C : les pointeurs

Lycée Thiers

Crédits

- Ce cours de OpenClassRoom

Crédits

- Ce cours de OpenClassRoom
- Un autre de Zeste de savoir

Crédits

- Ce cours de OpenClassRoom
- Un autre de Zeste de savoir
- Un cours d'Anne Canteaut

Crédits

- Ce cours de OpenClassRoom
- Un autre de Zeste de savoir
- Un cours d'Anne Canteaut
- Et toujours Wikipedia

Accès à la mémoire

- Un entier naturel codé en binaire peut-être vu comme :

Accès à la mémoire

- Un entier naturel codé en binaire peut-être vu comme :
  - ce qu'il est : un nombre destiné par exemple à faire l'objet d'opérations arithmétiques,

Accès à la mémoire

- Un entier naturel codé en binaire peut-être vu comme :
  - ce qu'il est : un nombre destiné par exemple à faire l'objet d'opérations arithmétiques,
  - une adresse mémoire. Dans ce cas l'utilisation arithmétique du nombre n'est pas ce qui nous intéresse.

Accès à la mémoire

- Un entier naturel codé en binaire peut-être vu comme :
  - ce qu'il est : un nombre destiné par exemple à faire l'objet d'opérations arithmétiques,
  - une adresse mémoire. Dans ce cas l'utilisation arithmétique du nombre n'est pas ce qui nous intéresse.
- Dans les processeurs ont donc été créés des *registres d'adresses* et dans les langages de programmation, un type dédié.

Accès à la mémoire

- Un entier naturel codé en binaire peut-être vu comme :
  - ce qu'il est : un nombre destiné par exemple à faire l'objet d'opérations arithmétiques,
  - une adresse mémoire. Dans ce cas l'utilisation arithmétique du nombre n'est pas ce qui nous intéresse.
- Dans les processeurs ont donc été créés des *registres d'adresses* et dans les langages de programmation, un type dédié.
- Pointeurs : Apparus dans AIGOL 68. Le langage C y a ajouté l'*arithmétique des pointeurs* : quand on incrémente un tel pointeur, il n'est en fait pas forcément incrémenté de un, mais de la taille du type pointé. Cette arithmétique (hors programme) permet donc de se déplacer dans la mémoire.

Accès à la mémoire

- Un entier naturel codé en binaire peut-être vu comme :
  - ce qu'il est : un nombre destiné par exemple à faire l'objet d'opérations arithmétiques,
  - une adresse mémoire. Dans ce cas l'utilisation arithmétique du nombre n'est pas ce qui nous intéresse.
- Dans les processeurs ont donc été créés des *registres d'adresses* et dans les langages de programmation, un type dédié.
- Pointeurs : Apparus dans AIGOL 68. Le langage C y a ajouté l'*arithmétique des pointeurs* : quand on incrémente un tel pointeur, il n'est en fait pas forcément incrémenté de un, mais de la taille du type pointé. Cette arithmétique (hors programme) permet donc de se déplacer dans la mémoire.
- L'utilisation des pointeurs permet d'avoir accès à la mémoire. On peut se déplacer de case mémoire en case mémoire. Avantage : optimisations sur l'utilisation de la mémoire ou la performance.

Déclaration

- Syntaxe :

```
1 type *nom_du_pointeur;
2
```

Déclaration

- Syntaxe :

```
1 type *nom_du_pointeur;
2
```

- L'astérisque peut être entourée d'espaces et placée n'importe où entre le type et l'identificateur.

```
1 int *ptr;// autorisé
2 int * ptr;// autorisé
3 int* ptr;// autorisé
4
```

Initialisation

- Comme les autres variables, un pointeur ne possède pas de valeur par défaut. pour lui en attribuer une, on utilise l'opérateur d'*adressage* (ou de *référencement*) `&` .

Initialisation

- Comme les autres variables, un pointeur ne possède pas de valeur par défaut. pour lui en attribuer une, on utilise l'opérateur d'*adressage* (ou de *référencement*) `&` .
- Déclaration puis affectation

```
1 int a = 10;
2 int *p;
3 p = &a;
4
```

Initialisation

- Comme les autres variables, un pointeur ne possède pas de valeur par défaut. pour lui en attribuer une, on utilise l'opérateur d'*adressage* (ou de *référencement*) `&`.
- Déclaration puis affectation

```
1 int a = 10;
2 int *p;
3 p = &a;
4
```

- Déclaration et affectation dans la foulée

```
1 int a = 10;
2 int *p = &a;
3
```

Initialisation

- Comme les autres variables, un pointeur ne possède pas de valeur par défaut. pour lui en attribuer une, on utilise l'opérateur d'*adressage* (ou de *référencement*) `&`.
- Déclaration puis affectation

```
1 int a = 10;
2 int *p;
3 p = &a;
4
```

- Déclaration et affectation dans la foulée

```
1 int a = 10;
2 int *p = &a;
3
```

- Savoir faire : déclaration et initialisation d'un pointeur dans un type donné ♡

Affichage des adresses

- Avec :

```
1 int  a =10;
2 int  *p = &a;
3 printf ("a=%d, &a=%p, &p=%p\n",*p,p,&p);
4
```

Affichage des adresses

- Avec :

```
1 int a =10;
2 int *p = &a;
3 printf ("a=%d, &a=%p, &p=%p\n",*p,p,&p);
4
```

- On obtient :

```
a=10, &a=0x7ffc7aaa612c, &p=0x7ffc7aaa6130
```

Affichage des adresses

- Avec :

```
1 int a =10;
2 int *p = &a;
3 printf ("a=%d, &a=%p, &p=%p\n",*p,p,&p);
4
```

- On obtient :

```
a=10, &a=0x7ffc7aaa612c, &p=0x7ffc7aaa6130
```

nom	adresse	valeur
p	0x7ffc7aaa6130	0x7ffc7aaa612c
	...	
a	0x7ffc7aaa612c	10

Indirection♡

- L'opérateur d'indirection (ou de *déréférencement*) `*` prend un pointeur comme opérande et se place juste avant celui-ci. On accède ainsi à la valeur de l'objet référencé par le pointeur, aussi bien pour la lire que pour la modifier.

Indirection♡

- L'opérateur d'indirection (ou de *déréférencement*) `*` prend un pointeur comme opérande et se place juste avant celui-ci. On accède ainsi à la valeur de l'objet référencé par le pointeur, aussi bien pour la lire que pour la modifier.

```
1 int a = 3;
2 int *p = &a;// '*' sert à la déclaration
3 printf("a=%d\n", *p);// '*' sert au déréférencement
4 *p = 20;// '*' sert au déréférencement
5 printf("a=%d\n", a);// a a été modifié
6
```

Indirection♡

- L'opérateur d'indirection (ou de *déréférencement*) `*` prend un pointeur comme opérande et se place juste avant celui-ci. On accède ainsi à la valeur de l'objet référencé par le pointeur, aussi bien pour la lire que pour la modifier.

```
1 int a = 3;
2 int *p = &a;// '*' sert à la déclaration
3 printf("a=%d\n", *p);// '*' sert au déréférencement
4 *p = 20;// '*' sert au déréférencement
5 printf("a=%d\n", a);// a a été modifié
6
```

- On obtient

```
a=3
a=20
```

Les différentes casquettes de `*`

L'opérateur `*` sert au moins à trois choses distinctes :

- À la déclaration de pointeurs : `int *p = &i;`

Les différentes casquettes de `*`

L'opérateur `*` sert au moins à trois choses distinctes :

- À la déclaration de pointeurs : `int *p = &i;`
- Au déréférencement : `*p=20; // i prend la valeur 20`

Les différentes casquettes de `*`

L'opérateur `*` sert au moins à trois choses distinctes :

- À la déclaration de pointeurs : `int *p = &i;`
- Au déréférencement : `*p=20; // i prend la valeur 20`
- À la multiplication `3*5;`

Pointeurs et constantes

Pour info :

- La règle du mot-clé `const` est qu'il s'applique sur ce qui est immédiatement à sa gauche, et que s'il n'y a rien à gauche alors ça s'applique sur l'élément immédiatement à droite

Pointeurs et constantes

Pour info :

- La règle du mot-clé `const` est qu'il s'applique sur ce qui est immédiatement à sa gauche, et que s'il n'y a rien à gauche alors ça s'applique sur l'élément immédiatement à droite
- Un pointeur peut être déclaré constant :

```
1 int * const ptr; /* Un pointeur constant sur int. non ct */
```

L'étoile est entre le type et le mot clé `const`

Pointeurs et constantes

Pour info :

- La règle du mot-clé `const` est qu'il s'applique sur ce qui est immédiatement à sa gauche, et que s'il n'y a rien à gauche alors ça s'applique sur l'élément immédiatement à droite
- Un pointeur peut être déclaré constant :

```
1 int * const ptr; /* Un pointeur constant sur int. non ct */
```

L'étoile est entre le type et le mot clé `const`

- Un pointeur peut pointer sur une constante

```
1 int const *ptr; /* Pointeur sur int constant. */
```

L'étoile est entre le mot clé `const` et le nom du pointeur

Pointeurs et constantes

Pour info :

- La règle du mot-clé `const` est qu'il s'applique sur ce qui est immédiatement à sa gauche, et que s'il n'y a rien à gauche alors ça s'applique sur l'élément immédiatement à droite
- Un pointeur peut être déclaré constant :

```
1 int * const ptr; /* Un pointeur constant sur int. non ct */
```

L'étoile est entre le type et le mot clé `const`

- Un pointeur peut pointer sur une constante

```
1 int const *ptr; /* Pointeur sur int constant. */
```

L'étoile est entre le mot clé `const` et le nom du pointeur

- Pointeur constant sur objet constant.

```
1 int const * const ptr; /* Pointeur constant sur int ct. */
2 const int * const ptr; /* Pointeur ct sur int ct. */
3
```

Piège à la déclaration

- On l'a vu `int* p;` et `int *p;` sont compris de la même façon par le compilateur.

Piège à la déclaration

- On l'a vu `int* p;` et `int *p;` sont compris de la même façon par le compilateur.
- `int *p;` permet au lecteur de savoir que `*p` est un entier. La forme avec l'étoile collée au nom est préférable.

Piège à la déclaration

- On l'a vu `int* p;` et `int *p;` sont compris de la même façon par le compilateur.
- `int *p;` permet au lecteur de savoir que `*p` est un entier. La forme avec l'étoile collée au nom est préférable.
- `int* x,y;` ne déclare pas deux pointeurs sur entier mais un pointeur sur entier et un entier.
  On le comprend mieux en écrivant `int *x,y;`

Utilité♡

- On peut utiliser les pointeurs pour faire des *effets de bords* en les passant en paramètres de fonctions. Ex : passer un pointeur sur une variable x en paramètre de f, permet d'avoir accès au contenu de x et donc de le modifier (voir 2 slides suivants).

Utilité♡

- On peut utiliser les pointeurs pour faire des *effets de bords* en les passant en paramètres de fonctions. Ex : passer un pointeur sur une variable x en paramètre de f, permet d'avoir accès au contenu de x et donc de le modifier (voir 2 slides suivants).
- Les pointeurs sont une des façons de contourner l'obligation qu'une fonction C ne renvoie qu'une seule valeur. L'autre façon courante étant l'utilisation de *structure* (ou enregistrement).

Utilité♡

- On peut utiliser les pointeurs pour faire des *effets de bords* en les passant en paramètres de fonctions. Ex : passer un pointeur sur une variable x en paramètre de f, permet d'avoir accès au contenu de x et donc de le modifier (voir 2 slides suivants).
- Les pointeurs sont une des façons de contourner l'obligation qu'une fonction C ne renvoie qu'une seule valeur. L'autre façon courante étant l'utilisation de *structure* (ou enregistrement).
- On peut les utiliser aussi pour stocker des adresses mémoires allouées *dynamiquement* (*i.e.* au run time) par l'application.

Conséquence du passage par valeur

```
1  int incremente(int y){
2        y = y+1;
3     return y;
4    }
5  int main(){
6    int y =3;
7     printf ("y=%d,incremente(y)=%d\n", y,incremente(y));
8     printf ("; y=%d\n",y);
9       return 0;
10   }
11
```

- y=3,incremente(y)=4; y=3

Conséquence du passage par valeur

```
1  int incremente(int y){
2        y = y+1;
3      return y;
4    }
5  int main(){
6    int y =3;
7     printf ("y=%d,incremente(y)=%d\n", y,incremente(y));
8     printf ("; y=%d\n",y);
9      return 0;
10   }
11
```

- y=3,incremente(y)=4; y=3
- Le contenu de la variable **y** de **main** n'a pas été modifié par l'appel de **incremente**. On dit que le passage de l'argument se fait *par valeur*, ce qui signifie que c'est une copie de **y** qui est passée en paramètre de la fonction. Pour changer **y** on utilisera son adresse, c'est à dire un *pointeur*.

Conséquence du passage par valeur

```
1  #include <stdio.h>
2
3  void incremente(int *y){
4      *y = *y+1;
5  }
6
7  int main(int argc, char *argv []) {
8    int y =3;
9     printf ("y=%d\n",y);
10   incremente(&y);
11    printf ("Après incremente(y) : y=%d\n",y);
12      return 0;
13 }
14
```

Après compilation/exécution :

```
$ ./a.out
y=3
Après incremente(y) : y=4
```

Pointeurs dans d'autres langages

- Dans les langages de plus haut niveau (ex : OCAML), l'utilisation des pointeurs est supprimée, au profit des *références* et des tableaux dynamiques gérés par le compilateur.

Pointeurs dans d'autres langages

- Dans les langages de plus haut niveau (ex : OCAML), l'utilisation des pointeurs est supprimée, au profit des *références* et des tableaux dynamiques gérés par le compilateur.
- On y gagne en simplicité, on évite de nombreux bugs mais on perd certaines possibilités d'optimisation.

« Retourner » plusieurs valeurs

Passage de pointeurs en paramètres

- On contourne l'obligation faite aux fonctions C de ne retourner qu'une seule valeur.

« Retourner » plusieurs valeurs

Passage de pointeurs en paramètres

- On contourne l'obligation faite aux fonctions C de ne retourner qu'une seule valeur.
- Exemple : On veut appliquer à un point une translation de vecteur donné et récupérer le résultat.

« Retourner » plusieurs valeurs

Passage de pointeurs en paramètres

- On contourne l'obligation faite aux fonctions C de ne retourner qu'une seule valeur.
- Exemple : On veut appliquer à un point une translation de vecteur donné et récupérer le résultat.
- On ne peut pas retourner un tuple de coordonées comme en Python ou Ocaml.

« Retourner » plusieurs valeurs

Passage de pointeurs en paramètres

- On contourne l'obligation faite aux fonctions C de ne retourner qu'une seule valeur.
- Exemple : On veut appliquer à un point une translation de vecteur donné et récupérer le résultat.
- On ne peut pas retourner un tuple de coordonées comme en Python ou Ocaml.
- Solution : utiliser des pointeurs sur les coordonnées du résultat.

```
1 void  translate (double x, double y, double vx, double vy,
2                  double *rx, double *ry)
3
```

applique une translation de vecteur $V(v_x, v_y)$ à un point $A(x, y)$ et met les coordonnées du résultat $R(r_x, r_y)$ dans deux variables.
Les coordonnées de A et V sont passées en paramètres. Pour le résultat, ce sont adresses de ses coordonnées qui sont passées en paramètres.

Retourner plusieurs valeurs (suite)

Passage de pointeurs en paramètres

```
1  void translate (double x, double y, double vx, double vy,
2                  double *rx, double *ry){
3    // translation de vecteur V(vx,vy)
4    *rx = x + vx;
5    *ry = y + vy;
6  }// fin translate
7
8  int main(void){
9    double x = 3, y = 6;// coordonnées du point A(x,y)
10   double rx, ry;// résultat de la translation
11    translate (x,y,1,2,&rx,&ry);// translater A de (1,2)
12    printf ("rx=%.1f ry=%.1f\n",rx,ry);
13   return EXIT_SUCCESS;} // fin main
14
```

Retourner plusieurs valeurs (suite)

Passage de pointeurs en paramètres

```c
void translate (double x, double y, double vx, double vy,
                double *rx, double *ry){
  // translation de vecteur V(vx,vy)
  *rx = x + vx;
  *ry = y + vy;
}// fin translate

int main(void){
  double x = 3, y = 6;// coordonnées du point A(x,y)
  double rx, ry;// résultat de la translation
   translate (x,y,1,2,&rx,&ry);// translater A de (1,2)
   printf ("rx=%.1f ry=%.1f\n",rx,ry);
  return EXIT_SUCCESS;} // fin main
```

- On obtient

```
x=4.0 y=8.0
```

Pointeur nul

- Un *pointeur nul* est un pointeur contenant une adresse invalide. Cette adresse invalide dépend du système d'exploitation, mais elle est la même pour tous les pointeurs nuls. Ainsi, deux pointeurs nuls ont une valeur égale.

Pointeur nul

- Un *pointeur nul* est un pointeur contenant une adresse invalide. Cette adresse invalide dépend du système d'exploitation, mais elle est la même pour tous les pointeurs nuls. Ainsi, deux pointeurs nuls ont une valeur égale.
- La constante `NULL` est définie dans l'en-tête `<stddef.h>`

```
1 int *p = NULL; /* Un pointeur nul */
2
```

Pointeur nul

- Un *pointeur nul* est un pointeur contenant une adresse invalide. Cette adresse invalide dépend du système d'exploitation, mais elle est la même pour tous les pointeurs nuls. Ainsi, deux pointeurs nuls ont une valeur égale.
- La constante `NULL` est définie dans l'en-tête `<stddef.h>`

```
1 int *p = NULL; /* Un pointeur nul */
2
```

- Le pointeur `NULL` est souvent utilisé comme marqueur de fin dans une structure définie récursivement (exemple : listes chaînées).

Pointeur nul

- Un *pointeur nul* est un pointeur contenant une adresse invalide. Cette adresse invalide dépend du système d'exploitation, mais elle est la même pour tous les pointeurs nuls. Ainsi, deux pointeurs nuls ont une valeur égale.
- La constante `NULL` est définie dans l'en-tête `<stddef.h>`

```
1 int *p = NULL; /* Un pointeur nul */
2
```

- Le pointeur `NULL` est souvent utilisé comme marqueur de fin dans une structure définie récursivement (exemple : listes chaînées).
- Il sert aussi à indiquer que quelque chose s'est mal passé (ex `malloc`)

Le mot clé `void`

- `void` n'est pas un type, c'est un mot clé. Plus précisément, c'est un type dit « incomplet », c'est à dire que sa taille n'est pas calculable et qu'il n'est pas utilisable dans des expressions.

Le mot clé `void`

- `void` n'est pas un type, c'est un mot clé. Plus précisément, c'est un type dit « incomplet », c'est à dire que sa taille n'est pas calculable et qu'il n'est pas utilisable dans des expressions.
- On l'utilise pour déclarer qu'une fonction ne renvoie pas de valeur : `void f(int x)`

Le mot clé `void`

- `void` n'est pas un type, c'est un mot clé. Plus précisément, c'est un type dit « incomplet », c'est à dire que sa taille n'est pas calculable et qu'il n'est pas utilisable dans des expressions.
- On l'utilise pour déclarer qu'une fonction ne renvoie pas de valeur : `void f(int x)`
- Pour les fonctions sans argument, préferer `int f(void)` (qui signifie explicitement que `f` ne prend pas d'argument) à `int f()` qui indique que le nombre d'arguments de `f` n'est pas connu.

Le mot clé `void`

- `void *` désigne un pointeur vers une valeur dont on ne connaît pas le type (c'est un pointeur *générique*). Un pointeur sur `void` est considéré comme un pointeur générique : il peut référencer n'importe quel type.
  Exemple :

```
1 void * malloc(  size_t  memorySize );//prototype de malloc
2
```

Le mot clé `void`

- `void *` désigne un pointeur vers une valeur dont on ne connaît pas le type (c'est un pointeur *générique*). Un pointeur sur `void` est considéré comme un pointeur générique : il peut référencer n'importe quel type.
  Exemple :

```
1 void * malloc(  size_t  memorySize );//prototype de malloc
2
```

- Les règles de typage du C permettent d'utiliser `void*` là où une valeur d'un certain type de pointeur est attendu. On peut donc écrire

```
1 int *p = malloc(siezof(int));// alocation d'un pointeur sur le tas.
2
```

Le mot clé `void`

- `void *` désigne un pointeur vers une valeur dont on ne connaît pas le type (c'est un pointeur *générique*). Un pointeur sur `void` est considéré comme un pointeur générique : il peut référencer n'importe quel type.
  Exemple :

```
1 void * malloc(  size_t  memorySize );//prototype de malloc
2
```

- Les règles de typage du C permettent d'utiliser `void*` là où une valeur d'un certain type de pointeur est attendu. On peut donc écrire

```
1 int *p = malloc(siezof(int));// alocation d'un pointeur sur le tas.
2
```

- Enfin `void *` permet le *polymorphisme*. Exemple : une fonction qui agit sur un tableau dont les éléments sont de type quelconque. Nous n'utilisons pas cette fonctionnalité.

Le mot clé `void`

Le spécifieur de format `%p` de `printf` attend un pointeur sur n'importe quel type et affiche sa valeur (donc une adresse), que le pointeur pointe sur un objet de type `int, double, char...`

```
1 int a =10;
2 int *p= &a;
3 printf ("adresse=\%p",p); // affichage d'une adresse
4
```

Pointeur comme valeur retournée

- Une fonction peut retourner un pointeur.

Pointeur comme valeur retournée

- Une fonction peut retourner un pointeur.
- Cependant : l'objet référencé par le pointeur retourné doit toujours exister au moment de son utilisation.

Pointeur comme valeur retournée

- Une fonction peut retourner un pointeur.
- Cependant : l'objet référencé par le pointeur retourné doit toujours exister au moment de son utilisation.
- Il faut se poser la question :

Pointeur comme valeur retournée

- Une fonction peut retourner un pointeur.
- Cependant : l'objet référencé par le pointeur retourné doit toujours exister au moment de son utilisation.
- Il faut se poser la question :
  - l'adresse retournée est-elle celle d'une variable locale à la fonction (donc gérée automatiquement dans la *pile d'exécution*) ? Si c'est le cas, cette adresse peut servir dans d'autres contextes d'exécution et son contenu risque d'échapper au programmeur distrait.

Pointeur comme valeur retournée

- Une fonction peut retourner un pointeur.
- Cependant : l'objet référencé par le pointeur retourné doit toujours exister au moment de son utilisation.
- Il faut se poser la question :
  - l'adresse retournée est-elle celle d'une variable locale à la fonction (donc gérée automatiquement dans la *pile d'exécution*) ? Si c'est le cas, cette adresse peut servir dans d'autres contextes d'exécution et son contenu risque d'échapper au programmeur distrait.
  - l'adresse retournée désigne-t-elle une zone mémoire allouée sur le *tas* ? Si c'est le cas, il faut bien penser à la libérer par la suite.

Pointeur comme valeur retournée

```
1  int *ptr(void){ // retourne un pointeur sur int
2      int n;
3      return &n;}
4
5  int main(void){
6      int *p = ptr();
7      *p = 10;
8       printf ("%d\n", *p);
9      return 0;}
10
```

Pointeur comme valeur retournée

```
1  int *ptr(void){ // retourne un pointeur sur int
2      int n;
3      return &n;}
4
5  int main(void){
6      int *p = ptr();
7      *p = 10;
8       printf ("%d\n", *p);
9      return 0;}
10
```

- On obtient

```
$ ./a.out
Erreur de segmentation (core dumped)
```

Pointeur comme valeur retournée

- Le problème vient de la ligne

```
1 int n; // dans ptr()
2
```

Pointeur comme valeur retournée

- Le problème vient de la ligne

```
1 int n; // dans ptr()
2
```

- Les variables locales comme `n` de `ptr` se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Le segment de mémoire dans lequel sont stockées les variables temporaires est appelé *segment de pile*

Pointeur comme valeur retournée

- Le problème vient de la ligne

```
1 int n; // dans ptr()
2
```

- Les variables locales comme `n` de `ptr` se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Le segment de mémoire dans lequel sont stockées les variables temporaires est appelé *segment de pile*
- Leur emplacement en mémoire est libéré à la fin de l'exécution d'une fonction secondaire comme `ptr`.

Pointeur comme valeur retournée

- Le problème vient de la ligne

```
1 int n; // dans ptr()
2
```

- Les variables locales comme `n` de `ptr` se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme. Le segment de mémoire dans lequel sont stockées les variables temporaires est appelé *segment de pile*
- Leur emplacement en mémoire est libéré à la fin de l'exécution d'une fonction secondaire comme `ptr`.
- Donc la fonction **ptr** renvoie une adresse qui n'est plus pertinente et qui sera utilisée par d'autres appels à d'autres fonctions. On dit que c'est un *pointeur fantôme*

Pointeur comme valeur retournée

Mot clé `static`

- On modifie juste la ligne 3 du code précédent :

```
1       static int n;// dans ptr
2
```

Pointeur comme valeur retournée

Mot clé `static`

- On modifie juste la ligne 3 du code précédent :

```
1       static int n;// dans ptr
2
```

- On obtient

```
$ ./a.out
10
```

Pointeur comme valeur retournée

Mot clé `static`

- On modifie juste la ligne 3 du code précédent :

```
1       static int n;// dans ptr
2
```

- On obtient

```
$ ./a.out
10
```

- Une variable locale déclarée `static` occupe toujours la même adresse en mémoire à chaque appel de la fonction.
  On verra plus tard que la partie de la mémoire (allouée au programme) qui contient les variables statiques (comme d'ailleurs les variables globales) est appelée *segment de données*.

Pointeur comme valeur retournée

Mot clé `static`

- On modifie juste la ligne 3 du code précédent :

```
1       static int n;// dans ptr
2
```

- On obtient

```
$ ./a.out
10
```

- Une variable locale déclarée `static` occupe toujours la même adresse en mémoire à chaque appel de la fonction.
  On verra plus tard que la partie de la mémoire (allouée au programme) qui contient les variables statiques (comme d'ailleurs les variables globales) est appelée *segment de données*.
- Cette solution n'est pas d'un grand intérêt sauf si on a absolument besoin que l'adresse retournée soit toujours la même ! On préfère *l'allocation dynamique* (voir plus loin).

Pointeur sur pointeur

- Un pointeur **p** a une adresse qu'on récupère avec `&p`

Pointeur sur pointeur

- Un pointeur **p** a une adresse qu'on récupère avec `&p`
- Dans le `main`, écrivons :

```
1 int a = 10;
2 int *pa = &a;// pointeur sur int
3 int **pp = &pa;// pointeur sur pointeur sur int
4 printf ("a =%d,*pa=%d,**pp=%d\n", a,*pa,**pp);
5 printf ("pa=%p,*pp=%p\n",pa,*pp);
6
```

Pointeur sur pointeur

- Un pointeur **p** a une adresse qu'on récupère avec `&p`
- Dans le `main`, écrivons :

```
1 int a = 10;
2 int *pa = &a;// pointeur sur int
3 int **pp = &pa;// pointeur sur pointeur sur int
4 printf ("a =%d,*pa=%d,**pp=%d\n", a,*pa,**pp);
5 printf ("pa=%p,*pp=%p\n",pa,*pp);
6
```

- Avec `*pp` on récupère l'adresse de **a** et avec `**pp`, la valeur de **a**.

Pointeur sur pointeur

- Un pointeur **p** a une adresse qu'on récupère avec `&p`
- Dans le `main`, écrivons :

```
1 int a = 10;
2 int *pa = &a;// pointeur sur int
3 int **pp = &pa;// pointeur sur pointeur sur int
4 printf ("a =%d,*pa=%d,**pp=%d\n", a,*pa,**pp);
5 printf ("pa=%p,*pp=%p\n",pa,*pp);
6
```

- Avec `*pp` on récupère l'adresse de **a** et avec `**pp`, la valeur de **a**.
- Après compilation/exécution :

```
$ ./a.out
a =10,*pa=10,**pp=10
pa=0x7ffc67074a04,*pp=0x7ffc67074a04
```

Complexification du code

- Très puissante, l'utilisation de pointeurs rend plus difficile le travail du développeur.

Complexification du code

- Très puissante, l'utilisation de pointeurs rend plus difficile le travail du développeur.
- ♡

  Si on ne fait pas attention avec les pointeurs, le programme peut accéder à une zone mémoire qui ne lui est pas allouée ou dans laquelle il ne peut pas écrire.
  Le processeur via le système d'exploitation engendre alors une *erreur de segmentation* (segmentation fault) qui provoque une exception voire plante l'application.

Complexification du code

- Très puissante, l'utilisation de pointeurs rend plus difficile le travail du développeur.
- ♡

  Si on ne fait pas attention avec les pointeurs, le programme peut accéder à une zone mémoire qui ne lui est pas allouée ou dans laquelle il ne peut pas écrire.
  Le processeur via le système d'exploitation engendre alors une *erreur de segmentation* (segmentation fault) qui provoque une exception voire plante l'application.
- ♡

  Si c'est le programmeur qui gère les allocations en mémoire, il ne doit pas oublier de libérer après usage la zone allouée sous peine de *fuite de mémoire* :
  Il s'agit d'un bug qui résulte d'une occupation croissante non contrôlés ou non désirée de la mémoire. DANGER : saturation possible de la RAM.

Un exemple de *segmentation fault*

```c
#include <stdio.h>
#include <stdlib.h>// contient EXIT_SUCCESS

int main(){
  int * variable_entiere ;// déclaration d'un pointeur
   printf (" saisir une valeur : ");
  scanf("%d", variable_entiere );
  return (EXIT_SUCCESS);// indique que tout s'est bien passé
}

```

```
$ ./a.out
entrer une valeur : 23
Erreur de segmentation (core dumped)
```

Un exemple de *segmentation fault*

```c
#include <stdio.h>
#include <stdlib.h>// contient EXIT_SUCCESS

int main(){
  int * variable_entiere ;// déclaration d'un pointeur
   printf (" saisir une valeur : ");
  scanf("%d", variable_entiere );
  return (EXIT_SUCCESS);// indique que tout s'est bien passé
}

```

```
$ ./a.out
entrer une valeur : 23
Erreur de segmentation (core dumped)
```

- Explication : le pointeur `variable_entiere` n'est pas initialisé donc contient n'importe quoi (possiblement, une adresse interdite en écriture).
  Quand `scanf` veut accèder à cette adresse l'OS le lui refuse.

Allouer de la mémoire ♡

Trois manières pour l'allocation de mémoire dans un programme :

- Statiquement, au cours de la compilation lorsque le compilateur lit des mots clés comme `const` ou `static` dans le code source. Adresses mémoires connues à l'initialisation de la mémoire du programme. Ces adresses correspondent à ce qu'on appelle le *segment de données* du programme.

Allouer de la mémoire ♡

Trois manières pour l'allocation de mémoire dans un programme :

- Statiquement, au cours de la compilation lorsque le compilateur lit des mots clés comme `const` ou `static` dans le code source. Adresses mémoires connues à l'initialisation de la mémoire du programme. Ces adresses correspondent à ce qu'on appelle le *segment de données* du programme.
- Dynamiquement, au cours de l'exécution :

Allouer de la mémoire ♡

Trois manières pour l'allocation de mémoire dans un programme :

- Statiquement, au cours de la compilation lorsque le compilateur lit des mots clés comme `const` ou `static` dans le code source. Adresses mémoires connues à l'initialisation de la mémoire du programme. Ces adresses correspondent à ce qu'on appelle le *segment de données* du programme.
- Dynamiquement, au cours de l'exécution :
  - soit de façon automatique sur la *pile d'exécution* : variables locales déclarées dans un bloc d'instructions,

Allouer de la mémoire ♡

Trois manières pour l'allocation de mémoire dans un programme :

- Statiquement, au cours de la compilation lorsque le compilateur lit des mots clés comme `const` ou `static` dans le code source. Adresses mémoires connues à l'initialisation de la mémoire du programme. Ces adresses correspondent à ce qu'on appelle le *segment de données* du programme.
- Dynamiquement, au cours de l'exécution :
  - soit de façon automatique sur la *pile d'exécution* : variables locales déclarées dans un bloc d'instructions,
  - soit à la demande *sur le tas* : en utilisant des fonctions d'allocation de la mémoire.

Allouer de la mémoire ♡

Trois manières pour l'allocation de mémoire dans un programme :

- Statiquement, au cours de la compilation lorsque le compilateur lit des mots clés comme `const` ou `static` dans le code source. Adresses mémoires connues à l'initialisation de la mémoire du programme. Ces adresses correspondent à ce qu'on appelle le *segment de données* du programme.
- Dynamiquement, au cours de l'exécution :
  - soit de façon automatique sur la *pile d'exécution* : variables locales déclarées dans un bloc d'instructions,
  - soit à la demande *sur le tas* : en utilisant des fonctions d'allocation de la mémoire.
- Il y a donc trois zones de données distinctes pour le programme. On affinera leur description dans un cours à venir.

segment de données	var. globales ou var. locales statiques
pile d'exécution	var. locales
tas	allocation dynamique par `malloc`

Allouer de l'espace

- Un pointeur non initialisé est égal à la constante symbolique `NULL` de `<stddef.h>`. Le test `p == NULL` permet de savoir si le pointeur p pointe vers un objet.

Allouer de l'espace

- Un pointeur non initialisé est égal à la constante symbolique `NULL` de `<stddef.h>`. Le test `p == NULL` permet de savoir si le pointeur p pointe vers un objet.
- On a vu comment initialiser un pointeur en lui affectant l'adresse d'une autre variable `int * p =&a;`

Allouer de l'espace

- Un pointeur non initialisé est égal à la constante symbolique `NULL` de `<stddef.h>`. Le test `p == NULL` permet de savoir si le pointeur p pointe vers un objet.
- On a vu comment initialiser un pointeur en lui affectant l'adresse d'une autre variable `int * p =&a;`
- Pour affecter directement une valeur à `*p`, il faut d'abord réserver à `*p` un espace-mémoire de taille adéquate avec `malloc` de `<stdlib.h>`.

Allouer de l'espace

- Un pointeur non initialisé est égal à la constante symbolique `NULL` de `<stddef.h>`. Le test `p == NULL` permet de savoir si le pointeur p pointe vers un objet.
- On a vu comment initialiser un pointeur en lui affectant l'adresse d'une autre variable `int * p =&a;`
- Pour affecter directement une valeur à `*p`, il faut d'abord réserver à `*p` un espace-mémoire de taille adéquate avec `malloc` de `<stdlib.h>`.
- L'adresse de cet espace-mémoire est la valeur de `p`

Allouer de l'espace

- Un pointeur non initialisé est égal à la constante symbolique `NULL` de `<stddef.h>`. Le test `p == NULL` permet de savoir si le pointeur p pointe vers un objet.
- On a vu comment initialiser un pointeur en lui affectant l'adresse d'une autre variable `int * p =&a;`
- Pour affecter directement une valeur à `*p`, il faut d'abord réserver à `*p` un espace-mémoire de taille adéquate avec `malloc` de `<stdlib.h>`.
- L'adresse de cet espace-mémoire est la valeur de `p`
- Cette opération est appelée *allocation dynamique*

La fonction `malloc`

- Syntaxe

```
1 void*  malloc( size_t  size );// allocation  en  nb de  bytes
2
```

La fonction `malloc`

- Syntaxe

```
1 void*  malloc( size_t  size );// allocation  en nb de bytes
2
```

- La valeur retournée est l'adresse du premier octet de la zone mémoire allouée. Si l'allocation n'a pu se réaliser (par manque de mémoire libre), la valeur de retour est la constante **NULL**.

La fonction `malloc`

- Syntaxe

```
1 void* malloc( size_t size );// allocation en nb de bytes
2
```

- La valeur retournée est l'adresse du premier octet de la zone mémoire allouée. Si l'allocation n'a pu se réaliser (par manque de mémoire libre), la valeur de retour est la constante **NULL**.
- Toute utilisation de `malloc` doit être suivie plus tard d'une libération de l'espace réservé

```
1 void free(void *ptr);
```

Le seul paramètre à passer est l'adresse du premier octet de la zone allouée et aucune valeur n'est retournée une fois cette opération réalisée.

Exemple ♡

- Réserver 32 octets et les libérer immédiatement.

Exemple ♡

- Réserver 32 octets et les libérer immédiatement.

```
#include <stdlib.h>// tiré de Wikipedia
int main(){
int * pointeur = malloc(8 * sizeof(int)); //8x4 octets

if(pointeur==NULL)
    printf("L'allocation n'a pu être réalisée\n");
else
{
    printf(" allocation : succès");
    printf("valeur de p=%p\n",p);
    free(pointeur); //Libération des 32 octets
    pointeur = NULL; // Invalidation du pointeur
  }...

```

Exemple ♡

- Réserver 32 octets et les libérer immédiatement.

```c
#include <stdlib.h>// tiré de Wikipedia
int main(){
int * pointeur = malloc(8 * sizeof(int)); //8x4 octets

if(pointeur==NULL)
    printf("L'allocation n'a pu être réalisée\n");
else
{
    printf(" allocation : succès");
    printf("valeur de p=%p\n",p);
    free(pointeur); //Libération des 32 octets
    pointeur = NULL; // Invalidation du pointeur
  }...

```

```
allocation : succès; valeur de pointeur=0x55fb8de5f260
```

Allocation dynamique vs automatique ♡

```c
int i = 3;
int *p;

p = (int*)malloc(sizeof(int));// alloc. dyn. sur le tas
*p = i;// attention : p ne pointe pas sur i

int *r=&i;// alloc. auto. sur la pile

printf("i=%d,*p=%d, *r=%d\n",i,*p,*r);
printf("p=%p, r=%p\n",p,r);

free(p);

```

Allocation dynamique vs automatique ♡

```c
int i = 3;
int *p;

p = (int*)malloc(sizeof(int));// alloc. dyn. sur le tas
*p = i;// attention : p ne pointe pas sur i

int *r=&i;// alloc. auto. sur la pile

printf("i=%d,*p=%d, *r=%d\n",i,*p,*r);
printf("p=%p, r=%p\n",p,r);

free(p);

```

- `p` ne pointe pas sur `i` mais `r` si. Modifier `*p` ne modifie pas `i`, mais modifier `*r` modifie `i`.

Assertions (rappel)

- Pour diverses raisons on peut souhaiter interrompre un programme si une condition n'est pas réalisée (ex : impossibilité d'allouer de la mémoire). Les *assertions* sont alors utiles.

Assertions (rappel)

- Pour diverses raisons on peut souhaiter interrompre un programme si une condition n'est pas réalisée (ex : impossibilité d'allouer de la mémoire). Les *assertions* sont alors utiles.
- Exemple d'assertion :

```
1 #include <assert.h>
2
3 int main() {
4     int i=0;// on veut imposer i>1
5     assert( i > 1 );// si i<=1 : arrêt du pgm
6     printf( "poursuite du pgm");
7     return 0;
8 }
9
```

Assertions (rappel)

- Pour diverses raisons on peut souhaiter interrompre un programme si une condition n'est pas réalisée (ex : impossibilité d'allouer de la mémoire). Les *assertions* sont alors utiles.
- Exemple d'assertion :

```
1 #include <assert.h>
2
3 int main() {
4     int i=0;// on veut imposer i>1
5     assert( i > 1 );// si i<=1 : arrêt du pgm
6     printf( "poursuite du pgm");
7     return 0;
8 }
9
```

- Trace d'exécution :

```
a.out: assertion2.c:8: main: Assertion ‘i > 1‘ failed.
Abandon (core dumped}
```

Assertions (suite)

- On la vu que les assertions servent à se protéger d'un comportement aberrant en arrêtant le programmme.

Assertions (suite)

- On la vu que les assertions servent à se protéger d'un comportement aberrant en arrêtant le programmme.
- Elles servent aussi en phase de conception d'un programme :

Assertions (suite)

- On la vu que les assertions servent à se protéger d'un comportement aberrant en arrêtant le programmme.
- Elles servent aussi en phase de conception d'un programme :
  - Il faut en effet compiler régulièrement (dès qu'on ajoute un bloc d'instructions ?)

Assertions (suite)

- On la vu que les assertions servent à se protéger d'un comportement aberrant en arrêtant le programmme.
- Elles servent aussi en phase de conception d'un programme :
  - Il faut en effet compiler régulièrement (dès qu'on ajoute un bloc d'instructions ?)
  - Si l'écriture du programme n'est pas encore terminée : ajouter des assertions dans les branches non écrites des instructions conditionnelles. Cela permet de compiler (donc de vérifier la syntaxe) et de tester les branches complétées.

```
1  int f(int x){
2      assert (x>0);// on quitte le pgm si x<=0
3      // bloc de code complètement écrit
4      // ne concernant que x>0 et à tester
5          ...
6          return ...
7      }
8   return NimporteQuoi;
9          }
```

Assertion et bon fonctionnement d'une allocation

Exemple d'assertion

- Pour diverses raisons, l'allocation dynamique peut échouer. Le pointeur est alors égal au pointeur `NULL`. On devrait toujours tester si l'allocation s'est bien déroulée.

Assertion et bon fonctionnement d'une allocation

Exemple d'assertion

- Pour diverses raisons, l'allocation dynamique peut échouer. Le pointeur est alors égal au pointeur `NULL`. On devrait toujours tester si l'allocation s'est bien déroulée.
- Il est plus prudent d'arrêter l'exécution du programme en cas d'échec d'allocation. On peut utiliser une *assertion*.

Assertion et bon fonctionnement d'une allocation

Exemple d'assertion

- Pour diverses raisons, l'allocation dynamique peut échouer. Le pointeur est alors égal au pointeur `NULL`. On devrait toujours tester si l'allocation s'est bien déroulée.
- Il est plus prudent d'arrêter l'exécution du programme en cas d'échec d'allocation. On peut utiliser une *assertion*.

```
1 int t[3] = {1.,2.,3.};
2 int * copy = NULL;// on veut copier t dans copy
3
4 /* Allocation de la mémoire */
5 copy = (int *) malloc( 3 * sizeof(int) );
6 assert( copy != NULL ); // si échec, arrêter le pgm
7
```

Arithmétique des pointeurs

(Hors programme) Les opérations valides sur les pointeurs sont

- l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;

Arithmétique des pointeurs

(Hors programme) Les opérations valides sur les pointeurs sont

- l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;

Arithmétique des pointeurs

(Hors programme) Les opérations valides sur les pointeurs sont

- l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
- la différence de deux pointeurs pointant tous deux vers des objets de même type. Le résultat est un entier.

Arithmétique des pointeurs

(Hors programme)

```
1 int main(){
2   int i = 3;
3   int *p1, *p2;
4   p1 = &i;
5   p2 = p1 + 1;
6    printf ("p1 = %p \t p2 = %p \t p2-p1=%ld\n",p1,p2,p2-p1);
7   return 0;}
```

On obtient

```
p1 = 0x7fff37e6dc34  p2 = 0x7fff37e6dc38  p2-p1=1
```

La différence entre les adresses est 4, c'est à dire la taille d'un `int`.
l'expression `p2 - p1` désigne en fait un entier dont la valeur est égale à **(p2 - p1)/sizeof(int)** .

Arithmétique des pointeurs

(Hors programme)

```
1 int main(){
2   double i = 3.;
3   double *p1, *p2;
4   p1 = &i;
5   p2 = p1 - 1;
6    printf ("p1 = %p \t p2 = %p \t p2-p1=%ld\n",p1,p2,p2-p1);
7   return 0;}
```

On obtient

```
$ ./a.out
p1 = 0x7ffcadcfc8e0  p2 = 0x7ffcadcfc8d8  p2-p1=-1
```

La différence entre les adresses est

$$e0_{16} - d8_{16} = 14_{10} \times 16_{10} + 0 - (13_{10} \times 16_{10} + 8_{10}) = 8_{10}$$

C'est la taille d'un `double`

Tableaux ♡

(Rappel)

- Déclaration de tableaux :

```
1 //type nomDuTableau[taille];
2 int tableau[4];// tableau de 4 entiers
3
```

Tableaux ♡

(Rappel)

- Déclaration de tableaux :

```
1 //type nomDuTableau[taille];
2 int tableau[4];// tableau de 4 entiers
3
```

- Déclaration et initialisation :

```
1 int tableau[4] = {1,2,3,4};// les 4 elts de tab sont initialisés
2 float tib[5] = {1.1,2.1}// seuls 2 elts de tib sont initialisés
```

Tableaux ♡

(Rappel)

- Déclaration de tableaux :

```
1 //type nomDuTableau[taille];
2 int tableau[4];// tableau de 4 entiers
3
```

- Déclaration et initialisation :

```
1 int tableau[4] = {1,2,3,4};// les 4 elts de tab sont initialisés
2 float tib[5] = {1.1,2.1}// seuls 2 elts de tib sont initialisés
```

- Accès aux éléments comme en PYTHON avec `tab[i]`

```
1 tab[1] = 23;// l'elt 1 de tab vaut 23
2 printf("tab[%d]=%d",1,tab[1]);
3
```

Accès aux éléments d'un tableau

(Rappel) Considérons `int tab[2];`

- Un nom de tableau comme `tab` désigne, après déclaration, par convention un pointeur constant sur le premier bloc de son premier élément.

Accès aux éléments d'un tableau

(Rappel) Considérons `int tab[2];`

- Un nom de tableau comme `tab` désigne, après déclaration, par convention un pointeur constant sur le premier bloc de son premier élément.
- On peut utiliser un pointeur initialisé à `tab` pour parcourir les éléments du tableau par *arithmétique des pointeurs* (hors programme en CPGE).

Accès aux éléments d'un tableau

(Rappel) Considérons `int tab[2];`

- Un nom de tableau comme `tab` désigne, après déclaration, par convention un pointeur constant sur le premier bloc de son premier élément.
- On peut utiliser un pointeur initialisé à `tab` pour parcourir les éléments du tableau par *arithmétique des pointeurs* (hors programme en CPGE).
- Pour accéder à `tab[2]`, le système ajoute à l'adresse du (premier bloc) de `tab[0]` le produit $2 * 4$

Accès aux éléments d'un tableau

(Rappel) Considérons `int tab[2];`

- Un nom de tableau comme `tab` désigne, après déclaration, par convention un pointeur constant sur le premier bloc de son premier élément.
- On peut utiliser un pointeur initialisé à `tab` pour parcourir les éléments du tableau par *arithmétique des pointeurs* (hors programme en CPGE).
- Pour accéder à `tab[2]`, le système ajoute à l'adresse du (premier bloc) de `tab[0]` le produit $2 * 4$

  le nombre 2 pour la position dans le tableau,

Accès aux éléments d'un tableau

(Rappel) Considérons `int tab[2];`

- Un nom de tableau comme `tab` désigne, après déclaration, par convention un pointeur constant sur le premier bloc de son premier élément.
- On peut utiliser un pointeur initialisé à `tab` pour parcourir les éléments du tableau par *arithmétique des pointeurs* (hors programme en CPGE).
- Pour accéder à `tab[2]`, le système ajoute à l'adresse du (premier bloc) de `tab[0]` le produit $2 * 4$
  le nombre 2 pour la position dans le tableau,
  le nombre 4 pour la taille d'un `int` (4 octets/`int`).

Passage d'un tableau en paramètre d'une fonction ♡

(Rappel)

- La fonction ci-dessous affiche les éléments d'un tableau

```
1 void affiche(int *tableau, int tailleTableau){
2     for (int i = 0; i < tailleTableau ; i++)
3         printf("%d\n", tab[i]);
4 }
5
```

Passage d'un tableau en paramètre d'une fonction ♡

(Rappel)

- La fonction ci-dessous affiche les éléments d'un tableau

```
1 void affiche(int *tableau, int tailleTableau){
2     for (int i = 0; i < tailleTableau ; i++)
3         printf("%d\n", tab[i]);
4 }
5
```

- Les tableaux sont en fait passés en paramètre comme des copies de pointeur sur leur 1er élément.

Passage d'un tableau en paramètre d'une fonction ♡

(Rappel)

- La fonction ci-dessous affiche les éléments d'un tableau

```
1 void affiche(int *tableau, int tailleTableau){
2     for (int i = 0; i < tailleTableau ; i++)
3         printf("%d\n", tab[i]);
4 }
5
```

- Les tableaux sont en fait passés en paramètre comme des copies de pointeur sur leur 1er élément.
- Exemple de **main** :

```
1 int main(){
2     int tab[4] = {1, 2, -12};
3     // afficher le contenu du tableau
4     affiche(tab, 4);
5     return 0;
6 }
7
```

Notion de *lvalue*

- Une *lvalue* (left value) est une expression désignant un objet modifiable.

Notion de *lvalue*

- Une *lvalue* (left value) est une expression désignant un objet modifiable.
- Les expressions qui sont des *lvalue* :

Notion de *lvalue*

- Une *lvalue* (left value) est une expression désignant un objet modifiable.
- Les expressions qui sont des *lvalue* :
  - identificateur de variable :

Notion de *lvalue*

- Une *lvalue* (left value) est une expression désignant un objet modifiable.
- Les expressions qui sont des *lvalue* :
  - identificateur de variable :
    - N'ayant pas reçu le qualifieur `const`

Notion de *lvalue*

- Une *lvalue* (left value) est une expression désignant un objet modifiable.
- Les expressions qui sont des *lvalue* :
  - identificateur de variable :
    - N'ayant pas reçu le qualifieur `const`
    - Autre qu'un nom de tableau

Notion de *lvalue*

- Une *lvalue* (left value) est une expression désignant un objet modifiable.
- Les expressions qui sont des *lvalue* :
  - identificateur de variable :
    - N'ayant pas reçu le qualifieur `const`
    - Autre qu'un nom de tableau
    - dans le cas d'une variable de type structure ou union, celle-ci ne doit pas comporter de champs constant.

Notion de *lvalue*

- Une *lvalue* (left value) est une expression désignant un objet modifiable.
- Les expressions qui sont des *lvalue* :
  - identificateur de variable :
    - N'ayant pas reçu le qualifieur `const`
    - Autre qu'un nom de tableau
    - dans le cas d'une variable de type structure ou union, celle-ci ne doit pas comporter de champs constant.
  - Expression de la forme `*p` ou `*(adr)` : `p` (resp. `adr`) étant une variable (resp.expression) de type pointeur sur un objet non constant.

Notion de *lvalue*

- Une *lvalue* (left value) est une expression désignant un objet modifiable.
- Les expressions qui sont des *lvalue* :
  - identificateur de variable :
    - N'ayant pas reçu le qualifieur `const`
    - Autre qu'un nom de tableau
    - dans le cas d'une variable de type structure ou union, celle-ci ne doit pas comporter de champs constant.
  - Expression de la forme `*p` ou `*(adr)` : `p` (resp. `adr`) étant une variable (resp.expression) de type pointeur sur un objet non constant.
  - Élément de tableau :

Notion de *lvalue*

- Une *lvalue* (left value) est une expression désignant un objet modifiable.
- Les expressions qui sont des *lvalue* :
  - identificateur de variable :
    - N'ayant pas reçu le qualifieur `const`
    - Autre qu'un nom de tableau
    - dans le cas d'une variable de type structure ou union, celle-ci ne doit pas comporter de champs constant.
  - Expression de la forme `*p` ou `*(adr)` : `p` (resp. `adr`) étant une variable (resp.expression) de type pointeur sur un objet non constant.
  - Élément de tableau :
    - autre que tableau (cas des tableaux multi-indice)

Notion de *lvalue*

- Une *lvalue* (left value) est une expression désignant un objet modifiable.
- Les expressions qui sont des *lvalue* :
  - identificateur de variable :
    - N'ayant pas reçu le qualifieur `const`
    - Autre qu'un nom de tableau
    - dans le cas d'une variable de type structure ou union, celle-ci ne doit pas comporter de champs constant.
  - Expression de la forme `*p` ou `*(adr)` : `p` (resp. `adr`) étant une variable (resp.expression) de type pointeur sur un objet non constant.
  - Élément de tableau :
    - autre que tableau (cas des tableaux multi-indice)
    - autre que structure ou union avec champ constant

Notion de *lvalue*

- Une *lvalue* (left value) est une expression désignant un objet modifiable.
- Les expressions qui sont des *lvalue* :
  - identificateur de variable :
    - N'ayant pas reçu le qualifieur `const`
    - Autre qu'un nom de tableau
    - dans le cas d'une variable de type structure ou union, celle-ci ne doit pas comporter de champs constant.
  - Expression de la forme `*p` ou `*(adr)` : `p` (resp. `adr`) étant une variable (resp.expression) de type pointeur sur un objet non constant.
  - Élément de tableau :
    - autre que tableau (cas des tableaux multi-indice)
    - autre que structure ou union avec champ constant
  - Champ de structure ou d'union : même précisions que pour les tableaux

Un tableau n'est pas une *lvalue*

- Considérons : `int tab[4];` Un nom de tableau comme `tab` utilisé plus loin dans le programme désigne en fait un pointeur constant (non modifiable) dont la valeur est l'adresse du (premier bloc du) `tab[0]`. Après déclaration, `tab` a pour valeur `&tab[0]`.

Un tableau n'est pas une *lvalue*

- Considérons : `int tab[4];` Un nom de tableau comme `tab` utilisé plus loin dans le programme désigne en fait un pointeur constant (non modifiable) dont la valeur est l'adresse du (premier bloc du) `tab[0]`. Après déclaration, `tab` a pour valeur `&tab[0]`.
- `tab=tab2` soulève une erreur à la compilation (les noms de tableau ne sont pas des *lvalue*).

Les fonctions d'allocation en CPGE

(D'après Koors)

`malloc` de `<stdlib.h>`

```
1 void * malloc(  size_t  memorySize );
```

- Cette fonction permet d'allouer un bloc de mémoire dans le tas (le heap en anglais).

Les fonctions d'allocation en CPGE

(D'après Koors)

`malloc` de `<stdlib.h>`

```
1 void * malloc( size_t memorySize );
```

- Cette fonction permet d'allouer un bloc de mémoire dans le tas (le heap en anglais).
- Attention : la mémoire allouée dynamiquement n'est pas automatiquement relachée. Il faudra donc, après utilisation, libérer ce bloc de mémoire via un appel à la fonction `free`.

Les fonctions d'allocation en CPGE

(D'après Koors)

`calloc` de `<stdlib.h>`

```
1    void * calloc( size_t elementCount, size_t elementSize );
2
```

Cette fonction alloue un bloc de mémoire en initialisant tous ces octets à la valeur 0. Bien que relativement proche de la fonction malloc, deux aspects les différencient :

- L'initialisation : `calloc` met tous les octets du bloc à la valeur 0 alors que malloc ne modifie pas la zone de mémoire.

Les fonctions d'allocation en CPGE

(D'après Koors)

`calloc` de `<stdlib.h>`

```
1    void * calloc( size_t elementCount, size_t elementSize );
2
```

Cette fonction alloue un bloc de mémoire en initialisant tous ces octets à la valeur 0. Bien que relativement proche de la fonction malloc, deux aspects les différencient :

- L'initialisation : `calloc` met tous les octets du bloc à la valeur 0 alors que malloc ne modifie pas la zone de mémoire.
- Les paramètres d'appels : `calloc` requière deux paramètres (le nombre d'éléments consécutifs à allouer et la taille d'un élément) alors que `malloc` attend la taille totale du bloc à allouer.

La fonction de libération

- `free` de **stdlib.h**

```
1 void free( void * pointer );
2
```

La fonction de libération

- `free` de **stdlib.h**

```
1 void free( void * pointer );
2
```

- La fonction `free` libère un bloc de mémoire alloué dynamiquement dans le tas (le heap, en anglais), via un appel à la fonction malloc (ou tout autre fonction d'allocation mémoire)

La fonction de libération

- `free` de **stdlib.h**

```
1 void free( void * pointer );
2
```

- La fonction `free` libère un bloc de mémoire alloué dynamiquement dans le tas (le heap, en anglais), via un appel à la fonction malloc (ou tout autre fonction d'allocation mémoire)
- ne jamais désallouer avec la fonction `free` un bloc de mémoire obtenu autrement que par une fonction d'allocation comme `malloc, calloc`

Squelette d'allocation ♡

- Puisqu'un nom de tableau désigne un pointeur constant :

Squelette d'allocation ♡

- Puisqu'un nom de tableau désigne un pointeur constant :
  - On ne peut pas créer de tableaux dont la taille est une variable du programme (sauf à utiliser des VLA),

Squelette d'allocation ♡

- Puisqu'un nom de tableau désigne un pointeur constant :
  - On ne peut pas créer de tableaux dont la taille est une variable du programme (sauf à utiliser des VLA),
  - on ne peut pas créer de tableaux bidimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments.

Squelette d'allocation ♡

- Puisqu'un nom de tableau désigne un pointeur constant :
  - On ne peut pas créer de tableaux dont la taille est une variable du programme (sauf à utiliser des VLA),
  - on ne peut pas créer de tableaux bidimensionnels dont les lignes n'ont pas toutes le même nombre d'éléments.
- Plutôt que de déclarer les tableaux comme des pointeurs constants, on peut manipuler des pointeurs alloués dynamiquement

```
1  #include <stdlib.h>// exemple Anne Canteaut
2  int main(){
3    int n;// la valeur sera par exemple saisie avec un scanf
4    int *tab;
5      ...
6     tab = (int*)malloc(n * sizeof(int));
7     // cases de tab non initialis ées
8      ...
9    free(tab);
10 }
11
```

Squelette d'allocation

- Avec `calloc`, le principe est le même mais en plus les cases du tableau sont initialisées à 0 :

```
1  #include <stdlib.h>// exemple Anne Canteaut
2  int main(){
3    int n;// la valeur sera par exemple saisie avec un scanf
4    int *tab;
5      ...
6     tab = (int*)calloc(n , sizeof(int));// syntaxe diff érente
7     // cases de tab initialis ées à 0
8      ...
9    free(tab);
10 }
11
```

Squelette d'allocation

- Avec `calloc`, le principe est le même mais en plus les cases du tableau sont initialisées à 0 :

```
1  #include <stdlib.h>// exemple Anne Canteaut
2  int main(){
3    int n;// la valeur sera par exemple saisie avec un scanf
4    int *tab;
5      ...
6     tab = (int*)calloc(n , sizeof(int));// syntaxe différente
7     // cases de tab initialisées à 0
8      ...
9    free(tab);
10 }
11
```

- Dans tous les cas, les éléments de `tab` sont manipulés avec l'opérateur d'indexation `[]`, exactement comme pour les tableaux.

Commentaire

- Dans le transparent précédent, `tab` n'est pas un tableau statique, c'est un pointeur.

Commentaire

- Dans le transparent précédent, `tab` n'est pas un tableau statique, c'est un pointeur.
- Mais on l'utilise comme un tableau : en particulier `tab[2]` désigne l'élément situé à l'adresse `&tab[0]+2*4`

Commentaire

- Dans le transparent précédent, `tab` n'est pas un tableau statique, c'est un pointeur.
- Mais on l'utilise comme un tableau : en particulier `tab[2]` désigne l'élément situé à l'adresse `&tab[0]+2*4`
- de plus, `tab` n'est pas un pointeur constant : c'est une *lvalue*. On peut écrire par exemple `tab++` ou `tab = &n`.

Tableau VS pointeur ♡

Les deux différences principales entre un tableau et un pointeur sont

- un pointeur devrait toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse, par exemple `p = &i` ;

Tableau VS pointeur ♡

Les deux différences principales entre un tableau et un pointeur sont

- un pointeur devrait toujours être initialisé, soit par une allocation dynamique, soit par affectation d'une expression adresse, par exemple `p = &i` ;
- un tableau n'est pas une *lvalue* ; il ne peut donc pas figurer à gauche d'un opérateur d'affectation. En particulier, un tableau ne supporte pas l'arithmétique (on ne peut pas écrire `tab++;`).

Tableaux à plusieurs dimensions ♡

- Un tableau à deux dimensions est, par définition, un tableau de tableaux.
  Après la déclaration, le nom du tableau est utilisé comme un pointeur constant vers un pointeur constant.

Tableaux à plusieurs dimensions ♡

- Un tableau à deux dimensions est, par définition, un tableau de tableaux.
  Après la déclaration, le nom du tableau est utilisé comme un pointeur constant vers un pointeur constant.
- Déclaration, initialisation et usage d'un tableau bi-dimensionnel :

```
1 int main(){
2   int mat[2][3]={{1,20,300},{4,50,600}};
3   for (int i=0; i<2; i++){
4       for (int j=0; j<3; j++) {printf("%3d ",mat[i][j]);}
5       printf ("\n");
6     }
7   return 0;
8 }
9
```

Tableaux à plusieurs dimensions ♡

- Un tableau à deux dimensions est, par définition, un tableau de tableaux.
  Après la déclaration, le nom du tableau est utilisé comme un pointeur constant vers un pointeur constant.
- Déclaration, initialisation et usage d'un tableau bi-dimensionnel :

```
1 int main(){
2   int mat[2][3]={{1,20,300},{4,50,600}};
3   for (int i=0; i<2; i++){
4       for (int j=0; j<3; j++) {printf("%3d ",mat[i][j]);}
5       printf ("\n");
6     }
7   return 0;
8 }
9
```

- Attention : les éléments de `mat` (6 ici) sont rangés consécutivement en mémoire. Il n'y a pas de « grille » dans la mémoire.

Tableaux à plusieurs dimensions (suite) ♡

- Avec l'exemple précédent, on obtient

```
 10 2 300
400 500 6
```

Tableaux à plusieurs dimensions (suite) ♡

- Avec l'exemple précédent, on obtient

```
 10 2 300
400 500 6
```

- Avec

```
1 int mat[N][M];
```

Tableaux à plusieurs dimensions (suite) ♡

- Avec l'exemple précédent, on obtient

```
 10 2 300
400 500 6
```

- Avec

```
1 int mat[N][M];
```

  - `mat` est utilisé comme un pointeur (constant) qui pointe vers un pointeur (constant) de type pointeur d'entiers. Le tableau `mat` est une adresse invariante égale à celle du premier élément : `&mat[0][0]`.

Tableaux à plusieurs dimensions (suite) ♡

- Avec l'exemple précédent, on obtient

```
 10 2 300
400 500 6
```

- Avec

```
1 int mat[N][M];
```

  - `mat` est utilisé comme un pointeur (constant) qui pointe vers un pointeur (constant) de type pointeur d'entiers. Le tableau `mat` est une adresse invariante égale à celle du premier élément : `&mat[0][0]`.
  - De même `mat[i]`, pour `i` entre 0 et `N-1`, est un pointeur constant vers un objet de type entier, qui est le premier élément de la « ligne » d'indice `i`. `mat[i]` a donc une valeur constante qui est égale à `&mat[i][0]`.

Tableaux à plusieurs dimensions (suite) ♡

- Avec l'exemple précédent, on obtient

```
 10 2 300
400 500 6
```

- Avec

```
1 int mat[N][M];
```

  - `mat` est utilisé comme un pointeur (constant) qui pointe vers un pointeur (constant) de type pointeur d'entiers. Le tableau `mat` est une adresse invariante égale à celle du premier élément : `&mat[0][0]`.
  - De même `mat[i]`, pour `i` entre 0 et `N-1`, est un pointeur constant vers un objet de type entier, qui est le premier élément de la « ligne » d'indice `i`. `mat[i]` a donc une valeur constante qui est égale à `&mat[i][0]`.
  - `mat, mat[0]` et `&mat[0][0]` désignent la même adresse.

Tableaux à plusieurs dimensions (suite)

Exercice

(Hors programme) Utiliser la propriété de contiguïté en mémoire pour écrire une fonction `void display(int n, int m, int t[n][m])` qui affiche sur une ligne tous les coefficients du tableau bi-dimensionnel `t`. Une seule boucle est autorisée.

Dans `main`, écrivons :

```
1 int t[3][2]={{1,2},{3,4},{5,6}};
2 display(3,2,t);
```

Après compilation/exécution :

```
1,2,3,4,5,6,
```

Allocation dynamique de « matrice »♡

- On déclare un pointeur qui pointe sur un objet de type `type *` (deux dimensions) de la même manière qu'un pointeur avec

```
1 type **nom-du-pointeur;
2
```

Allocation dynamique de « matrice »♡

- On déclare un pointeur qui pointe sur un objet de type `type *` (deux dimensions) de la même manière qu'un pointeur avec

```
type **nom-du-pointeur;
```

- Le squelette du programme devient :

```
int k, n;
int **tab;
 ...
//tab. de k pointeurs d'entiers
tab = (int**)malloc(k * sizeof(int*));
for (int i = 0; i < k; i++)
  // tab. de n entiers
  tab[i] = (int*)malloc(n * sizeof(int));
     ....
for (int i = 0; i < k; i++)
  free(tab[i]);// libération espaces des lignes
free(tab);// libération espace
```

Allocation dynamique de matrice ♡

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

int main(){
  int n=2, m=3;
  int **tab;
  tab = (int**)malloc(n * sizeof(int*));
  for (int i = 0; i < n; i++){
    tab[i] = (int*)malloc(m * sizeof(int));// tab. d'entiers
    for (int j=0; j<m; j++)// on peut s'en passer avec calloc
      tab[i][j] = 0;
  }
  ...
  for (int i = 0; i < n; i++)
    free(tab[i]);// libération espaces des lignes
  free(tab);// libération espace
  return 0;
}
```

Tableaux statiques 2D VS pointeur de pointeur

- Considérons le pointeur de pointeur utilisé comme un tableau à deux dimensions dans le transparent précédent : `int ** tab = malloc(2 * sizeof(int *))`. Chaque ≪ ligne ≫ est déclarée par `tab[i] = (int*)malloc(3 * sizeof(int));`.

Tableaux statiques 2D VS pointeur de pointeur

- Considérons le pointeur de pointeur utilisé comme un tableau à deux dimensions dans le transparent précédent : `int ** tab = malloc(2 * sizeof(int *))`. Chaque ≪ ligne ≫ est déclarée par `tab[i] = (int*)malloc(3 * sizeof(int));`.
- Considérons un tableau statique à deux dimensions : `int t[2][3]`.

Tableaux statiques 2D VS pointeur de pointeur

- Considérons le pointeur de pointeur utilisé comme un tableau à deux dimensions dans le transparent précédent : `int ** tab = malloc(2 * sizeof(int *))`. Chaque « ligne » est déclarée par `tab[i] = (int*)malloc(3 * sizeof(int));`.
- Considérons un tableau statique à deux dimensions : `int t[2][3]`.
- Dans `tab`, les éléments sont rangés par « lignes », deux « lignes » n'étant pas nécessairement consécutives en mémoire.

Tableaux statiques 2D VS pointeur de pointeur

- Considérons le pointeur de pointeur utilisé comme un tableau à deux dimensions dans le transparent précédent : `int ** tab = malloc(2 * sizeof(int *))`. Chaque « ligne » est déclarée par `tab[i] = (int*)malloc(3 * sizeof(int));`.
- Considérons un tableau statique à deux dimensions : `int t[2][3]`.
- Dans `tab`, les éléments sont rangés par « lignes », deux « lignes » n'étant pas nécessairement consécutives en mémoire.
- Dans `t`, les éléments sont tous contigus en mémoire. Le dernier octet du dernier élément de la « ligne i » est voisin du premier octet du premier élément de la « ligne $i+1$ ».

Tableaux statiques 2D VS pointeur de pointeur

Passage en paramètre

- Les deux déclarations suivantes ne sont pas équivalentes :

```
1 void foo(int n, int m, int ** tab);
2 void moo(int n, int m, int t[n][m]);
```

Tableaux statiques 2D VS pointeur de pointeur

Passage en paramètre

- Les deux déclarations suivantes ne sont pas équivalentes :

```
1 void foo(int n, int m, int ** tab);
2 void moo(int n, int m, int t[n][m]);
```

- Pour accéder à `t[i][j]`, `moo` calcule `*(t + i * m + j)`.
  Rappel : le nom d'un tableau est un alias vers la localisation en mémoire de ce tableau : c.a.d. `&t[0][0]`.

Tableaux statiques 2D VS pointeur de pointeur

Passage en paramètre

- Les deux déclarations suivantes ne sont pas équivalentes :

```
1 void foo(int n, int m, int ** tab);
2 void moo(int n, int m, int t[n][m]);
```

- Pour accéder à `t[i][j]`, `moo` calcule `*(t + i * m + j)`. Rappel : le nom d'un tableau est un alias vers la localisation en mémoire de ce tableau : c.a.d. `&t[0][0]`.
- Pour accéder à `tab[i][j]`, `foo` calcule `*(*(tab + i) + j)` (observer le double déréférencement).

Tableaux statiques 2D VS pointeur de pointeur

Passage en paramètre

- Les deux déclarations suivantes ne sont pas équivalentes :

```
1 void foo(int n, int m, int ** tab);
2 void moo(int n, int m, int t[n][m]);
```

- Pour accéder à `t[i][j]`, `moo` calcule `*(t + i * m + j)`. Rappel : le nom d'un tableau est un alias vers la localisation en mémoire de ce tableau : c.a.d. `&t[0][0]`.
- Pour accéder à `tab[i][j]`, `foo` calcule `*(*(tab + i) + j)` (observer le double déréférencement).
- Les appels `foo(2,3,t)` et `moo(2,3,tab)` risquent tous deux de mener à une erreur de segmentation (seg fault).

Tableaux statiques 2D VS pointeur de pointeur

Passage en paramètre

- Les deux déclarations suivantes ne sont pas équivalentes :

```
1 void foo(int n, int m, int ** tab);
2 void moo(int n, int m, int t[n][m]);
```

- Pour accéder à `t[i][j]`, `moo` calcule `*(t + i * m + j)`. Rappel : le nom d'un tableau est un alias vers la localisation en mémoire de ce tableau : c.a.d. `&t[0][0]`.
- Pour accéder à `tab[i][j]`, `foo` calcule `*(*(tab + i) + j)` (observer le double déréférencement).
- Les appels `foo(2,3,t)` et `moo(2,3,tab)` risquent tous deux de mener à une erreur de segmentation (seg fault).
  - Dans `foo(2,3,t)`, la fonction traite `t+i` comme une adresse à déréférencer ; ce qu'elle n'est pas.

Tableaux statiques 2D VS pointeur de pointeur

Passage en paramètre

- Les deux déclarations suivantes ne sont pas équivalentes :

```
1 void foo(int n, int m, int ** tab);
2 void moo(int n, int m, int t[n][m]);
```

- Pour accéder à `t[i][j]`, `moo` calcule `*(t + i * m + j)`. Rappel : le nom d'un tableau est un alias vers la localisation en mémoire de ce tableau : c.a.d. `&t[0][0]`.
- Pour accéder à `tab[i][j]`, `foo` calcule `*(*(tab + i) + j)` (observer le double déréférencement).
- Les appels `foo(2,3,t)` et `moo(2,3,tab)` risquent tous deux de mener à une erreur de segmentation (seg fault).
  - Dans `foo(2,3,t)`, la fonction traite `t+i` comme une adresse à déréférencer ; ce qu'elle n'est pas.
  - Dans `moo(2,3,tab)`, la fonction considère que les éléments `tab[i][m-1]` et `tab[i+1][0]` sont contigus ; ce qu'ils ne sont pas.

Allocation dynamique de matrice (suite)

Pour créer un tableau à 3 dimensions, utiliser un pointeur vers un objet de type `type **` :

```
1 type ***nom-du-pointeur;
2
```

Chaîne de caractères

- Une chaîne de caractères est un tableau à une dimension d'objets de type char, se terminant par le caractère nul `'\0'`.

Chaîne de caractères

- Une chaîne de caractères est un tableau à une dimension d'objets de type char, se terminant par le caractère nul `'\0'`.
- On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type `char`.

Chaîne de caractères

- Une chaîne de caractères est un tableau à une dimension d'objets de type char, se terminant par le caractère nul `'\0'`.
- On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type `char`.
- Les *constantes chaînes* sont notées entre doubles quotes comme `"hello"` mais le caractère de fin de chaîne n'apparaît pas explicitement.

Chaîne de caractères

- Une chaîne de caractères est un tableau à une dimension d'objets de type char, se terminant par le caractère nul `'\0'`.
- On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type `char`.
- Les *constantes chaînes* sont notées entre doubles quotes comme `"hello"` mais le caractère de fin de chaîne n'apparaît pas explicitement.
- Les constantes chaînes sont toujours placées en mémoire statique. Elles sont immuables.

Chaîne de caractères

- Une chaîne de caractères est un tableau à une dimension d'objets de type char, se terminant par le caractère nul `'\0'`.
- On peut donc manipuler toute chaîne de caractères à l'aide d'un pointeur sur un objet de type `char`.
- Les *constantes chaînes* sont notées entre doubles quotes comme `"hello"` mais le caractère de fin de chaîne n'apparaît pas explicitement.
- Les constantes chaînes sont toujours placées en mémoire statique. Elles sont immuables.
- Le caractère immuable permet au compilateur d'allouer une seule occurrence de chaque chaîne distincte. En clair, dans

```
1 printf (" hello");
2 printf (" hello");
3
```

les deux occurrences de **hello** sont stockées au même endroit en mémoire.

Les chaînes de caractères comme tableaux de `char` ♡

- Déclaration comme pointeur (constant) :

```
1 char chaine[5];
2
```

Les chaînes de caractères comme tableaux de `char` ♡

- Déclaration comme pointeur (constant) :

```
1 char chaine[5];
2
```

- Le mot `hello` est déclaré et initialisé avec

```
1 char chaine[6] = {'h','e','l','l','o','\0'};
```

Le *caractère* `'\0'` représente la fin du mot. Sa présence est impérative. Ainsi « hello » nécessite 6 caractères. Les guillemets sont simples.

Les chaînes de caractères comme tableaux de `char` ♡

- Déclaration comme pointeur (constant) :

```
1 char chaine[5];
2
```

- Le mot `hello` est déclaré et initialisé avec

```
1 char chaine[6] = {'h','e','l','l','o','\0'};
```

  Le *caractère* `'\0'` représente la fin du mot. Sa présence est impérative. Ainsi « hello » nécessite 6 caractères. Les guillemets sont simples.

- `"h"` représente la chaîne de caractères `{'h',\0}` et `'h'` ben... le caractère « `h` » !

Les chaînes de caractères comme tableaux de `char` ♡

- Déclaration comme pointeur (constant) :

```
1 char chaine[5];
2
```

- Le mot `hello` est déclaré et initialisé avec

```
1 char chaine[6] = {'h','e','l','l','o','\0'};
```

Le *caractère* `'\0'` représente la fin du mot. Sa présence est impérative. Ainsi « hello » nécessite 6 caractères. Les guillemets sont simples.

- `"h"` représente la chaîne de caractères `{'h',\0}` et `'h'` ben... le caractère « `h` » !
- Initialisation plus rapide :

```
1 char chaine[] = "Salut";// taille automatiqu. calculée
2 // caractère de fin de chaîne automatiquement ajouté.
3 // Observer les doubles quote.
4 // chaine est de lg 6 !!
```

Accès aux éléments ♡

- Comme en PYTHON, l'accès aux constituants de la chaîne se fait avec l'opérateur `[]` .

Accès aux éléments ♡

- Comme en PYTHON, l'accès aux constituants de la chaîne se fait avec l'opérateur `[]` .
- Lecture :

```
1  char chaine[] = "hello";
2  int i =0;
3  while(chaine[i]!='\0'){// lecture
4      printf("%c",chaine[i]);
5      i++; }
```

Affiche « hello ».

Accès aux éléments ♡

- Comme en PYTHON, l'accès aux constituants de la chaîne se fait avec l'opérateur `[]` .
- Lecture :

```
1  char chaine[] = "hello";
2  int i =0;
3  while(chaine[i]!='\0'){// lecture
4      printf ("%c",chaine[i]);
5      i++; }
```

Affiche « hello ».

- Modification :

```
1 i=0;
2  while(chaine[i]!='\0'){// codage de César
3     chaine[i] = chaine[i]+1;
4     i++;}
5
```

La chaîne est devenue `ifmmp` .

Affichage et saisie au clavier ♡

- Le spécifieur de format `%s` vaut pour `printf` et `scanf` :

```
1 int main(){
2   char chaine[] = "hello";
3    printf ("%s\n", chaine);
4    printf ("entrer un mot de 5 lettres sans espace : ");
5   scanf("%s", chaine);// chaine est un pointeur
6    printf ("%s\n", chaine);
7   return 0;}
8
```

Affichage et saisie au clavier ♡

- Le spécifieur de format `%s` vaut pour `printf` et `scanf` :

```
1 int main(){
2   char chaine[] = "hello";
3    printf ("%s\n", chaine);
4    printf ("entrer un mot de 5 lettres sans espace : ");
5   scanf("%s", chaine);// chaine est un pointeur
6    printf ("%s\n", chaine);
7   return 0;}
8
```

- On obtient :

```
hello
entrer un mot de 5 lettres sans espace : salut
salut
```

Longueur

- On dispose d'un moyen pour calculer la longueur d'une chaîne de caractères : on itère jusqu'à tomber sur le caractèrenul `'\0'`.

Longueur

- On dispose d'un moyen pour calculer la longueur d'une chaîne de caractères : on itère jusqu'à tomber sur le caractèrenul `'\0'`.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int i;
6    char *chaine;
7
8    chaine = "toto et gogo";
9    for (i = 0; chaine[i]!='\0'; i++);// observer le ";"
10
11    printf ("nombre de caracteres = %d\n",i);
12   return 0;
13 }
14
```

Longueur

- On dispose d'un moyen pour calculer la longueur d'une chaîne de caractères : on itère jusqu'à tomber sur le caractèrenul `'\0'`.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int i;
6    char *chaine;
7
8    chaine = "toto et gogo";
9    for (i = 0; chaine[i]!='\0'; i++);// observer le ";"
10
11    printf ("nombre de caracteres = %d\n",i);
12   return 0;
13 }
14
```

- Avec cette méthode, la complexité est linéaire : il y a n étapes pour calculer la longueur (si $n = |\textbf{chaine}|$).

Longueur

- Un autre moyen de calculer la longueur d'une chaîne de caractères est d'utiliser la fonction `strlen` de `<string.h>`.

Longueur

- Un autre moyen de calculer la longueur d'une chaîne de caractères est d'utiliser la fonction `strlen` de `<string.h>`.
- Son prototype est le suivant

```
1       size_t  strlen ( const char * theString );
2
```

Elle prend donc en paramètre un pointeur sur un `char` constant (on ne modifie pas la chaîne).

Longueur

- Un autre moyen de calculer la longueur d'une chaîne de caractères est d'utiliser la fonction `strlen` de `<string.h>`.
- Son prototype est le suivant

```
1       size_t   strlen ( const char * theString );
2
```

Elle prend donc en paramètre un pointeur sur un `char` constant (on ne modifie pas la chaîne).

- On l'utilise ainsi

```
1 #include <stdio.h>
2 # include <string.h>
3
4 int  main(){
5     ...
6     printf ("nombre de caracteres = %zu\n",strlen(chaine));
7     ...}
8
```

Longueur

- Un autre moyen de calculer la longueur d'une chaîne de caractères est d'utiliser la fonction `strlen` de `<string.h>`.
- Son prototype est le suivant

```
1       size_t  strlen ( const char * theString );
2
```

Elle prend donc en paramètre un pointeur sur un `char` constant (on ne modifie pas la chaîne).

- On l'utilise ainsi

```
1 #include <stdio.h>
2 # include <string.h>
3
4 int main(){
5     ...
6     printf ("nombre de caracteres = %zu\n",strlen(chaine));
7     ...}
8
```

- Mais la complexité reste linéaire !

Comparer

- `int strcmp( const char * first, const char * second );`
  de **string.h**

Comparer

- `int strcmp( const char * first, const char * second );` de **string.h**
- Compare les deux chaînes `first, second`

Comparer

- `int strcmp( const char * first, const char * second );` de **string.h**
- Compare les deux chaînes `first, second`
- Si la valeur de retour est nulle, les chaînes sont égales

Comparer

- `int strcmp( const char * first, const char * second );` de **string.h**
- Compare les deux chaînes `first, second`
- Si la valeur de retour est nulle, les chaînes sont égales
- Si la valeur renvoyée est négative, le premier caractère qui ne correspond pas a une valeur inférieure dans `first` que dans `second`

Comparer

- `int strcmp( const char * first, const char * second );` de **string.h**
- Compare les deux chaînes `first, second`
- Si la valeur de retour est nulle, les chaînes sont égales
- Si la valeur renvoyée est négative, le premier caractère qui ne correspond pas a une valeur inférieure dans `first` que dans `second`
- Si la valeur renvoyée est positive, le premier caractère qui ne correspond pas a une valeur supérieure dans `first` que dans `second` .

Comparer

- `int strcmp( const char * first, const char * second );` de **string.h**
- Compare les deux chaînes `first, second`
- Si la valeur de retour est nulle, les chaînes sont égales
- Si la valeur renvoyée est négative, le premier caractère qui ne correspond pas a une valeur inférieure dans `first` que dans `second`
- Si la valeur renvoyée est positive, le premier caractère qui ne correspond pas a une valeur supérieure dans `first` que dans `second` .
- La variante `strncmp(first,second,n)` ne compare que les *n* premiers caractères des chaînes.

Comparer

- `int strcmp( const char * first, const char * second );` de **string.h**
- Compare les deux chaînes `first, second`
- Si la valeur de retour est nulle, les chaînes sont égales
- Si la valeur renvoyée est négative, le premier caractère qui ne correspond pas a une valeur inférieure dans `first` que dans `second`
- Si la valeur renvoyée est positive, le premier caractère qui ne correspond pas a une valeur supérieure dans `first` que dans `second` .
- La variante `strncmp(first,second,n)` ne compare que les *n* premiers caractères des chaînes.
- `first==second` ne doit être utilisé que pour des chaînes de caractères littérales (dans le doute, on évite !)

Convertir en entier

- `int atoi( const char * theString );` de **strlib**

Convertir en entier

- `int atoi( const char * theString );` de **strlib**
- Cette fonction permet de transformer une chaîne de caractères, représentant une valeur entière comme `"12356"`, en une valeur numérique de type `int`. Le terme d'≪ atoi ≫ est un acronyme signifiant : ASCII to integer.

Convertir en entier

- `int atoi( const char * theString );` de **strlib**
- Cette fonction permet de transformer une chaîne de caractères, représentant une valeur entière comme `"12356"`, en une valeur numérique de type `int`. Le terme d'≪ atoi ≫ est un acronyme signifiant : ASCII to integer.
- Retourne la valeur 0 si la chaîne de caractères ne contient pas une représentation de valeur numérique.
  Il n'est donc pas possible de distinguer la chaîne "0" d'une chaîne ne contenant pas un nombre entier.
  L'utilisation de la fonction `strtol` permet bien de distinguer les deux cas.

Modifier une constante chaîne

- La norme n'indique pas si les constantes chaînes comme `"hello"` sont modifiables ou non. Juste qu'elles sont placées en mémoire statique.

Modifier une constante chaîne

- La norme n'indique pas si les constantes chaînes comme `"hello"` sont modifiables ou non. Juste qu'elles sont placées en mémoire statique.
- Pour imposer un caractère immuable indépendamment des implémentations, préférer :

```
1    const char * a = "hello";
2
```

Concaténation ♡

- Pour concaténer deux chaînes de caractères, allouer à une troisième chaîne autant de place que la somme des longueurs.
- Parcourir l'une après l'autre les deux chaînes et entrer leurs caractères dans le résultat.

```
1  int main()
2  {// exemple Anne Canteaut
3    char *chaine1, *chaine2, *res;// 3 pointeurs
4
5    chaine1 = "chaine ";
6    chaine2 = "de caracteres";
7     size_t lgc1 = strlen(chaine1), lgc2 = strlen(chaine2);
8    res = (char*)malloc((lgc1+lgc2+1) * sizeof(char));
9    for (int i = 0; i < lgc1; i++)
10     res[i] = chaine1[i];
11   for (int i = lgc1; i < lgc1+lgc2; i++)
12     res[i]= chaine2[i-lgc1];
13   res[lgc1+lgc2]="\0";
14    printf ("%s\n",res); free(res);
15   return 0;
```

Tableau de constantes chaînes

- Une constante chaîne est convertie par le compilateur en un pointeur sur son premier caractère. On peut donc facilement constituer un tableau de constantes chaînes.

Tableau de constantes chaînes

- Une constante chaîne est convertie par le compilateur en un pointeur sur son premier caractère. On peut donc facilement constituer un tableau de constantes chaînes.
- On peut bien sûr toujours initialiser individuellement :

```
1 char *jour[7];
2 jour[0] = "lundi";
3 jour[1] = "mardi";...
4
```

Tableau de constantes chaînes

- Une constante chaîne est convertie par le compilateur en un pointeur sur son premier caractère. On peut donc facilement constituer un tableau de constantes chaînes.
- On peut bien sûr toujours initialiser individuellement :

```
1 char *jour[7];
2 jour[0] = "lundi";
3 jour[1] = "mardi";...
4
```

- Mais on peut aussi profiter de la syntaxe d'initialisation des tableaux en écrivant directement :

```
1 char *jour[7] = {"lundi","mardi",...}
2
```