Université Savoie Mont Blanc,
UFR SceM
INFO505, L3 INFO

September 3, 2023
J.-O. Lachaud

Fiche 3 — structures, pointeurs

(structures, pointeurs, passage par adresse)

Cette fiche présente deux constructeurs de types complexes du C: les structures qui représentent des agrégats de données, et les pointeurs qui permettent de manipuler les adresses en mémoire.

1 Structures

Une structure en C (mot-clé `struct`) permet d'agréger des variables (possiblement de types différents) dans une seule entité. Une variable de type structure est alors une sorte de super-variable, qui contient autant de variables que de types qu'elle agrégèe. Le constructeur de type `struct` est récursif, c'est-à-dire qu'une structure peut agréger des structures.

1.1 Déclaration et usage

La syntaxe de déclaration d'une *structure* `nom_struct`, et d'une variable `var` de ce type structure prend la forme suivante:

```
// déclaration typiquement dans un header
struct nom_struct {
  type1 champ1;
  type2 champ2;
  ...
};
...
// définition dans un source
struct nom_struct var;
```

Les variables agrégées dans une structure s'appellent les *champs* de la structure (*field* en anglais). Les champs sont nommés pour qu'on puisse ensuite accéder aux variables agrégées via ce nom.

Par exemple, on veut représenter un joueur, avec nom, prénom, pseudo, nombre préféré:

```
struct SJoueur {
  char nom    [ 32 ];
  char prenom [ 32 ];
  char pseudo [ 16 ];
  int  number;
};
```

On peut ensuite déclarer une variable de type `struct SJoueur`, et initialiser ses champs directement avec la notation `= { <val-champ1> , <val-champ2> ,, ... }`.

```
struct SJoueur jcvd = { "Vandamme", "Jean-Claude", "Aware", 99999 };
struct SJoueur inconnu; // pas d'initialisation
// Affiche les données du joueur jcvd.
printf( "nom=%s prenom=%s pseudo=%s number=%d\n",
        jcvd.nom, jcvd.prenom, jcvd.pseudo, jcvd.number );
// Les lignes ci-dessous affichent des valeurs non déterminées.
printf( "nom=%s prenom=%s pseudo=%s number=%d\n",
        inconnu.nom, inconnu.prenom, inconnu.pseudo, inconnu.number );
```

donne le résultat suivant à l'exécution:

```
nom=Vandamme prenom=Jean-Claude pseudo=Aware number=99999
nom=BP@% prenom=pù£ pseudo= number=-292976344
```

Comme illustré dans le code ci-dessus on accède à un champ d'une variable de type structure grâce à la notation "pointée", i.e. on met un "`.`" suivi du nom du champ.

Le champ spécifié se comporte comme la variable/symbole de type correspondant. Ainsi `jcvd.number` est une variable de type `int`. De même `jcvd.nom` est un tableau de `char`, donc c'est une expression

dont la valeur est l'adresse en mémoire de la variable `jcvd.nom[0]`. Enfin `jcvd.prenom[3]` est une variable de type `char`.

Il est souvent pratique de déclarer un alias pour un type structure, afin d'éviter d'écrire `struct` à chaque utilisation de la structure. On écrirait typiquement après la déclaration de la structure `struct SJoueur` la ligne:

```
typedef struct SJoueur Joueur;
```

ce qui permet de disposer du type `Joueur` pour déclarer des variables:

```
Joueur jr = { "Ewing", "John Ross Jr", "J.R.", 666 };
```

1.2 Emboîtement de structures

Comme tout constructeur de type C (tableau, pointeur, union), il est possible de définir des structures contenant des types quelconques, par exemple d'autres structures. Les accès aux champs (i.e. variables) internes sont naturels en enchaînant les notations pointées. Ainsi, constituons un type `Equipe`:

```
struct SEquipe {
  Joueur leader;
  Joueur sidekicks[ 3 ];
};
typedef struct SEquipe Equipe;

Equipe rouge =
{ { "Stark", "Tony", "Iron Man", 1963 },
  { { "Rogers", "Steven", "Captain America", 1940 },
    { "Romanov", "Natacha", "Veuve noire", 1964 },
    { "Wilson", "Samuel", "Faucon", 1969 }
  }
};

// Accès au champ number de la variable rouge.sidekicks[ 0 ] de type Joueur
int first_apparition = rouge.sidekicks[ 0 ].number;
```

Les variables de type structure sont des variables comme les autres, on peut donc utiliser l'opérateur `=` dessus (même si elles contiennent des tableaux !).

```
Equipe verte   = rouge; // valide, attention tout est recopié
Joueur ironman = rouge.leader; // valide, attention tout est recopié
```

Les variables `verte` et `ironman` ont les mêmes valeurs respectives que `rouge` et `rouge.leader` mais sont des variables *distinctes*. Autrement dit, c'est des clones.

1.3 Passage de structures en paramètre

On peut passer des structures en paramètre à des fonctions. Le passage est *par valeur*, c'est à dire que le paramètre est un clone de l'argument donné à l'appel. Modifier le paramètre ne modifiera pas l'argument.

```
void Joueur_affiche( Joueur j )
{
  printf( "nom=%s prenom=%s pseudo=%s number=%d",
          j.nom, j.prenom, j.pseudo, j.number );
}
void Equipe_affiche( Equipe e )
{
  printf( "leader =< " );
  Joueur_affiche( e.leader );
  printf( " >\n" );
  for ( int i = 0; i < 3; i++ )
    {
      printf( "sidekick %d =< ", i );
      Joueur_affiche( e.sidekicks[ i ] );
      printf( " >\n" );
    }
}
...
Equipe_affiche( rouge );
```

NB: Attention, les structures peuvent vite occuper pas mal de place en mémoire. Par exemple `sizeof(Joueur)` donne 84 octets et `sizeof(Equipe)` donne 336 octets. Cela veut dire que 336 octets sont recopiés dans la variable locale `e` à chaque appel de la fonction `Equipe_affiche`. On préférera très souvent le passage par adresse de la structure pour minimiser le temps d'exécution.

- Les structures sont des types qui agrègent autant de types souhaités.
- Une variable de type structure est donc l'agrégat d'autant de variables des types agrégés, variables appelés champs de la structure.
- Une structure occupe toujours une taille mémoire identique (que l'on peut connaître grâce à `sizeof`)
- On accède aux champs d'une structure via la notation pointée "`.`" suivie du nom du champ.
- Passée en paramètre, une structure est passée par valeur, c'est-à-dire que le paramètre est une variable locale à la fonction qui a clonée les valeurs de la structure donnée en argument.
- On préfère donc en général passer les structures par adresse, pour minimiser le coût mémoire et le coût en temps de copier les valeurs.

Question 1. Proposez une structure `Polynome` pour représenter les polynômes de degré 2 (i.e de la forme $ax^2 + bx + c$), de façon à ce qu'on puisse par exemple définir les polynômes suivants:

```
Polynome P0 = {  0.0,  0.0,  0.0 };// polynôme nul P(x) = 0
Polynome P1 = {  1.0,  2.0,  0.0 };// droite de pente 2 P(x) = 1 + 2x
Polynome P2 = {  1.0,  0.0, -1.0 };// parabole P(x) = 1 - x^2
Polynome P3 = { -1.0, -1.0,  1.0 };// parabole P(x) = -1 - x + x^2
```

Question 2. Ecrivez les fonctions suivantes:

```
// Evalue P(x)
double Polynome_eval   ( Polynome P, double x );
// Retourne le coefficient de P(x) devant x^k, k=0,1,2
double Polynome_coef   ( Polynome P, int k );
// Retourne la plus petite racine de P(x), ou NAN sinon.
double Polynome_racine1( Polynome P );
// Retourne la plus grande racine de P(x), ou NAN sinon.
double Polynome_racine2( Polynome P );
```

NB: pour tester si un `double t` vaut `NAN`, vous pouvez utiliser `isnan( t )`, présent dans le module `<math.h>` ou la curieuse expression booléenne `t != t`, qui n'est vraie que pour les Not-A-Number !

Question 3. Peut-on écrire une fonction pour changer un coefficient de P, avec la signature suivante ?

```
// change le coefficient de P(x) devant x^k, k=0,1,2
void Polynome_set_coef( Polynome P, int k, double c );
```

2 Pointeurs

2.1 Définition et opérateurs essentiels

On appelle *pointeur* un couple adresse en mémoire/type de l'élément pointé.

Tout pointeur vers un type `T` a un type noté `T*`. On voit qu'il s'agit d'un autre usage de l'opérateur multiplication "`*`". Afin de les différencier plus facilement, je le note `*` dans les codes.

Une variable dont le type permet de stocker un pointeur est appelée *variable pointeur*. C'est une variable comme les autres, sauf que la donnée qu'elle contient est un nombre qui désigne une adresse en mémoire. Sa taille en octet dépend donc de la taille du bus d'adresse de votre carte mère. Tous les pointeurs (e.g. `int*`, `double*`, `char*`, `void*`, `Equipe*`, `Joueur*`) occupent tous la même taille en mémoire, soit 4 octets = 32 bits (vieille architecture 32 bits), soit 8 octets = 64 bits (tous les PC/mac modernes).

L'*opérateur d'adresse* '`&`' appliqué à une *variable* (en fait à toute lvalue, c'est-à-dire un identifiant désignant une case en mémoire) retourne un pointeur qui est l'adresse de la variable, et dont le type est le type de la variable.

L'*opérateur d'indirection* '`*`' appliqué à un *pointeur* (en fait toute expression dont la valeur est une adresse en mémoire couplé à un type) retourne la *variable* stockée à l'adresse pointée. C'est l'opérateur inverse de l'opérateur d'adresse.

```
int* ptr;      // variable pointeur vers int, non initialisé.
int a = 7;     // variable de type int
ptr   = &a;    // la valeur de ptr est l'adresse en mémoire de a
int b = *ptr; // comme ptr pointe vers la zone mémoire affectée à a,
               // *ptr est la variable a, donc sa valeur est 7
*ptr  = 3;     // comme *ptr est la variable a, a vaut 3 après.
&a    = &b;    // INVALIDE, car &a est une valeur (une adresse), pas une variable
```

On voit que le système de pointeurs permet de définir des alias à des variables. Dans l'exemple précédent, `*ptr` est un alias pour la variable `a`.

On note que pour toute variable `x`, `*&x` est la variable `x`.

Inversement, pour tout pointeur `t` (donc pas forcément une variable), `&*t` est le pointeur `t`.

Attention, déclarer une variable pointeur vers un type `T` n'implique pas du tout qu'une variable de type `T` est créée ou même existe par ailleurs.

```
Equipe* ptrE;              // ptrE est une variable pointeur, de taille 8 octets
// mais aucune variable de type Equipe n'existe pour autant.
Equipe_affiche( *ptrE ); // a donc toutes les chances de planter !
*ptrE.number = 17;         // a donc aussi toutes les chances de planter !
ptrE = &rouge;             // à partir de là, on peut faire une indirection.
Equipe_affiche( *ptrE ); // affiche l'équipe rouge
```

2.2 Passage par adresse, ou passage en entrée/sortie du C

Les pointeurs vont nous permettre de faire des fonctions C qui peuvent modifier des données passées en paramètre et allouées en dehors de la fonction. Le principe sera de passer les adresses en mémoire des données (que l'on veut lire ou modifier) plutôt ques les valeurs de ces données. Grâce à l'opérateur d'indirection, on pourra alors chercher les variables pointées et les utiliser en lecture ou en écriture.

On reprend ci-dessous l'exemple classique de la fonction d'échange des valeurs de deux variables.

```
void echange( int* pi, int* pj )
{ // Comme la valeur de pi est l'adresse de x,
  // *pi est un alias pour la variable x (*pi = *&x = x)
  int t = *pi;
  *pi = *pj;
  *pj = t;
}

int main( void )
{
  int x = 4;
  int y = 8;
  echange( &x, &y );
  // x et y sont changés: x=8, y=4.
}
```

2.3 Notation flèche pour les pointeurs vers une structure

On a vu précédemment que les variables de type structure sont passés par valeur en paramètre. Or ça peut être coûteux en temps et en mémoire si la structure occupe une grande zone mémoire. En général, on choisit en C de passer les structures par adresse, même si on ne cherche pas à modifier les données stockées. Ainsi seulement 8 octets sont passés en paramètres (l'adresse de la variable structure), et non des centaines d'octets (comme pour l'exemple des structures `Joueur` et `Equipe`). Cela donnerait:

```
void Joueur_affiche( Joueur* ptrJ )
{ // *ptrJ est un alias de jcvd
  printf( "nom=%s prenom=%s pseudo=%s number=%d",
          (*ptrJ).nom, (*ptrJ).prenom, (*ptrJ).pseudo, (*ptrJ).number );
}
...
Joueur jcvd = { "Vandamme", "Jean-Claude", "Aware", 99999 };
Joueur_affiche( &jcvd );
// ou en passant par une variable pointeur
Joueur* ptrJ = &jcvd;
Joueur_affiche( ptrJ );
```

On note néanmoins la lourdeur de l'utilisation des champs de la structure pointée: on fait d'abord une indirection sur le pointeur pour obtenir la variable structure, puis on précise le champ avec la notation ".".

Pour éviter ça, le C propose la notation flèchée "`->`", qui mise entre un pointeur vers une structure et un nom de champ, a le même rôle que la syntaxe précédente. Dit autrement "`ptrS->champ`" est équivalent à "`(*ptrS).champ`". On écrirait alors le code précédent plutôt ainsi:

```
void Joueur_affiche( Joueur* ptrJ )
{ // *ptrJ est un alias de jcvd
  printf( "nom=%s prenom=%s pseudo=%s number=%d",
          ptrJ->nom, ptrJ->prenom, ptrJ->pseudo, ptrJ->number );
}
...
```

2.4 Les tableaux sont (presque toujours) des pointeurs

A l'initialisation dans un bloc d'instruction, un tableau C provoque la création d'autant de variables que le nombre spécifié en tre crochets. Ensuite le nom du tableau n'est en fait qu'une valeur qui est l'adresse vers sa première case (plus précisément un pointeur puisque le type de la valeur pointée est connu).

On note que l'opérateur d'adresse `&` sur un tableau `t` retourne la même chose que `t` tout court. Il est donc inutile de passer son adresse pour un passage par adresse. De même on note que `t` est identique à `&t[0]`.

En revanche, déclaré en tant que paramètre d'une fonction, un tableau `T t[]` est en fait une variable pointeur vers `T`, initialisée à l'adresse donnée par l'argument donné à l'appel de la fonction. On peut bien sûr l'utiliser comme un tableau, avec des accès grâce à l'opérateur [].

```
void remove_punctuation( char s[] ); // en fait char* s
{
  int i = 0;
  while ( s[ i ] != 0 )
  {
    char c = s[ i ];
    if ( c == '.' || c == ',' || c == ';' || c == ':'
         || c == '?' || c == '!' || c == '\'' || c == '\"' )
      s[ i ] = ' ';
    i += 1;
  }
}
...
char str[] = "Messieurs les Anglais, tirez les premiers !";
remove_punctuation( str );
printf( "%s\n", str ); // Messieurs les Anglais  tirez les premiers  "
```

Les différences entre variable pointeur et tableau sont donc dans l'initialisation dans un bloc d'instruction.

On note aussi le cas spécifique de l'initialisation d'un tableau de caractères ou d'une variable pointeur vers caractère par un littéral :

```
char* nom1 = "Borjour";   /* alloué dans la section DATA, nom1 pointe donc
                             vers une zone non modifiable. */
nom1[ 2 ] = 'n';          /* est donc invalide; */
char nom2[] = "Borjour"; /* alloué sur la pile, nom2 pointe donc
                             vers une zone modifiable. */
nom2[ 2 ] = 'n';          /* est donc valide;
```

2.5 Arithmétique des pointeurs et opérateur crochet

La force des pointeurs en C est qu'ils ne sont pas limités à la seule zone mémoire qu'ils pointent. On peut *déplacer* les pointeurs, *comparer* les pointeurs, *affecter* les pointeurs à des variables (pointeurs), *calculer* la distance entre deux pointeurs. Il y a aussi un lien très fort entre tableaux et pointeurs. Cet ensemble d'opérations est appelé *arithmétique des pointeurs*.

Addition et soustraction d'entiers. Tout pointeur `t` de type `T*`, ou pointeur vers `T`, représente l'adresse en mémoire d'une variable de type `T`. Le pointeur `t+1` désigne lui la zone mémoire placée juste après la zone pointée par `t`. Si on imagine qu'il y a une suite de variables de type `T` consécutives en mémoire, alors `t` désigne la première variable, `t+1` désigne la deuxième variable, `t+2` désigne la 3ème variable, etc.

En général, les variables du même type consécutives en mémoire sont des tableaux. On voit alors que le pointeur permet lui aussi de se déplacer dans le tableau.

```
int t[ 4 ] = { 12, 3, 9, 15 };
bool  ok_0 = *t     == 12; // t[ 0 ]
bool  ok_1 = *(t+1) == 3;  // t[ 1 ]
bool  ok_2 = *(t+2) == 9;  // t[ 2 ]
bool  ok_3 = *(t+3) == 15; // t[ 3 ]
bool att_4 = *(t+4) == ?;  // t[ 4 ] hors mémoire !
```

Dit autrement, si `t` est un pointeur et `n` est un entier, on a équivalence des notations

"à la pointeur"		"à la tableau"
`*t`	⇔	`t[ 0 ]`
`*(t+n)`	⇔	`t[ n ]`
`*(n+t)`	⇔	`t[ n ]`
`t`	⇔	`&t[ 0 ]`
`t+n`	⇔	`&t[ n ]`
`n+t`	⇔	`&t[ n ]`

Le nombre `n` pouvant être négatif, les règles pour la soustraction sont les mêmes.

En résumé, le compilateur se sert de la taille occupée par le type `T` (i.e. `sizeof(T)`) pour calculer le vrai décalage d'adresse en mémoire. Par exemple, si `T` était `int`, donc `sizeof(int)==4` octets, alors la notation `t+n` correspond à additionner `4*n` à l'adresse `t`.

NB (fun): les règles précédentes impliquent que les notations `t[ n ]` et `n[ t ]` sont équivalentes. Cette seconde notation est à proscrire (sauf si vous soumettez du code à l'IOCCC !).

Comparer des pointeurs. Les pointeurs étant des adresses en mémoire, on peut les comparer (en tant qu'entiers), avec les opérateurs classiques `==`, `!=`, `<`, `>`, `<=`, `>=`.

Ce genre de comparaison entre pointeurs a de sens pour des pointeurs qui pointent dans une zone mémoire continue. Comparer des pointeurs dans le même tableau a un sens. Comparer des adresses de variables définies dans des fonctions différentes n'a pas de sens en général.

Variables pointeur. Les pointeurs vers `T` peuvent être stockés dans des variables de type `T*`. Ensuite ces variables sont affectables par l'opérateur `=`, modifiables par les opérateurs `+=`, `-=`, `++` ou `--` de manière naturelle.

Le code suivant initialise un tableau à zéro, via des pointeurs:

```
int  t[ 10 ];  // t est un pointeur vers int, mais pas une variable pointeur
int* p = t;    // p pointe au même endroit que t, mais est une variable
int* q = t+10; // q pointe après la fin du tableau, ie &t[ 10 ]
for ( ; p != q; p++ )
  *p = 0;      // *p est successivement chaque variable t[ i ]
```

Distance entre pointeurs. Enfin on peut calculer la distance entre 2 pointeurs vers un même type (au sens de nombre de variables entre), en faisant la soustraction d'une pointeur à l'autre. Sur l'exemple précédent, on aurait:

```
int  t[ 10 ];  // t est un pointeur vers int, mais pas une variable pointeur
int* p = t;    // p pointe au même endroit que t, mais est une variable
int* q = t+10; // q pointe après la fin du tableau, ie &t[ 10 ]
int  d = q - p; // d vaut 10
int  e = p - q; // e vaut -10
```

On voit que la distance est compatible avec les opérateurs de comparaison, c'est-à-dire:

```
p - q == 0    ⇔   p == q
p - q != 0    ⇔   p != q
p - q >= 0    ⇔   p >= q
p - q <= 0    ⇔   p <= q
p - q > 0     ⇔   p > q
p - q < 0     ⇔   p < q
```

Exemple : (Palindrome) La fonction suivante détermine si la chaîne de caractères données est un palindrome.

```
bool palindrome( char* p )
{
  char* q = p;           // p est au début du mot
  while (*q != 0 ) q++; // met q après la fin du mot
  while ( p < --q ) // tant que p et q ne se croise pas
    if ( *p++ != *q ) // teste si caractères identiques
      return false;     // si non, ce n'est pas un palindrome
  return true; // lorsque p et q se croise, c'est un palindrome
}
```

Une fois équipé du type pointeur, on voit que beaucoup d'algorithmes qui utilisent un our plusieurs indices pour parcourir des tableaux pourrait être écrits à l'aide de pointeurs qui visitent les tableaux. L'avantage est de pouvoir parfois faire du code plus rapide. L'inconvénient est souvent une lisibilité peut-être moindre, surtout pour les programmeurs peu habitués au C.

2.6 Le type pointeur void* et le transtypage

Pour finir cette fiche assez dense, il existe un type pointeur particulier en C, défini comme `void*`, qui permet de stocker des pointeurs "quelconques", genre `int*`, `char*`, `Equipe*`, etc.

A quoi ça sert ? On s'en sert pour représenter des pointeurs génériques, à des endroits où on n'a pas besoin de savoir le type pointé ou des endroits où on ne connait pas encore le type qui sera pointé. Par exemple, la bibliothèque standard C offre des fonctions pour allouer de la mémoire (`malloc` et autres) ou des fonctions pour trier des données (`qsort`). Ces fonctions utilisent des pointeurs `void*` comme paramètres ou valeur de retour.

Tout pointeur `T*` peut être convertit en pointeur `void*` sans mécanisme particulier. A l'inverse, si vous voulez transformer un pointeur `void*` en un pointeur `T*`, vous devez l'indiquer au compilateur en faisant un transtypage, ou *cast*, comme ci-dessous:

```
int  t[ 1 ] = 0x04030201;  // t est un pointeur int*
void* p = t; // valide, p a la même adresse que t
char* q = (char*) p; // cast pour voir t comme un tableau de char
// Sur x86_64, processeur little endian
char a = q[ 0 ]; // donc a = 0x01 = 1
char b = q[ 1 ]; // donc b = 0x02 = 2
char c = q[ 2 ]; // donc c = 0x03 = 3
char d = q[ 3 ]; // donc d = 0x04 = 4
```

Attention, Les pointeurs `void*` ne disposent pas des opérations arithmétiques, ni même de l'opérateur d'indirection `*` (et bien sûr l'opérateur `[ n ]` ne marche pas). En effet, ne connaissant pas le type pointé, le compilateur ne peut inférer la taille ou la valeur de la variable pointée. Si vous voulez récupérer la valeur pointée par un tel pointeur, c'est à vous de savoir dans quel type pointeur il doit être transtypé/casté. Cela dépend du contexte.

- On appelle *pointeur* un couple adresse en mémoire/type de l'élément pointé `T`. Son type est `T*`.
- L'*opérateur d'adresse* '`&`' appliqué à une *variable* de type `T` retourne un pointeur `T*` dont la valeur est l'adresse de la variable.
- L'*opérateur d'indirection* '`*`' appliqué à un *pointeur* `T*` retourne la *variable* stockée à l'adresse pointée.
- Ces opérateurs sont inverses l'un de l'autre.
- On peut déplacer en mémoire les pointeurs `T*`, cela les fait se déplacer de variables `T` en variables `T`.
- La notation `t[ n ]`, où `t` est un pointeur/tableau et `n` un entier, est équivalente à la notation `*(t+n)`.
- Les pointeurs peuvent donc servir à parcourir les tableaux.
- Les pointeurs `void*` représentent des pointeurs vers des types quelconques.
- Cette généricité empêche l'utilisation de la plupart des opérateurs dessus, sauf le transtypage/cast `(T*)` qui permet au programmeur de forcer le type pointé.
- Si `s` est un pointeur vers une variable de type structure, avec un champ `nom`, alors la syntaxe "`s->nom`" est préférée à la syntaxe "`(*s).nom`".

Question 4. Moyenne d'un tableau. Le code suivant calcule la moyenne d'un tableau. Ecrivez-le avec des pointeurs seulement.

```
double moyenne( double p[], int n )
{
  double s = 0.0;
  for ( int i = 0; i < n; ++i )
    s += p[ i ];
  return s / n;
}
```

Question 5. Plus petit élément d'un tableau (i.e. argument minimal) Ecrivez une fonction qui retourne un pointeur vers un élément du tableau donné en entrée, tel que cet élément est plus petit ou égal aux autres. Le tableau est spécifié via un intervalle de pointeur `[p,q)`, où `p` pointe au début du tableau et `q` juste après son dernier élément. Si `p == q` (tableau vide), retourne `p`.

```
int* argmin( int* p, int* q );
...
int t[ 8 ] = { 4, 2, 17, 14, 3, 8, 2, 12 };
int* m = argmin( t, t + 8 ); // m = &t[1] ou &t[6]
```

Question 6. Sous-chaîne d'une chaîne de caractères. Ecrivez une fonction qui teste si une chaîne `s` apparaît dans une chaîne `t` et retourne un pointeur dans `t` là où `s` apparaît, ou `NULL` sinon.

```
char* find( char* s, char* t);
...
```

```
char* f1 = find( "cav", "capitaine caverne" ); // trouvé
char* f2 = find( "arverne", "capitaine caverne" ); // NULL
```