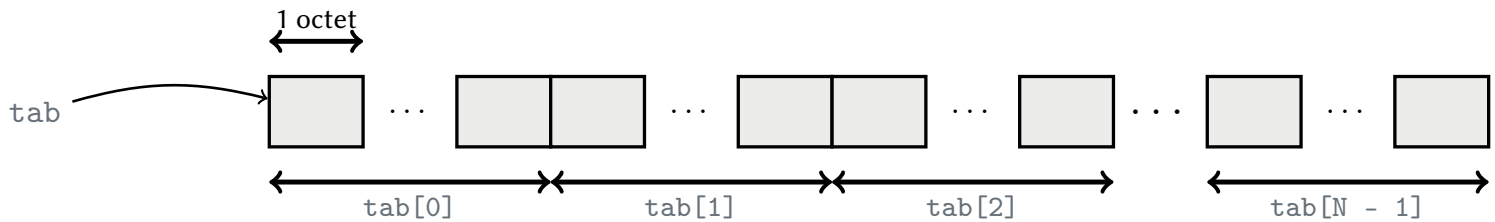
Pointeurs et tableaux statiques

Un tableau est un pointeur vers le 1er élément qui le constitue. Les autres éléments du tableau sont contigus en mémoire et situés à des adresses plus élevées.

On déclare un tableau statique de taille `N` de valeurs de type `T` par

```
T tab[N];
```



On accède à la valeur du i^e élément de `tab` par

```
tab[i]
```

On affecte une valeur `VAL` de type `T` en i^e position dans `tab` par

```
tab[i] = VAL;
```

Pointeurs et tableaux statiques

Sachant qu'un tableau `tab` est un pointeur et que ses éléments sont contigus en mémoire, la syntaxe

```
tab[i]
```

est équivalente à

```
*(tab + i)
```

Pointeurs et tableaux statiques

Sachant qu'un tableau `tab` est un pointeur et que ses éléments sont contigus en mémoire, la syntaxe

```
tab[i]
```

est équivalente à

```
*(tab + i)
```

Le type du pointeur `tab` permet de faire un décalage correct en fonction de la taille en mémoire de ses éléments.

Pointeurs et tableaux statiques

Sachant qu'un tableau `tab` est un pointeur et que ses éléments sont contigus en mémoire, la syntaxe

```
tab[i]
```

est équivalente à

```
*(tab + i)
```

Le type du pointeur `tab` permet de faire un décalage correct en fonction de la taille en mémoire de ses éléments.

En effet, si `ptr` est un pointeur pointant sur une zone mémoire de type `T`, la valeur de l'expression `ptr + i` dépend de la taille nécessaire pour représenter une valeur de type `T` (c.-à-d. de `sizeof(T)`).

Pointeurs et tableaux statiques

Considérons les instructions suivantes :

```
int tab[2];
char *ptr;

tab[0] = 300;
tab[1] = 60;

printf("%p %p\n",
    tab + 0, tab + 1);
printf("%d %d\n",
    tab[0], tab[1]);

ptr = (char *) tab;
printf("%p %p\n",
    ptr + 0, ptr + 1);
printf("%d %d\n",
    ptr[0], ptr[1]);
```

Pointeurs et tableaux statiques

Considérons les instructions suivantes :

```
int tab[2];
char *ptr;

tab[0] = 300;
tab[1] = 60;

printf("%p %p\n",
    tab + 0, tab + 1);
printf("%d %d\n",
    tab[0], tab[1]);

ptr = (char *) tab;
printf("%p %p\n",
    ptr + 0, ptr + 1);
printf("%d %d\n",
    ptr[0], ptr[1]);
```

Elles affichent

```
0x7fffc38b5d60 0x7fffc38b5d64
300 60
0x7fffc38b5d60 0x7fffc38b5d61
44 1
```

Pointeurs et tableaux statiques

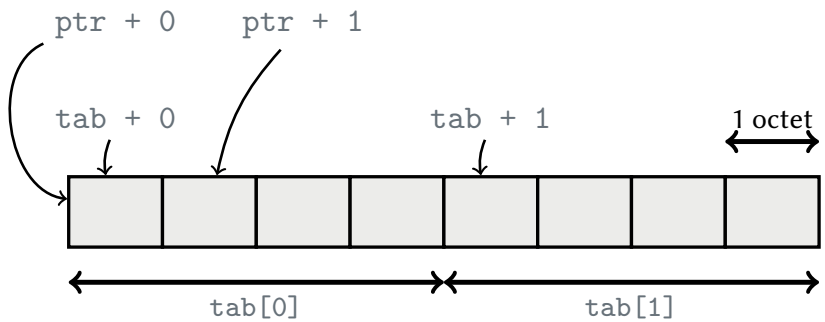
Considérons les instructions suivantes :

```
int tab[2];
char *ptr;

tab[0] = 300;
tab[1] = 60;

printf("%p %p\n",
    tab + 0, tab + 1);
printf("%d %d\n",
    tab[0], tab[1]);

ptr = (char *) tab;
printf("%p %p\n",
    ptr + 0, ptr + 1);
printf("%d %d\n",
    ptr[0], ptr[1]);
```

Elles affichent

```
0x7fffc38b5d60 0x7fffc38b5d64
300 60
0x7fffc38b5d60 0x7fffc38b5d61
44 1
```

Le pointeur `ptr` interprète les éléments du tableau `tab` comme des valeurs de taille 1 octet (= `sizeof(char)`).



Plan

Passage par valeur — rappel

Nous savons qu'il est impossible de modifier la valeur d'un argument passé à une fonction car celle-ci travaille sur une **copie de sa valeur** et non pas sur l'éventuelle variable figurant en argument.

Passage par valeur — rappel

Nous savons qu'il est impossible de modifier la valeur d'un argument passé à une fonction car celle-ci travaille sur une **copie de sa valeur** et non pas sur l'éventuelle variable figurant en argument.

```
void incrementer(int nb) {
    nb = nb + 1;
}
...
num = 5;
incrementer(num);
printf("%d\n", num);
```

Ces instructions affichent 5.

Passage par valeur — rappel

Nous savons qu'il est impossible de modifier la valeur d'un argument passé à une fonction car celle-ci travaille sur une **copie de sa valeur** et non pas sur l'éventuelle variable figurant en argument.

```
void incrementer(int nb) {
    nb = nb + 1;
}
...
num = 5;
incrementer(num);
printf("%d\n", num);
```

Ces instructions affichent 5.

En ligne 6, c'est la **valeur** de `num` qui est transmise et non pas la variable elle-même.

Passage par valeur — rappel

Nous savons qu'il est impossible de modifier la valeur d'un argument passé à une fonction car celle-ci travaille sur une **copie de sa valeur** et non pas sur l'éventuelle variable figurant en argument.

```
void incrementer(int nb) {
    nb = nb + 1;
}
...
num = 5;
incrementer(num);
printf("%d\n", num);
```

Ces instructions affichent 5.

En ligne 6, c'est la **valeur** de `num` qui est transmise et non pas la variable elle-même.

Ceci est encore plus clair quand il s'agit de l'appel

```
incrementer(2 * num + 1)
```

qui est considéré. La fonction travaille en effet avec la valeur **issue de l'évaluation** de l'expression `2 * num + 1` qui est recopiée dans la variable locale (paramètre) `nb`.

Passage par adresse

Cependant, si l'on donne l'adresse et le type d'une zone mémoire (ce qui revient à donner un pointeur vers la zone mémoire considérée), il devient possible de la modifier.

Passage par adresse

Cependant, si l'on donne l'adresse et le type d'une zone mémoire (ce qui revient à donner un pointeur vers la zone mémoire considérée), il devient possible de la modifier.

```
void incrementer(int *ptr_nb) {
    *ptr_nb = *ptr_nb + 1;
}
...
num = 5;
incrementer(&num);
printf("%d\n", num);
```

Ces instructions affichent 6.

Passage par adresse

Cependant, si l'on donne l'adresse et le type d'une zone mémoire (ce qui revient à donner un pointeur vers la zone mémoire considérée), il devient possible de la modifier.

```
void incrementer(int *ptr_nb) {
    *ptr_nb = *ptr_nb + 1;
}
...
num = 5;
incrementer(&num);
printf("%d\n", num);
```

Ces instructions affichent 6.

En ligne 6, c'est (la valeur de) l'**adresse** de `num` qui est transmise. Celle-ci universelle (visible partout).

Passage par adresse

Cependant, si l'on donne l'adresse et le type d'une zone mémoire (ce qui revient à donner un pointeur vers la zone mémoire considérée), il devient possible de la modifier.

```
void incrementer(int *ptr_nb) {
    *ptr_nb = *ptr_nb + 1;
}
...
num = 5;
incrementer(&num);
printf("%d\n", num);
```

Ces instructions affichent 6.

En ligne 6, c'est (la valeur de) l'**adresse** de `num` qui est transmise. Celle-ci universelle (visible partout).

C'est un passage par adresse.

Qualificateur `const`

Dans certains cas, un passage par adresse n'est pas fait pour modifier la valeur située à l'adresse spécifiée.

Qualificateur const

Dans certains cas, un passage par adresse n'est pas fait pour modifier la valeur située à l'adresse spécifiée.

P.ex., cette fonction affiche, sans la modifier, la chaîne de caractères `chaine` en substituant des étoiles `'*'` aux caractères `c`.

```
void eto(      char *chaine, char c) {
    int i = 0;
    while (chaine[i] != '\0') {
        if (chaine[i] == c)
            putchar('*');
        else
            putchar(chaine[i]);
        i += 1;
    }
}
```

Qualificateur `const`

Dans certains cas, un passage par adresse n'est pas fait pour modifier la valeur située à l'adresse spécifiée.

P.ex., cette fonction affiche, sans la modifier, la chaîne de caractères `chaine` en substituant des étoiles `'*'` aux caractères `c`.

On souhaite **protéger** `chaine` de toute modification sur son contenu.

```
void eto(      char *chaine, char c) {
    int i = 0;
    while (chaine[i] != '\0') {
        if (chaine[i] == c)
            putchar('*');
        else
            putchar(chaine[i]);
        i += 1;
    }
}
```

Qualificateur const

Dans certains cas, un passage par adresse n'est pas fait pour modifier la valeur située à l'adresse spécifiée.

P.ex., cette fonction affiche, sans la modifier, la chaîne de caractères `chaine` en substituant des étoiles `'*'` aux caractères `c`.

On souhaite **protéger** `chaine` de toute modification sur son contenu.

```
void eto(const char *chaine, char c) {
    int i = 0;
    while (chaine[i] != '\0') {
        if (chaine[i] == c)
            putchar('*');
        else
            putchar(chaine[i]);
        i += 1;
    }
}
```

Pour cela, on place le qualificateur `const` devant la déclaration de `chaine`.

Qualificateur `const`

Ainsi, plus généralement,

```
const T *ID
```

déclare un paramètre `ID`, pointeur sur une zone mémoire de type `T`, dont le contenu est protégé en écriture.

Qualificateur const

Ainsi, plus généralement,

```
const T *ID
```

déclare un paramètre `ID`, pointeur sur une zone mémoire de type `T`, dont le contenu est protégé en écriture.

```
void exemple_1(const int *x) {
    *x = 40;
}
```

Cette tentative directe pour modifier la valeur à l'adresse `x` est sanctionnée par le compilateur par une erreur.

Qualificateur const

Ainsi, plus généralement,

```
const T *ID
```

déclare un paramètre `ID`, pointeur sur une zone mémoire de type `T`, dont le contenu est protégé en écriture.

```
void exemple_1(const int *x) {
    *x = 40;
}
```

Cette tentative directe pour modifier la valeur à l'adresse `x` est sanctionnée par le compilateur par une erreur.

```
void exemple_2(const int *x) {
    int *tmp;
    tmp = x;
    *tmp = 40;
}
```

Cette tentative détournée pour modifier la valeur à l'adresse `x` est sanctionnée par le compilateur par un avertissement. Il est ainsi possible de modifier une valeur protégée.

Qualificateur const

Ainsi, plus généralement,

```
const T *ID
```

déclare un paramètre `ID`, pointeur sur une zone mémoire de type `T`, dont le contenu est protégé en écriture.

```
void exemple_1(const int *x) {
    *x = 40;
}
```

Cette tentative directe pour modifier la valeur à l'adresse `x` est sanctionnée par le compilateur par une erreur.

```
void exemple_2(const int *x) {
    int *tmp;
    tmp = x;
    *tmp = 40;
}
```

Cette tentative détournée pour modifier la valeur à l'adresse `x` est sanctionnée par le compilateur par un avertissement. Il est ainsi possible de modifier une valeur protégée.

Le qualificateur `const` est avant tout une **aide pour le développeur** : il informe d'un comportement à adopter.

Qualificateur `const`

Le qualificateur `const` permet quelques subtilités :

```
T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T` (il n'est pas possible de faire pointer `ID` ailleurs).

Qualificateur `const`

Le qualificateur `const` permet quelques subtilités :

```
T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T` (il n'est pas possible de faire pointer `ID` ailleurs). De plus,

```
const T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T`, dont le contenu est protégé en écriture.

Qualificateur const

Le qualificateur `const` permet quelques subtilités :

```
T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T` (il n'est pas possible de faire pointer `ID` ailleurs). De plus,

```
const T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T`, dont le contenu est protégé en écriture.

Voici un exemple qui illustre les quatre cas de figure :

```
void exemple_3(int *a, const int *b,
        int *const c,
        const int *const d) {

    int e;



}
```

Qualificateur const

Le qualificateur `const` permet quelques subtilités :

```
T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T` (il n'est pas possible de faire pointer `ID` ailleurs). De plus,

```
const T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T`, dont le contenu est protégé en écriture.

Voici un exemple qui illustre les quatre cas de figure :

```
void exemple_3(int *a, const int *b,
        int *const c,
        const int *const d) {

    int e;

    a = &e; /* Autorise */
    *a = 3; /* Autorise */
                                    }
```

Qualificateur const

Le qualificateur `const` permet quelques subtilités :

```
T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T` (il n'est pas possible de faire pointer `ID` ailleurs). De plus,

```
const T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T`, dont le contenu est protégé en écriture.

Voici un exemple qui illustre les quatre cas de figure :

```
void exemple_3(int *a, const int *b,
        int *const c,
        const int *const d) {

    int e;

    a = &e; /* Autorise */
    *a = 3; /* Autorise */
                                    b = &e; /* Autorise */
                                    *b = 3; /* Interdit */
                                }
```

Qualificateur const

Le qualificateur `const` permet quelques subtilités :

```
T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T` (il n'est pas possible de faire pointer `ID` ailleurs). De plus,

```
const T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T`, dont le contenu est protégé en écriture.

Voici un exemple qui illustre les quatre cas de figure :

```
void exemple_3(int *a, const int *b,
        int *const c,
        const int *const d) {

    int e;

    a = &e; /* Autorise */
    *a = 3; /* Autorise */

    b = &e; /* Autorise */
    *b = 3; /* Interdit */

    c = &e; /* Interdit */
    *c = 3; /* Autorise */
}
```

Qualificateur const

Le qualificateur `const` permet quelques subtilités :

```
T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T` (il n'est pas possible de faire pointer `ID` ailleurs). De plus,

```
const T *const ID
```

déclare un paramètre `ID`, pointeur fixe sur une zone mémoire de type `T`, dont le contenu est protégé en écriture.

Voici un exemple qui illustre les quatre cas de figure :

```
void exemple_3(int *a, const int *b,
        int *const c,
        const int *const d) {

    int e;

    a = &e; /* Autorise */
    *a = 3; /* Autorise */

    b = &e; /* Autorise */
    *b = 3; /* Interdit */

    c = &e; /* Interdit */
    *c = 3; /* Autorise */

    d = &e; /* Interdit */
    *d = 3; /* Interdit */
}
```

Plan

Données persistantes en mémoire

Nous savons que toutes les variables locales à une fonction n'existent plus après son appel.

Données persistantes en mémoire

Nous savons que toutes les variables locales à une fonction n'existent plus après son appel.

Pour créer des données persistantes dans une fonction, il faut écrire dans la mémoire ailleurs que dans la pile.

Données persistantes en mémoire

Nous savons que toutes les variables locales à une fonction n'existent plus après son appel.

Pour créer des données persistantes dans une fonction, il faut écrire dans la mémoire ailleurs que dans la pile.

On écrit pour cela dans le tas. Cela se fait en deux temps :

Données persistantes en mémoire

Nous savons que toutes les variables locales à une fonction n'existent plus après son appel.

Pour créer des données persistantes dans une fonction, il faut écrire dans la mémoire ailleurs que dans la pile.

On écrit pour cela dans le tas. Cela se fait en deux temps :

1. on demande au système d'allouer (de réserver) une certaine portion du tas ;

Données persistantes en mémoire

Nous savons que toutes les variables locales à une fonction n'existent plus après son appel.

Pour créer des données persistantes dans une fonction, il faut écrire dans la mémoire ailleurs que dans la pile.

On écrit pour cela dans le tas. Cela se fait en deux temps :

1. on demande au système d'allouer (de réserver) une certaine portion du tas ;
2. on écrit ensuite dans cette zone en lui affectant la valeur souhaitée.

Données persistantes en mémoire

Nous savons que toutes les variables locales à une fonction n'existent plus après son appel.

Pour créer des données persistantes dans une fonction, il faut écrire dans la mémoire ailleurs que dans la pile.

On écrit pour cela dans le tas. Cela se fait en deux temps :

1. on demande au système d'allouer (de réserver) une certaine portion du tas ;
2. on écrit ensuite dans cette zone en lui affectant la valeur souhaitée.

Pour allouer une portion du tas, on utilise la fonction

```
void *malloc(size_t size);
```

Données persistantes en mémoire

Nous savons que toutes les variables locales à une fonction n'existent plus après son appel.

Pour créer des données persistantes dans une fonction, il faut écrire dans la mémoire ailleurs que dans la pile.

On écrit pour cela dans le tas. Cela se fait en deux temps :

1. on demande au système d'allouer (de réserver) une certaine portion du tas ;
2. on écrit ensuite dans cette zone en lui affectant la valeur souhaitée.

Pour allouer une portion du tas, on utilise la fonction

```
void *malloc(size_t size);
```

Elle renvoie un pointeur de type générique `void *` sur une donnée en mémoire de taille `size` octets.

Utilisation de `malloc`

Pour allouer une zone de la mémoire pouvant accueillir `N` valeurs d'un type `T`, on utilise l'instruction

```
ptr = (T *) malloc(sizeof(T) * N);
```

où `ptr` est un pointeur pointant sur une zone de la mémoire de type `T`.

Utilisation de `malloc`

Pour allouer une zone de la mémoire pouvant accueillir `N` valeurs d'un type `T`, on utilise l'instruction

```
ptr = (T *) malloc(sizeof(T) * N);
```

où `ptr` est un pointeur pointant sur une zone de la mémoire de type `T`.

Explications :

- `(T *)` sert à préciser que la zone de la mémoire à allouer est de type `T`. Cette précision est nécessaire car, par défaut, `malloc` renvoie un pointeur sur une zone non typée (`void *`);

Utilisation de `malloc`

Pour allouer une zone de la mémoire pouvant accueillir `N` valeurs d'un type `T`, on utilise l'instruction

```
ptr = (T *) malloc(sizeof(T) * N);
```

où `ptr` est un pointeur pointant sur une zone de la mémoire de type `T`.

Explications :

- `(T *)` sert à préciser que la zone de la mémoire à allouer est de type `T`. Cette précision est nécessaire car, par défaut, `malloc` renvoie un pointeur sur une zone non typée (`void *`);
- l'argument `sizeof(T) * N` permet de demander à allouer une zone mémoire de taille `sizeof(T) * N` octets. Celle-ci pourra donc accueillir `N` valeurs de type `T`.

Utilisation de `malloc`

Pour allouer une zone de la mémoire pouvant accueillir `N` valeurs d'un type `T`, on utilise l'instruction

```
ptr = (T *) malloc(sizeof(T) * N);
```

où `ptr` est un pointeur pointant sur une zone de la mémoire de type `T`.

Explications :

- `(T *)` sert à préciser que la zone de la mémoire à allouer est de type `T`. Cette précision est nécessaire car, par défaut, `malloc` renvoie un pointeur sur une zone non typée (`void *`);
- l'argument `sizeof(T) * N` permet de demander à allouer une zone mémoire de taille `sizeof(T) * N` octets. Celle-ci pourra donc accueillir `N` valeurs de type `T`.

Après exécution de cette instruction, `ptr` pointe sur le début d'une zone de la mémoire de taille `sizeof(T) * N` octets.

Tests d'erreurs et `malloc`

C'est le système d'exploitation qui, lors de l'appel à `malloc` se charge de fournir une adresse pour la zone mémoire à allouer.

Tests d'erreurs et `malloc`

C'est le système d'exploitation qui, lors de l'appel à `malloc` se charge de fournir une adresse pour la zone mémoire à allouer.

Il se peut que pour une raison ou une autre, il ne soit pas possible d'allouer la zone mémoire demandée. Dans ce cas, `malloc` renvoie une valeur spéciale : `NULL`. Cette valeur vaut `0` et est une constante définie dans `stdlib.h`.

Tests d'erreurs et `malloc`

C'est le système d'exploitation qui, lors de l'appel à `malloc` se charge de fournir une adresse pour la zone mémoire à allouer.

Il se peut que pour une raison ou une autre, il ne soit pas possible d'allouer la zone mémoire demandée. Dans ce cas, `malloc` renvoie une valeur spéciale : `NULL`. Cette valeur vaut `0` et est une constante définie dans `stdlib.h`.

Il est impératif de tester, pour toute allocation dynamique réalisée, son bon déroulement. On procède de la manière suivante :

```
char *ptr;
ptr = (char *) malloc(sizeof(char) * 1);
if (NULL == ptr)
    ACTION
```

Tests d'erreurs et `malloc`

C'est le système d'exploitation qui, lors de l'appel à `malloc` se charge de fournir une adresse pour la zone mémoire à allouer.

Il se peut que pour une raison ou une autre, il ne soit pas possible d'allouer la zone mémoire demandée. Dans ce cas, `malloc` renvoie une valeur spéciale : `NULL`. Cette valeur vaut `0` et est une constante définie dans `stdlib.h`.

Il est impératif de tester, pour toute allocation dynamique réalisée, son bon déroulement. On procède de la manière suivante :

```
char *ptr;
ptr = (char *) malloc(sizeof(char) * 1);
if (NULL == ptr)
    ACTION
```

De cette manière, si l'allocation s'est mal passée, on prend en charge cette erreur par les instructions `ACTION`.

Tests d'erreurs et `malloc`

C'est le système d'exploitation qui, lors de l'appel à `malloc` se charge de fournir une adresse pour la zone mémoire à allouer.

Il se peut que pour une raison ou une autre, il ne soit pas possible d'allouer la zone mémoire demandée. Dans ce cas, `malloc` renvoie une valeur spéciale : `NULL`. Cette valeur vaut `0` et est une constante définie dans `stdlib.h`.

Il est impératif de tester, pour toute allocation dynamique réalisée, son bon déroulement. On procède de la manière suivante :

```
char *ptr;
ptr = (char *) malloc(sizeof(char) * 1);
if (NULL == ptr)
    ACTION
```

De cette manière, si l'allocation s'est mal passée, on prend en charge cette erreur par les instructions `ACTION`.

`ACTION` peut être un `exit(EXIT_FAILURE);` si l'on se trouve dans le `main` ou un `return NULL;` si l'on se trouve dans une fonction.

Libération de mémoire

Pour désallouer une zone de la mémoire, on utilise la fonction

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

Libération de mémoire

Pour désallouer une zone de la mémoire, on utilise la fonction

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

On l'utilise de la manière suivante :

```
free(ptr);
ptr = NULL;
```

où `ptr` est un pointeur. La 2^{e} ligne sert à ne plus conserver l'accès sur la zone libéré (non indispensable mais peut éviter des erreurs).

Libération de mémoire

Pour désallouer une zone de la mémoire, on utilise la fonction

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

On l'utilise de la manière suivante :

```
free(ptr);
ptr = NULL;
```

où `ptr` est un pointeur. La 2e ligne sert à ne plus conserver l'accès sur la zone libéré (non indispensable mais peut éviter des erreurs).

```
short *ptr;
ptr = (short *)
    malloc(sizeof(short) * 15);
free(ptr);
ptr = NULL;
```

La l. 2 alloue une zone de la mémoire pouvant accueillir `15` valeurs de type `short`. En l. 3, cette zone est libérée. Il est d'ores impossible d'y accéder.

Libération de mémoire

Pour désallouer une zone de la mémoire, on utilise la fonction

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

On l'utilise de la manière suivante :

```
free(ptr);
ptr = NULL;
```

où `ptr` est un pointeur. La 2e ligne sert à ne plus conserver l'accès sur la zone libéré (non indispensable mais peut éviter des erreurs).

```
short *ptr;
ptr = (short *)
    malloc(sizeof(short) * 15);
free(ptr);
ptr = NULL;
```

La l. 2 alloue une zone de la mémoire pouvant accueillir `15` valeurs de type `short`. En l. 3, cette zone est libérée. Il est d'ores impossible d'y accéder.

Important : pour éviter les fuites mémoire, il faut désallouer toute zone de la mémoire qui n'est plus utilisée.

La fonction `calloc`

La fonction

```
void *calloc(size_t nmemb, size_t size);
```

alloue et renvoie un pointeur sur une zone de la mémoire de `nmemb * size` octets, tous initialisés avec des `0`.

La fonction `calloc`

La fonction

```
void *calloc(size_t nmemb, size_t size);
```

alloue et renvoie un pointeur sur une zone de la mémoire de `nmemb * size` octets, tous initialisés avec des `0`.

P.ex.,

```
long *ptr;
ptr = (long *) calloc(13, sizeof(long));
```

alloue une zone de la mémoire pouvant accueillir `13` valeurs de type `long`, initialisées à `0`.

Plan

Tableaux dynamiques

Un tableau dynamique de `N` valeurs de type `T` est un pointeur pointant vers une zone de la mémoire de `sizeof(T) * N` octets.

Tableaux dynamiques

Un tableau dynamique de `N` valeurs de type `T` est un pointeur pointant vers une zone de la mémoire de `sizeof(T) * N` octets.

P.ex.,

```
int *tab;
tab = (int *) malloc(sizeof(int) * 66);
```

déclare un tableau dynamique de valeurs de type `int` de taille `66`.

Tableaux dynamiques

Un tableau dynamique de `N` valeurs de type `T` est un pointeur pointant vers une zone de la mémoire de `sizeof(T) * N` octets.

P.ex.,

```
int *tab;
tab = (int *) malloc(sizeof(int) * 66);
```

déclare un tableau dynamique de valeurs de type `int` de taille `66`.

On lit et écrit dans un tableau dynamique de la même manière que dans un tableau statique. En effet, `tab` pointe vers la première case du tableau, et pour tout $0 \leqslant$ `i` $<$ `66`, `(tab + i)` pointe vers la case d'indice `i`.

Tableaux dynamiques

Un tableau dynamique de `N` valeurs de type `T` est un pointeur pointant vers une zone de la mémoire de `sizeof(T) * N` octets.

P.ex.,

```
int *tab;
tab = (int *) malloc(sizeof(int) * 66);
```

déclare un tableau dynamique de valeurs de type `int` de taille `66`.

On lit et écrit dans un tableau dynamique de la même manière que dans un tableau statique. En effet, `tab` pointe vers la première case du tableau, et pour tout $0 \leqslant$ `i` $<$ `66`, `(tab + i)` pointe vers la case d'indice `i`.

La fonction `free` vue précédemment permet de désallouer un tableau. On utilise donc

```
free(tab);
```

pour libérer la zone mémoire occupée par `tab` (c.-à-d. les `66` entiers situés à partir de l'adresse `tab`).

Création d'un tableau dynamique à deux dimensions

Un tableau à deux dimensions est un tableau dont chaque case est elle-même un tableau.

Création d'un tableau dynamique à deux dimensions

Un tableau à deux dimensions est un tableau dont chaque case est elle-même un tableau. Un tableau dynamique à deux dimensions est donc un **pointeur sur un pointeur**.

Création d'un tableau dynamique à deux dimensions

Un tableau à deux dimensions est un tableau dont chaque case est elle-même un tableau. Un tableau dynamique à deux dimensions est donc un **pointeur sur un pointeur**. Ceci se généralise dans le cas des tableaux à plus de deux dimensions.

Création d'un tableau dynamique à deux dimensions

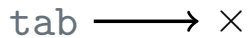
Un tableau à deux dimensions est un tableau dont chaque case est elle-même un tableau. Un tableau dynamique à deux dimensions est donc un **pointeur sur un pointeur**. Ceci se généralise dans le cas des tableaux à plus de deux dimensions.

Les instructions

```
int i, j, **tab;
tab = (int **) malloc(sizeof(int *) * 4);
if (tab == NULL) exit(EXIT_FAILURE);
for (i = 0 ; i < 4 ; ++i) {
    tab[i] = (int *) malloc(sizeof(int) * 3);
    if (tab[i] == NULL) exit(EXIT_FAILURE);
    for (j = 0 ; j < 3 ; ++j)
        tab[i][j] = 0;
}
```

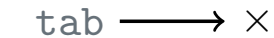
permettent de créer un tableau dynamique à deux dimensions de taille 4×3 de valeurs de type `int` initialisées à `0`.

Création d'un tableau dynamique à deux dimensions

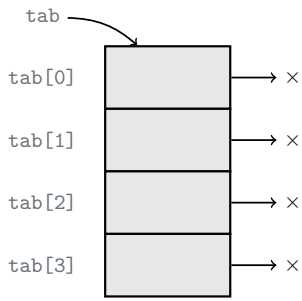


Initialement, `tab` est pointeur vers une zone indéterminée de la mémoire.

Création d'un tableau dynamique à deux dimensions

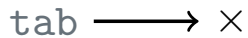


Initialement, `tab` est pointeur vers une zone indéterminée de la mémoire.

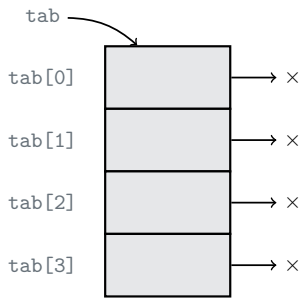


Juste après la 1re allocation dynamique, `tab` est dans cet état. L'espace de mémoire de `4` pointeurs sur des entiers a été réservé.

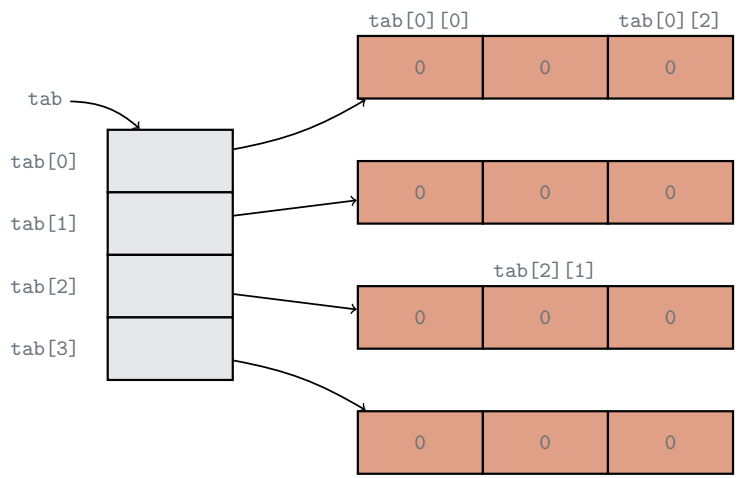
Création d'un tableau dynamique à deux dimensions



Initialement, `tab` est pointeur vers une zone indéterminée de la mémoire.



Juste après la 1re allocation dynamique, `tab` est dans cet état. L'espace de mémoire de `4` pointeurs sur des entiers a été réservé.



Après les allocations dynamiques des tableaux à une dimension de taille `3` et des initialisations des cases d'entiers, `tab` est dans cet état.

Destruction d'un tableau dynamique à deux dimensions

Pour libérer l'espace occupé par un tableau à deux dimensions, on utilise plusieurs fois la fonction `free`.

Destruction d'un tableau dynamique à deux dimensions

Pour libérer l'espace occupé par un tableau à deux dimensions, on utilise plusieurs fois la fonction `free`.

Les instructions

```
for (i = 0 ; i < 4 ; ++i) {
    free(tab[i]);
    tab[i] = NULL;
}
free(tab);
tab = NULL;
```

permettent de libérer l'espace mémoire occupé par un tableau `tab` à deux dimensions de taille $4 \times N$ de valeurs de type `T` où `N` est un entier strictement positif quelconque et `T` est un type.

Destruction d'un tableau dynamique à deux dimensions

Pour libérer l'espace occupé par un tableau à deux dimensions, on utilise plusieurs fois la fonction `free`.

Les instructions

```
for (i = 0 ; i < 4 ; ++i) {
    free(tab[i]);
    tab[i] = NULL;
}
free(tab);
tab = NULL;
```

permettent de libérer l'espace mémoire occupé par un tableau `tab` à deux dimensions de taille $4 \times N$ de valeurs de type `T` où `N` est un entier strictement positif quelconque et `T` est un type.

Remarque : la connaissance de la seconde dimension du tableau (`N`) n'est pas utile et n'intervient pas dans la suite d'instructions ci-dessus.

La fonction `realloc`

Il est possible de modifier la taille d'un tableau dynamique via la fonction

```
void *realloc(void *ptr, size_t size);
```

La fonction `realloc`

Il est possible de modifier la taille d'un tableau dynamique via la fonction

```
void *realloc(void *ptr, size_t size);
```

Celle-ci renvoie un pointeur sur une zone de la mémoire de taille `size` octets. Le contenu de cette zone mémoire est le même que celui de la zone pointée par `ptr`.

La fonction `realloc`

Il est possible de modifier la taille d'un tableau dynamique via la fonction

```
void *realloc(void *ptr, size_t size);
```

Celle-ci renvoie un pointeur sur une zone de la mémoire de taille `size` octets. Le contenu de cette zone mémoire est le même que celui de la zone pointée par `ptr`.

Attention : l'adresse de la zone de la mémoire réallouée peut être différente de son adresse d'origine.

La fonction `realloc`

Il est possible de modifier la taille d'un tableau dynamique via la fonction

```
void *realloc(void *ptr, size_t size);
```

Celle-ci renvoie un pointeur sur une zone de la mémoire de taille `size` octets. Le contenu de cette zone mémoire est le même que celui de la zone pointée par `ptr`.

Attention : l'adresse de la zone de la mémoire réallouée peut être différente de son adresse d'origine. En effet,

```
int *tab;
tab = (int *) malloc(sizeof(int) * 150);
printf("%p\n", tab);
tab = (int *) realloc(tab, sizeof(int) * 250000);
printf("%p\n", tab);
```

affiche
0x1205010
0x7fb123f9d010.

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {









}
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {


    }



}
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;

    }



}
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }



}
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;


}
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;
    t_reelle += 1;

}
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;
    t_reelle += 1;
    scanf(" %c", &car);
}
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;
    t_reelle += 1;
    scanf(" %c", &car);
}
chaine[t_reelle] = '\0';
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;
    t_reelle += 1;
    scanf(" %c", &car);
}
chaine[t_reelle] = '\0';
chaine = (char *) realloc(chaine, t_reelle + 1); /* Diminution. */
```

Exemple d'utilisation de `realloc`

Construction d'une chaîne de caractères, caractère par caractère, dans un tableau dynamique. La lecture s'arrête sur lecture de `'!'`.

```
char car, *chaine;
int t_max, t_reelle;
t_reelle = 0;
t_max = 2;
chaine = (char *) malloc(sizeof(char) * t_max); /* Allocation initiale. */
scanf(" %c", &car);
while (car != '!') {
    if (t_reelle + 1 >= t_max) {
        t_max *= 2;
        chaine = (char *) realloc(chaine, t_max); /* Augmentation. */
    }
    chaine[t_reelle] = car;
    t_reelle += 1;
    scanf(" %c", &car);
}
chaine[t_reelle] = '\0';
chaine = (char *) realloc(chaine, t_reelle + 1); /* Diminution. */
printf("%s\n", chaine);
```