvalue de type
                  "double *" */
```

Déclaration

Syntaxe :

- **Pointeur normal** : <*type*> * <*identificateur[= adresse_init]*>;
- **Pointeur générique** : **void** * <*identificateur[= adresse_init]*>;

type est un type quelconque différent de **void**, en particulier *type* peut être un type pointeur.
void * (C ANSI) désigne un pointeur sur un objet de type quelconque, et il n'a pas de type associé.

```
int * pt; /* pointeur vers une lvalue de type "int" */
double ** pt2; /* pointeur vers une lvalue de type
                  "double *" */
```

Les opérateurs & et *

&, l'**opérateur unaire** *adresse de*, a été introduit avec la fonction scanf. **&** n'a de sens qu'à gauche d'une *lvalue*.

* est un **opérateur unaire** *contenu de*, appelé opérateur de déréférencement. * n'a de sens qu'à gauche d'une adresse (valide) d'objet. **Il permet d'accéder au contenu d'un objet référencé par une adresse**. En particulier l'opérateur * permet d'accéder au contenu pointé par un pointeur.

Attention : * ne peut pas s'appliquer directement sur un pointeur générique `void *`, car il a besoin de connaître le type concret de l'objet à lire.

Les opérateurs & et *

&, l'**opérateur unaire** *adresse de*, a été introduit avec la fonction scanf. & n'a de sens qu'à gauche d'une *lvalue*.

* est un **opérateur unaire** *contenu de*, appelé opérateur de déréférencement. * n'a de sens qu'à gauche d'une adresse (valide) d'objet. **Il permet d'accéder au contenu d'un objet référencé par une adresse**. En particulier l'opérateur * permet d'accéder au contenu pointé par un pointeur.

Attention : * ne peut pas s'appliquer directement sur un pointeur générique `void *`, car il a besoin de connaître le type concret de l'objet à lire.

Les opérateurs & et *

&, l'**opérateur unaire** *adresse de*, a été introduit avec la fonction scanf. & n'a de sens qu'à gauche d'une *lvalue*.

* est un **opérateur unaire** *contenu de*, appelé opérateur de déréférencement. * n'a de sens qu'à gauche d'une adresse (valide) d'objet. **Il permet d'accéder au contenu d'un objet référencé par une adresse**. En particulier l'opérateur * permet d'accéder au contenu pointé par un pointeur.

Attention : * ne peut pas s'appliquer directement sur un pointeur générique **void** *, car il a besoin de connaître le type concret de l'objet à lire.

Exemple

```c
#include <stdio.h>
int main(void) {
   int a=6;
   int* pta=&a;          /* init pointeur vers a */
   int** pt_pta=&pta; /* init pointeur vers pta */
   printf("%d  %d  %d\n", a, *pta, **pt_pta);
   a=7;
   printf("%d  %d  %d\n", a, *pta, **pt_pta);
   *pta = 8;      /* *pta est une lvalue */
   printf("%d  %d  %d\n", a, *pta, **pt_pta);
   **pt_pta = 9; /* **pt_pta est une lvalue */
   printf("%d  %d  %d\n", a, *pta, **pt_pta);
   return 0;
}
```

Précisions

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
   int i=1;
   int *ad, ad2;   /* ad2 est de type int */
   int *ad3, *ad4; /* ad4 est de type int* */

   ad=&i;    /* ad est une lvalue */
   (*ad)++;  /* même rôle que i++; *ad est une lvalue */
   *ad += 5; /* même rôle que i += 5; */

   printf("%d  %d\n", i, *ad);

   system("PAUSE");

   *ad4 = 4; /* erreur dynamique, car ad4 ne pointe sur rien */
   (&i)++;   /* erreur statique car &i est constante (pas lvalue) */
   ad++;     /* autorisé, mais inutile ici (cf. arithmétique des pointeurs) */

   return 0;
}
```

Solution à notre problème d'échange (1/2)

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   void echange (int * ad1, int * ad2) ;
   int n=10, p=20 ;
   printf("avant appel : %d %d\n", n, p) ;

   echange(&n, &p) ; /* les arguments effectifs sont les @ des var */

   printf("après appel : %d %d\n", n, p) ;
   system("PAUSE") ;
   return 0 ;
}
void echange (int * ad1, int * ad2) /* @ modifiable pointant sur un entier modifiable */
{
  int tmp ;
  tmp = * ad1 ;   /* l'entier pointé par ad1 est lu et sa valeur affectée à tmp */
  * ad1 = * ad2 ; /* l'entier pointé par ad1 prend la valeur de l'entier pointé par ad2 */
  * ad2 = tmp ;   /* l'entier pointé par ad2 prend la valeur de tmp */
}
```

Solution à notre problème d'échange (2/2)

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   void echange (int * const ad1, int * const ad2) ;
   int n=10, p=20 ;
   printf("avant appel : %d %d\n", n, p) ;
   echange(&n, &p) ; /* les arguments effectifs sont les @ des var */
   printf("après appel : %d %d\n", n, p) ;
   system("PAUSE") ;
   return 0 ;
}
void echange (int * const ad1, int * const ad2) /* @ constante pointant sur un int modifiable */
{
  int tmp ;
  tmp = * ad1 ;
  * ad1 = * ad2 ;
  * ad2 = tmp ;
}
/* remarque : le cas d'une @ modifiable pointant sur un int constant : const int * ad */
```

Qualificatif **const** dans les arguments d'une fonction

Lorsqu'une fonction a des arguments de type donnée/entrée, et qu'on utilise un pointeur pour accéder à la lvalue pointée, il faut utiliser le qualificatif **const** pour interdir la modification de l'objet pointé dans la fonction : **const** <*type*>* <*argument*>

Arithmétique des adresses/pointeurs (1/2)

Si *ad* contient l'adresse d'un type **int** (**int** * ad ;), alors $ad + 1$ est l'adresse de l'entier suivant en mémoire. Et $ad + k$ désignera l'adresse du k-ième élément suivant (si k >0, sinon précédent) de type **int** en mémoire.

On peut généraliser cela à toute adresse non générique ($\neq$ **void***) : **si** *ad* est l'adresse d'un objet de type *type* **alors** $ad + k$ est l'adresse d'un objet de type *type* situé à l'adresse *ad*+k***sizeof**(*type*).

De même, si *ad* contient l'adresse d'un type *type* (type * ad ;), alors *ad++* incrémente l'adresse contenue dans *ad* afin qu'elle désigne l'objet suivant. En déduire ce qui se passe pour la décrémentation.

Arithmétique des adresses/pointeurs (1/2)

Si *ad* contient l'adresse d'un type **int** (**int** * ad ;), alors $ad + 1$ est l'adresse de l'entier suivant en mémoire. Et $ad + k$ désignera l'adresse du k-ième élément suivant (si k >0, sinon précédent) de type **int** en mémoire.

On peut généraliser cela à toute adresse non générique (≠ **void***) : **si** *ad* est l'adresse d'un objet de type *type* **alors** $ad + k$ est l'adresse d'un objet de type *type* situé à l'adresse *ad*+k***sizeof**(*type*).

De même, si *ad* contient l'adresse d'un type *type* (type * ad ;), alors *ad++* incrémente l'adresse contenue dans *ad* afin qu'elle désigne l'objet suivant. En déduire ce qui se passe pour la décrémentation.

Arithmétique des adresses/pointeurs (1/2)

Si *ad* contient l'adresse d'un type **int** (**int** * ad ;), alors $ad + 1$ est l'adresse de l'entier suivant en mémoire. Et $ad + k$ désignera l'adresse du k-ième élément suivant (si k >0, sinon précédent) de type **int** en mémoire.

On peut généraliser cela à toute adresse non générique (≠ **void***) : **si** *ad* est l'adresse d'un objet de type *type* **alors** $ad + k$ est l'adresse d'un objet de type *type* situé à l'adresse *ad*+k***sizeof**(*type*).

De même, si *ad* contient l'adresse d'un type *type* (type * ad ;), alors *ad++* incrémente l'adresse contenue dans ad afin qu'elle désigne l'objet suivant. En déduire ce qui se passe pour la décrémentation.

Arithmétique des adresses/pointeurs (2/2)

Comment accéder à l'objet (la lvalue) pointé par $ad + k$?

```
int tab[10] = {1,2,3,4,5,6,7,8,9,10};
int * ad = &tab[5];
printf("%d  %d\n", ad[-2], *(ad - 2) );
```

Arithmétique des adresses/pointeurs (2/2)

Comment accéder à l'objet (la lvalue) pointé par $ad + k$?
2 notations équivalentes en C :

- *(*ad*+k)
- *ad*[k]

Mais attention **il faut rester dans la zone allouée** (sinon il y a un risque de plantage)...

```
int tab[10] = {1,2,3,4,5,6,7,8,9,10};
int * ad = &tab[5];
printf("%d  %d\n", ad[-2], *(ad - 2) );
```

Les tableaux 1D et les pointeurs (1/2)

- Un identificateur de tableau "alloué par le compilateur" est une *constante pointeur* : c'est l'adresse de son 1er élément. En particulier, il n'est pas possible d'utiliser un opérateur d'in/décrémentation sur un tel identificateur de tableau.
- `int t[10];` la notation `t` est équivalente à la notation `&t[0]` ; `t[0]` est équivalent à `*t`.
- `t+k` est équivalent à `&t[k]`
- `t[k]` est équivalent à `* (t+k)`

Les tableaux 1D et les pointeurs (1/2)

- Un identificateur de tableau "alloué par le compilateur" est une *constante pointeur* : c'est l'adresse de son 1er élément. En particulier, il n'est pas possible d'utiliser un opérateur d'in/décrémentation sur un tel identificateur de tableau.
- **int** t[10] ; la notation t est équivalente à la notation **&t[0]** ; t[0] est équivalent à *t.
- t+k est équivalent à &t[k]
- t[k] est équivalent à * (t+k)

Les tableaux 1D et les pointeurs (1/2)

- Un identificateur de tableau "alloué par le compilateur" est une *constante pointeur* : c'est l'adresse de son 1er élément. En particulier, il n'est pas possible d'utiliser un opérateur d'in/décrémentation sur un tel identificateur de tableau.
- **int** t[10] ; la notation t est équivalente à la notation **&t[0]** ; t[0] est équivalent à *t.
- t+k est équivalent à &t[k]
- t[k] est équivalent à * (t+k)

Les tableaux 1D et les pointeurs (1/2)

- Un identificateur de tableau "alloué par le compilateur" est une *constante pointeur* : c'est l'adresse de son 1er élément. En particulier, il n'est pas possible d'utiliser un opérateur d'in/décrémentation sur un tel identificateur de tableau.
- **int** t[10] ; la notation t est équivalente à la notation **&t[0]** ; t[0] est équivalent à *t.
- t+k est équivalent à &t[k]
- t[k] est équivalent à * (t+k)

Les tableaux 1D et les pointeurs (2/2)

```
int t[100]; /* On veut init à 4 les cases de t : */
/* manière 1 */
int i;
for (i=0 ; i<100 ; i++)
   t[i] = 4 ;

/* manière 2 */
int i;
for (i=0 ; i<100 ; i++)
   *(t+i) = 4 ;

/* manière 3 */
int i;
int * pt ;
for (i=0, pt=t ; i<100 ; i++, pt++)
   *pt = 4 ;

/* manière 4 */
int * pt ;
for (pt=t ; pt<t+100 ; pt++)
   *pt = 4 ;
```

Règles

- **Initialiser un pointeur dès sa création**, soit à l'aide de l'adresse d'une lvalue (opérateur &), soit en utilisant NULL (prédéfinie dans stdio.h).
- N'appliquer un opérateur d'accès à la valeur (* ou []) seulement si le pointeur pointe sur une lvalue (ne pas le faire s'il pointe sur NULL ou une adresse mémoire arbitraire).
- Ne pas réaliser d'opération d'accès à la valeur ou d'affectation dans une zone mémoire non allouée.
- Ne comparer que des pointeurs du même type (opérateurs relationnels).
- Éviter de faire des casts sur les pointeurs (à cause des contraintes d'alignement mémoire), sauf pour convertir un pointeur générique `void*`.

Règles

- **Initialiser un pointeur dès sa création**, soit à l'aide de l'adresse d'une lvalue (opérateur &), soit en utilisant NULL (prédéfinie dans `stdio.h`).
- N'appliquer un opérateur d'accès à la valeur (* ou []) seulement si le pointeur pointe sur une lvalue (ne pas le faire s'il pointe sur NULL ou une adresse mémoire arbitraire).
- Ne pas réaliser d'opération d'accès à la valeur ou d'affectation dans une zone mémoire non allouée.
- Ne comparer que des pointeurs du même type (opérateurs relationnels).
- Éviter de faire des casts sur les pointeurs (à cause des contraintes d'alignement mémoire), sauf pour convertir un pointeur générique `void*`.

Règles

- **Initialiser un pointeur dès sa création**, soit à l'aide de l'adresse d'une lvalue (opérateur &), soit en utilisant NULL (prédéfinie dans `stdio.h`).
- N'appliquer un opérateur d'accès à la valeur (* ou []) seulement si le pointeur pointe sur une lvalue (ne pas le faire s'il pointe sur NULL ou une adresse mémoire arbitraire).
- Ne pas réaliser d'opération d'accès à la valeur ou d'affectation dans une zone mémoire non allouée.
- Ne comparer que des pointeurs du même type (opérateurs relationnels).
- Éviter de faire des casts sur les pointeurs (à cause des contraintes d'alignement mémoire), sauf pour convertir un pointeur générique `void*`.

Règles

- **Initialiser un pointeur dès sa création**, soit à l'aide de l'adresse d'une lvalue (opérateur &), soit en utilisant NULL (prédéfinie dans `stdio.h`).
- N'appliquer un opérateur d'accès à la valeur (* ou []) seulement si le pointeur pointe sur une lvalue (ne pas le faire s'il pointe sur NULL ou une adresse mémoire arbitraire).
- Ne pas réaliser d'opération d'accès à la valeur ou d'affectation dans une zone mémoire non allouée.
- Ne comparer que des pointeurs du même type (opérateurs relationnels).
- Éviter de faire des casts sur les pointeurs (à cause des contraintes d'alignement mémoire), sauf pour convertir un pointeur générique `void*`.

Règles

- **Initialiser un pointeur dès sa création**, soit à l'aide de l'adresse d'une lvalue (opérateur &), soit en utilisant NULL (prédéfinie dans `stdio.h`).
- N'appliquer un opérateur d'accès à la valeur (* ou []) seulement si le pointeur pointe sur une lvalue (ne pas le faire s'il pointe sur NULL ou une adresse mémoire arbitraire).
- Ne pas réaliser d'opération d'accès à la valeur ou d'affectation dans une zone mémoire non allouée.
- Ne comparer que des pointeurs du même type (opérateurs relationnels).
- Éviter de faire des casts sur les pointeurs (à cause des contraintes d'alignement mémoire), sauf pour convertir un pointeur générique **`void*`**.

Plan

Revenons sur les fonctions (1/2)

Lorsque l'identificateur d'un tableau est passé en argument effectif d'une fonction, c'est l'adresse du tableau (donc l'adresse du 1er élément) qui est transmise à la fonction, d'où la possibilité de modifier les éléments du tableau depuis la fonction.

Syntaxe d'un paramètre tableau 1D d'une fonction :

- Tableau de **taille fixe** :
  <type> <id-fct>(..., <type-elm> <id-tab>[<exp-const>], ...)
- Tableau de **taille variable** (notation à préférer) :
  <type> <id-fct>(..., <type-elm> <id-tab>[], int <id-taille>,...)
- Tableau de **taille variable** :
  *<type> <id-fct>(..., <type-elm> * <id-tab>, int <id-taille>,...)*

Revenons sur les fonctions (1/2)

Lorsque l'identificateur d'un tableau est passé en argument effectif d'une fonction, c'est l'adresse du tableau (donc l'adresse du 1er élément) qui est transmise à la fonction, d'où la possibilité de modifier les éléments du tableau depuis la fonction.

Syntaxe d'un paramètre tableau 1D d'une fonction :

- Tableau de **taille fixe** :
  <type> *<id-fct>*(..., *<type-elm>* *<id-tab>*[*<exp-const>*], ...)
- Tableau de **taille variable** (notation à préférer) :
  <type> *<id-fct>*(..., *<type-elm>* *<id-tab>*[], int *<id-taille>*,...)
- Tableau de **taille variable** :
  <type> *<id-fct>*(..., *<type-elm>* * *<id-tab>*, int *<id-taille>*,...)

Revenons sur les fonctions (2/2)

Comment connaître la taille d'un tableau tab de taille variable à l'intérieur de la fonction (tab est un argument) ?

- La passer en argument de la fonction.

Revenons sur les fonctions (2/2)

Comment connaître la taille d'un tableau tab de taille variable à l'intérieur de la fonction (tab est un argument) ?

- **La passer en argument de la fonction**.

Revenons sur les fonctions

Syntaxe d'un paramètre tableau nD d'une fonction :

- Tableau de **taille fixe** :

  <type> *<id-fct>*(..., *<type-elm>* *<id-tab>*[*<exp-const1>*]...[*<exp-constn>*], ...)

- Tableau de **taille fixe** (omission 1ère dim) :

  <type> *<id-fct>*(..., *<type-elm>* *<id-tab>*[][*<exp-const2>*]...[*<exp-constn>*], ...)

Tableau de taille fixe avec taille en argument

```
/* Attention : si dans les arguments, la dimension apparaît à droite du tableau ça n'est plus
correct */
void ma_fonction_tab1D(int nbe, int tab1D[nbe])
{
   [déclarations et instructions]
}

void ma_fonction_tab2D(int nb_l, int nb_c, int tab2D[nb_l][nb_c])
{
   [déclarations et instructions]
}
```