

Plan

Pointeurs de fonction

Principe

En paramètre et en retour

Généricité

Implantation de monoïdes

Plan

Pointeurs de fonction

Principe

En paramètre et en retour

Généricité

Implantation de monoïdes

Pointeurs de fonction

En C, on peut manipuler divers types d'objets : des variables d'un type scalaire, des tableaux, des variables d'un type structuré, des adresses, *etc.*

Pointeurs de fonction

En C, on peut manipuler divers types d'objets : des variables d'un type scalaire, des tableaux, des variables d'un type structuré, des adresses, *etc.*

À l'inverse, les **fonctions** n'entrent pas dans cette catégorie d'objets directement manipulables.

Pointeurs de fonction

En C, on peut manipuler divers types d'objets : des variables d'un type scalaire, des tableaux, des variables d'un type structuré, des adresses, *etc.*

À l'inverse, les **fonctions** n'entrent pas dans cette catégorie d'objets directement manipulables.

Cependant, au même titre qu'une variable, toute fonction possède une **adresse** en mémoire. Il devient alors possible de réaliser des opérations sur les fonctions au moyen de leur adresse.

Pointeurs de fonction

En C, on peut manipuler divers types d'objets : des variables d'un type scalaire, des tableaux, des variables d'un type structuré, des adresses, *etc.*

À l'inverse, les **fonctions** n'entrent pas dans cette catégorie d'objets directement manipulables.

Cependant, au même titre qu'une variable, toute fonction possède une **adresse** en mémoire. Il devient alors possible de réaliser des opérations sur les fonctions au moyen de leur adresse.

On parle alors de **pointeur de fonction**.

Adresse d'une fonction

Si `fct` est une fonction, la syntaxe

`&fct`

permet d'accéder à l'adresse de `fct`.

Adresse d'une fonction

Si `fct` est une fonction, la syntaxe

`&fct`

permet d'accéder à l'adresse de `fct`.

```
#include <stdio.h>
int somme(int a, int b) {
    return a + b;
}
int produit(int a, int b) {
    return a * b;
}
int main() {
    printf("%p\n", &somme);
    printf("%p\n", &produit);
    return 0;
}
```

Ce programme affiche `0x40052d` et `0x400541`, respectivement les adresses des fonctions `somme` et `produit`.

Adresse d'une fonction

Si `fct` est une fonction, la syntaxe

`&fct`

permet d'accéder à l'adresse de `fct`.

```
#include <stdio.h>
int somme(int a, int b) {
    return a + b;
}
int produit(int a, int b) {
    return a * b;
}
int main() {
    printf("%p\n", &somme);
    printf("%p\n", &produit);
    return 0;
}
```

Ce programme affiche `0x40052d` et `0x400541`, respectivement les adresses des fonctions `somme` et `produit`.

Note : aux lignes 9 et 10, il est possible de ne pas mentionner les `&`. Le compilateur comprend implicitement qu'il s'agit de pointeurs de fonction.

Le type pointeur de fonction

La syntaxe

$$T (*FCT)(T_1, \dots, T_N);$$

où

- ▶ T, T_1, \dots, T_N sont des types;
- ▶ FCT est un identificateur;

permet de **déclarer** un pointeur de fonction.

Le type pointeur de fonction

La syntaxe

$$T (*FCT)(T_1, \dots, T_N);$$

où

- ▶ T, T_1, \dots, T_N sont des types;
- ▶ FCT est un identificateur;

permet de **déclarer** un pointeur de fonction.

Celui-ci a FCT pour identificateur et peut être l'adresse d'une fonction de type de retour T et de signature (T_1, \dots, T_N) .

Le type pointeur de fonction

La syntaxe

$$T (*FCT)(T_1, \dots, T_N);$$

où

- ▶ T, T_1, \dots, T_N sont des types;
- ▶ FCT est un identificateur;

permet de **déclarer** un pointeur de fonction.

Celui-ci a FCT pour identificateur et peut être l'adresse d'une fonction de type de retour T et de signature (T_1, \dots, T_N) .

```
float moyenne(int a, int b) {
    return (0.0 + a + b) / 2;
}
...
/* Decl. d'un ptr de fonction */
float (*moy)(int, int);

/* Utilisation */
moy = &moyenne;
printf("%f\n", moy(2, 3));
```

Pour la même raison que dans l'exemple précédent, il est possible à la ligne 9 de ne pas mentionner le `&`.

Cependant, pour la clarté du code, nous prenons la **convention de mentionner tous les `&`**.

Champs pointeurs de fonction

Un champ d'un type structuré peut être un pointeur sur une fonction.

Champs pointeurs de fonction

Un **champ d'un type structuré** peut être un pointeur sur une fonction.

Ceci déclare un type structuré sensé modéliser des suites d'entiers :

```
typedef struct {  
    int t;  
    int (*t_suiv)(int);  
} Suite;
```

`t` contient le terme courant de la suite et `t_suiv` est la fonction qui, étant donné un terme en entrée, calcule le terme suivant.

Champs pointeurs de fonction

Un **champ d'un type structuré** peut être un pointeur sur une fonction.

Ceci déclare un type structuré sensé modéliser des suites d'entiers :

```
typedef struct {
    int t;
    int (*t_suiv)(int);
} Suite;
```

`t` contient le terme courant de la suite et `t_suiv` est la fonction qui, étant donné un terme en entrée, calcule le terme suivant.

```
int suivant(Suite *s) {
    s->t = s->t_suiv(s->t);
    return s->t;
}

int mul_2(int x) {
    return 2 * x;
}

...
int i;
Suite s;
s.t = 1;
s.t_suiv = &mul_2;
for (i = 0 ; i < 6 ; ++i) {
    printf("%d ", suivant(&s));
}
```

Ceci affiche **1 2 4 8 16 32**.

Tableaux de pointeurs de fonction

Il est possible de manipuler des tableaux de pointeurs de fonction.

Tableaux de pointeurs de fonction

Il est possible de manipuler des **tableaux de pointeurs de fonction**.

Pour cela, on procède en deux étapes :

1. on déclare un type alias pour le pointeur de fonction. Syntaxe :

```
typedef T (*FCT)(T1, ..., TN);
```

Tableaux de pointeurs de fonction

Il est possible de manipuler des **tableaux de pointeurs de fonction**.

Pour cela, on procède en deux étapes :

1. on déclare un type alias pour le pointeur de fonction. Syntaxe :

```
typedef T (*FCT) (T1, ..., TN);
```

2. on déclare ensuite le tableau de manière usuelle. Syntaxe :

```
FCT tab[M];
```

Tableaux de pointeurs de fonction

Il est possible de manipuler des **tableaux de pointeurs de fonction**.

Pour cela, on procède en deux étapes :

1. on déclare un type alias pour le pointeur de fonction. Syntaxe :

```
typedef T (*FCT)(T1, ..., TN);
```

2. on déclare ensuite le tableau de manière usuelle. Syntaxe :

```
FCT tab[M];
```

La syntaxe plus directe

```
T (*tab[M])(T1, ..., TN);
```

existe mais rend le code plus difficile à lire. Elle déclare un tableau `tab` de pointeurs de fonction sans la déclaration de type préalable de la 1^{re} méthode.

Tableaux de pointeurs de fonction

```
/* Alias pour ptr. de fct. */
typedef int (*opb)(int, int);

int add(int a, int b) {
    return a + b;
}
int mul(int a, int b) {
    return a * b;
}

...
opb tab[2];

tab[0] = &add;
tab[1] = &mul;
printf("%d %d\n",
        tab[0](10, 20),
        tab[1](10, 20));
```

Ceci affiche 30 200.

Tableaux de pointeurs de fonction

```
/* Alias pour ptr. de fct. */
typedef int (*opb)(int, int);

int add(int a, int b) {
    return a + b;
}
int mul(int a, int b) {
    return a * b;
}

...
opb tab[2];

tab[0] = &add;
tab[1] = &mul;
printf("%d %d\n",
        tab[0](10, 20),
        tab[1](10, 20));
```

Ceci affiche **30 200**.

Les tableaux de pointeurs de fonction peuvent être **dynamiques**. La ligne 10 peut être remplacée par

```
opb *tab;
tab = (opb *) malloc(sizeof(opb) * 2);
```

Plan

Pointeurs de fonction

Principe

En paramètre et en retour

Généricité

Implantation de monoïdes

Pointeur de fonction en paramètre

Une fonction peut être paramétrée par un pointeur de fonction. Un paramètre pointeur de fonction est spécifié avec la même syntaxe que celle qui sert à le déclarer.

Pointeur de fonction en paramètre

Une fonction peut être paramétrée par un pointeur de fonction. Un paramètre pointeur de fonction est spécifié avec la même syntaxe que celle qui sert à le déclarer.

P.ex.,

```
int appliquer(int n, int k, int (*f)(int)) {  
    int i;  
    for (i = 0 ; i < k ; ++i)  
        n = f(n);  
    return n;  
}
```

est une fonction est paramétrée par un pointeur de fonction acceptant un entier et renvoyant un entier.

Pointeur de fonction en paramètre

```
int add_1(int n) {  
    return n + 1;  
}
```

```
int mul_2(int n) {  
    return 2 * n;  
}
```

```
...  
printf("%d\n",  
       appliquer(3, 4, &add_1));  
  
printf("%d\n",  
       appliquer(3, 4, &mul_2));
```

Pointeur de fonction en paramètre

```
int add_1(int n) {  
    return n + 1;  
}  
  
int mul_2(int n) {  
    return 2 * n;  
}
```

```
...  
printf("%d\n",  
        appliquer(3, 4, &add_1));  
  
printf("%d\n",  
        appliquer(3, 4, &mul_2));
```

Le 1^{er} appel à `appliquer` calcule

$$(((3 + 1) + 1) + 1) + 1$$

et affiche donc **7**.

Pointeur de fonction en paramètre

```
int add_1(int n) {  
    return n + 1;  
}  
  
int mul_2(int n) {  
    return 2 * n;  
}
```

```
...  
printf("%d\n",  
        appliquer(3, 4, &add_1));  
  
printf("%d\n",  
        appliquer(3, 4, &mul_2));
```

Le 1^{er} appel à `appliquer` calcule

$$(((3 + 1) + 1) + 1) + 1$$

et affiche donc **7**.

Le 2^e appel à `appliquer` calcule

$$(((3 * 2) * 2) * 2) * 2$$

et affiche donc **48**.

Renvoi d'un pointeur de fonction

Il est possible de définir des fonctions dont le **type de retour** est un **pointeur de fonction**.

Renvoi d'un pointeur de fonction

Il est possible de définir des fonctions dont le **type de retour** est un **pointeur de fonction**.

Pour cela, on procède en deux étapes :

1. on déclare un type alias **R** pour le pointeur de fonction que l'on souhaite renvoyer;

Renvoi d'un pointeur de fonction

Il est possible de définir des fonctions dont le **type de retour** est un **pointeur de fonction**.

Pour cela, on procède en deux étapes :

1. on déclare un type alias **R** pour le pointeur de fonction que l'on souhaite renvoyer ;
2. on définit la fonction souhaitée, dont le type de retour est **R**.

Renvoi d'un pointeur de fonction

Il est possible de définir des fonctions dont le **type de retour** est un **pointeur de fonction**.

Pour cela, on procède en deux étapes :

1. on déclare un type alias **R** pour le pointeur de fonction que l'on souhaite renvoyer;
2. on définit la fonction souhaitée, dont le type de retour est **R**.

La syntaxe plus directe

```
R (*FCT(T1 ARG1, ..., TN ARGN))(R1, ..., RM) {  
    ...  
}
```

permet de définir directement une fonction **FCT** de signature $(T1, \dots, TN)$ renvoyant l'adresse d'une fonction de type de retour **R** et de signature $(R1, \dots, RM)$. Cependant, le code devient illisible.

Renvoi d'un pointeur de fonction

Exemple : opération aléatoire sur des entiers.

```
/* Definition des operations */      /* Type de retour */
int add(int a, int b) {              typedef int (*opb)(int, int);
    return a + b;
}
int sub(int a, int b) {
    return a - b;
}
int mod(int a, int b) {
    return a % b;
}

opb op_alea() {
    opb tab[3];
    tab[0] = &add;
    tab[1] = &sub;
    tab[2] = &mod;
    return tab[rand() % 3];
}
```

Renvoi d'un pointeur de fonction

Exemple : opération aléatoire sur des entiers.

```
/* Definition des operations */      /* Type de retour */
int add(int a, int b) {              typedef int (*opb)(int, int);
    return a + b;
}
int sub(int a, int b) {              opb op_alea() {
    return a - b;                    opb tab[3];
}                                     tab[0] = &add;
int mod(int a, int b) {              tab[1] = &sub;
    return a % b;                    tab[2] = &mod;
}                                     return tab[rand() % 3];
}                                     }
```

On peut utiliser `op_alea` de la manière suivante :

```
int n;
n = op_alea()(3, 4);
```

Ceci affecte, de manière aléatoire, 7, -1 ou 3 à n.

Plan

Pointeurs de fonction

Principe

En paramètre et en retour

Généricité

Implantation de monoïdes

Principe de généricité

Une **fonction** est dite **générique** si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Principe de généricité

Une **fonction** est dite **générique** si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Exemples :

- ▶ une fonction qui teste si deux valeurs sont égales;

Principe de généricité

Une **fonction** est dite **générique** si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Exemples :

- ▶ une fonction qui teste si deux valeurs sont égales;
- ▶ une fonction qui affiche plusieurs fois une même valeur.

Principe de généricité

Une **fonction** est dite **générique** si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Exemples :

- ▶ une fonction qui teste si deux valeurs sont égales;
- ▶ une fonction qui affiche plusieurs fois une même valeur.

Une **structure de donnée** est dite **générique** si elle peut représenter des données dont le type n'est pas fixé.

Principe de généricité

Une **fonction** est dite **générique** si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Exemples :

- ▶ une fonction qui teste si deux valeurs sont égales;
- ▶ une fonction qui affiche plusieurs fois une même valeur.

Une **structure de donnée** est dite **générique** si elle peut représenter des données dont le type n'est pas fixé.

Exemples :

- ▶ une liste dont les éléments sont d'un type non fixé;

Principe de généricité

Une **fonction** est dite **générique** si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Exemples :

- ▶ une fonction qui teste si deux valeurs sont égales;
- ▶ une fonction qui affiche plusieurs fois une même valeur.

Une **structure de donnée** est dite **générique** si elle peut représenter des données dont le type n'est pas fixé.

Exemples :

- ▶ une liste dont les éléments sont d'un type non fixé;
- ▶ un arbre binaire dont les éléments sont d'un type non fixé;

Principe de généricité

Une **fonction** est dite **générique** si elle peut accepter des arguments qui ne sont pas seulement ceux d'un type bien précis.

Exemples :

- ▶ une fonction qui teste si deux valeurs sont égales;
- ▶ une fonction qui affiche plusieurs fois une même valeur.

Une **structure de donnée** est dite **générique** si elle peut représenter des données dont le type n'est pas fixé.

Exemples :

- ▶ une liste dont les éléments sont d'un type non fixé;
- ▶ un arbre binaire dont les éléments sont d'un type non fixé;
- ▶ un tableau dont les éléments sont d'un type non fixé.

Le type `void *`

Pour manipuler une donnée dont le type n'est pas spécifié à l'avance, on utilise son **adresse**.

Le type `void *`

Pour manipuler une donnée dont le type n'est pas spécifié à l'avance, on utilise son **adresse**.

Il s'agit donc d'une adresse dont on ne connaît pas le type : c'est une adresse de type `void *`.

Le type `void *`

Pour manipuler une donnée dont le type n'est pas spécifié à l'avance, on utilise son **adresse**.

Il s'agit donc d'une adresse dont on ne connaît pas le type : c'est une adresse de type `void *`.

Le type `void *` est appelé **type pointeur générique**.

Le type `void *`

Pour manipuler une donnée dont le type n'est pas spécifié à l'avance, on utilise son **adresse**.

Il s'agit donc d'une adresse dont on ne connaît pas le type : c'est une adresse de type `void *`.

Le type `void *` est appelé **type pointeur générique**.

Pour convertir un pointeur générique `ptr_g` vers un pointeur d'un type connu `T`, on utilise l'**opérateur de coercition**

```
(T *) ptr_g.
```

Le type `void *`

Pour manipuler une donnée dont le type n'est pas spécifié à l'avance, on utilise son **adresse**.

Il s'agit donc d'une adresse dont on ne connaît pas le type : c'est une adresse de type `void *`.

Le type `void *` est appelé **type pointeur générique**.

Pour convertir un pointeur générique `ptr_g` vers un pointeur d'un type connu `T`, on utilise l'**opérateur de coercition**

$$(T *) ptr_g.$$

Avant de pouvoir interpréter (c.-à-d. déréférencer) la valeur située à une adresse spécifiée par un pointeur générique, **le convertir est primordial**.

Fonction générique de comparaison

```
int ega(int nbo, void *x, void *y) {  
    char *xc, *yc;  
    int i;  
  
    xc = (char *) x;  
    yc = (char *) y;  
  
}
```

La fonction `ega` est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique de comparaison

```
int ega(int nbo, void *x, void *y) {  
    char *xc, *yc;  
    int i;  
  
    xc = (char *) x;  
    yc = (char *) y;  
    for (i = 0 ; i < nbo ; ++i)  
  
}
```

La fonction `ega` est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique de comparaison

```
int ega(int nbo, void *x, void *y) {  
    char *xc, *yc;  
    int i;  
  
    xc = (char *) x;  
    yc = (char *) y;  
    for (i = 0 ; i < nbo ; ++i)  
        if (xc[i] != yc[i])  
            return 0;  
  
}
```

La fonction `ega` est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique de comparaison

```
int ega(int nbo, void *x, void *y) {
    char *xc, *yc;
    int i;

    xc = (char *) x;
    yc = (char *) y;
    for (i = 0 ; i < nbo ; ++i)
        if (xc[i] != yc[i])
            return 0;
    return 1;
}
```

La fonction `ega` est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

Fonction générique de comparaison

```
int ega(int nbo, void *x, void *y) {
    char *xc, *yc;
    int i;

    xc = (char *) x;
    yc = (char *) y;
    for (i = 0 ; i < nbo ; ++i)
        if (xc[i] != yc[i])
            return 0;
    return 1;
}
```

La fonction `ega` est générique : elle permet de tester l'égalité entre deux variables dont le **type n'est pas connu lors de l'écriture de la fonction**.

On l'utilise de la manière suivante :

```
ega(sizeof(T), &t1, &t2)
```

pour comparer deux variables `t1` et `t2` de type `T`.

Fonction générique d'affichage de tableau

```
void aff_tab(void **tab, int n, void (*aff_elt)(void *)) {  
    int i;  
  
    for (i = 0 ; i < n ; ++i) {  
  
    }  
}
```

La fonction `aff_tab` est générique : elle permet d'afficher les éléments d'un tableau dont le **type n'est pas connu lors de l'écriture de la fonction.**

Fonction générique d'affichage de tableau

```
void aff_tab(void **tab, int n, void (*aff_elt)(void *)) {
    int i;

    for (i = 0 ; i < n ; ++i) {
        aff_elt(tab[i]);
    }
}
```

La fonction `aff_tab` est générique : elle permet d'afficher les éléments d'un tableau dont le **type n'est pas connu lors de l'écriture de la fonction.**

Fonction générique d'affichage de tableau

```
void aff_tab(void **tab, int n, void (*aff_elt)(void *)) {
    int i;

    for (i = 0 ; i < n ; ++i) {
        aff_elt(tab[i]);
        printf(" ");
    }
}
```

La fonction `aff_tab` est générique : elle permet d'afficher les éléments d'un tableau dont le **type n'est pas connu lors de l'écriture de la fonction.**

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille `13` de pointeurs sur des **entiers** :

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille `13` de pointeurs sur des **entiers** :

```
void aff_int(void *x) {  
    int e;  
  
}
```

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille `13` de pointeurs sur des **entiers** :

```
void aff_int(void *x) {  
    int e;  
    e = *((int *) x);  
  
}
```

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille 13 de pointeurs sur des **entiers** :

```
void aff_int(void *x) {  
    int e;  
    e = *((int *) x);  
    printf("%d", e);  
}
```

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille 13 de pointeurs sur des **entiers** :

```
void aff_int(void *x) {
    int e;
    e = *((int *) x);
    printf("%d", e);
}

/* Version raccourcie. */
void aff_int(void *x) {
    printf("%d", *((int *) x));
}
```

Fonction générique d'affichage de tableau

On l'utilise la fonction `aff_tab` de la manière suivante.

Pour afficher un tableau `tab` de taille `13` de pointeurs sur des **entiers** :

```
void aff_int(void *x) {
    int e;
    e = *((int *) x);
    printf("%d", e);
}

/* Version raccourcie. */
void aff_int(void *x) {
    printf("%d", *((int *) x));
}

...
aff_tab((void **) tab, 13, &aff_int);
```

Fonction générique d'affichage de tableau

Pour afficher un tableau `tab` de taille 23 de pointeurs sur des variables de **type structuré** `Date` :

```
typedef struct {
    int jour;
    int mois;
    int annee;
} Date;
...
void aff_date(void *d) {
    Date dd;
    dd = *((Date *) d);
    printf("%d-%d-%d", dd.jour, dd.mois, dd.annee);
}
...
aff_tab((void **) tab, 23, &aff_date);
```

Listes génériques

On souhaite définir une structure de donnée liste dont les types des éléments ne sont pas fixés.

Listes génériques

On souhaite définir une structure de donnée liste dont les types des éléments ne sont pas fixés.

Pour cela, on utilise un pointeur générique pour le champ qui contient l'élément de chaque cellule :

```
typedef struct _Cellule {  
    struct _Cellule *suiv;  
    void *e;  
} Cellule;
```

```
typedef Cellule *Liste;
```

Listes génériques

On souhaite définir une structure de donnée liste dont les types des éléments ne sont pas fixés.

Pour cela, on utilise un pointeur générique pour le champ qui contient l'élément de chaque cellule :

```
typedef struct _Cellule {  
    struct _Cellule *suiv;  
    void *e;  
} Cellule;
```

```
typedef Cellule *Liste;
```

Le type `Liste` permet ainsi de représenter des listes génériques.

Listes génériques

On souhaite définir une structure de donnée liste dont les types des éléments ne sont pas fixés.

Pour cela, on utilise un pointeur générique pour le champ qui contient l'élément de chaque cellule :

```
typedef struct _Cellule {  
    struct _Cellule *suiv;  
    void *e;  
} Cellule;
```

```
typedef Cellule *Liste;
```

Le type `Liste` permet ainsi de représenter des listes génériques.

C'est une structure de donnée générique car le **type des éléments** que les futures listes pourront contenir **n'est pas connu lors de l'écriture de la fonction**.

Listes génériques

La fonction

```
void aff_lst(Liste lst, void (*aff_elt)(void *)) {
    Cellule *x;

    assert(lst != NULL);
    assert(aff_elt != NULL);

    for (x = lst ; x != NULL ; x = x->suiv) {
        aff_elt(x->e);
        printf(" ");
    }
}
```

est une fonction générique pour l'affichage des éléments d'une liste générique.

Listes génériques

On l'utilise de la manière suivante (dans le cas ici d'une liste d'entiers).

```
void aff_int(void *e) {
    printf("%d", *((int *) e));
}
...
Liste lst;
int a, b, c;
a = 3; b = 14; c = 414;

lst = (Cellule *) malloc(sizeof(Cellule));
lst->e = &a;
lst->suiv = (Cellule *) malloc(sizeof(Cellule));
lst->suiv->e = &b;
lst->suiv->suiv = (Cellule *) malloc(sizeof(Cellule));
lst->suiv->suiv->e = &c;
lst->suiv->suiv->suiv = NULL;
aff_lst(lst, &aff_int);
```

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques.
Entre autres :

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques.
Entre autres :

1. `void aff_lst(Liste lst, void (*aff_elt)(void *));`

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques.
Entre autres :

1. `void aff_lst(Liste lst, void (*aff_elt)(void *));`
2. `void *elt_indice(Liste lst, int i);`

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques.
Entre autres :

1. `void aff_lst(Liste lst, void (*aff_elt)(void *));`
2. `void *elt_indice(Liste lst, int i);`
3. `int est_triee(Liste lst, int (*est_inf)(void *, void *));`

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques.
Entre autres :

1. `void aff_lst(Liste lst, void (*aff_elt)(void *));`
2. `void *elt_indice(Liste lst, int i);`
3. `int est_triee(Liste lst, int (*est_inf)(void *, void *));`
4. `void *max(Liste lst, int (*est_inf)(void *, void *));`

Listes génériques

Beaucoup de fonctions sur les listes peuvent ainsi être rendues génériques.
Entre autres :

1. `void aff_lst(Liste lst, void (*aff_elt)(void *));`
2. `void *elt_indice(Liste lst, int i);`
3. `int est_triee(Liste lst, int (*est_inf)(void *, void *));`
4. `void *max(Liste lst, int (*est_inf)(void *, void *));`

Les cas 3 et 4 supposent que les éléments représentés par les listes sont comparables au moyen d'une fonction `est_inf` à fournir.

Plan

Pointeurs de fonction

Principe

En paramètre et en retour

Généricité

Implantation de monoïdes

Monoïdes

En maths, un **monoïde** est un triplet $(\mathcal{M}, \bullet, \mathbf{1})$ où

Monoïdes

En maths, un **monoïde** est un triplet $(\mathcal{M}, \bullet, \mathbf{1})$ où

1. \mathcal{M} est un ensemble ;

Monoïdes

En maths, un **monoïde** est un triplet $(\mathcal{M}, \bullet, \mathbf{1})$ où

1. \mathcal{M} est un ensemble ;
2. \bullet est une application $\bullet : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$ associative, c.-à-d., pour tous $x, y, z \in \mathcal{M}$, on a $(x \bullet y) \bullet z = x \bullet (y \bullet z)$;

Monoïdes

En maths, un **monoïde** est un triplet $(\mathcal{M}, \bullet, \mathbf{1})$ où

1. \mathcal{M} est un ensemble ;
2. \bullet est une application $\bullet : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$ associative, c.-à-d., pour tous $x, y, z \in \mathcal{M}$, on a $(x \bullet y) \bullet z = x \bullet (y \bullet z)$;
3. $\mathbf{1} \in \mathcal{M}$ est un élément unitaire, c.-à-d., pour tout $x \in \mathcal{M}$, on a $x \bullet \mathbf{1} = x = \mathbf{1} \bullet x$.

Monoïdes

En maths, un **monoïde** est un triplet $(\mathcal{M}, \bullet, \mathbf{1})$ où

1. \mathcal{M} est un ensemble ;
2. \bullet est une application $\bullet : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$ associative, c.-à-d., pour tous $x, y, z \in \mathcal{M}$, on a $(x \bullet y) \bullet z = x \bullet (y \bullet z)$;
3. $\mathbf{1} \in \mathcal{M}$ est un élément unitaire, c.-à-d., pour tout $x \in \mathcal{M}$, on a $x \bullet \mathbf{1} = x = \mathbf{1} \bullet x$.

Exemples :

- ▶ $(\mathbb{N}, +, 0)$ est le **monoïde additif** des entiers naturels ;

Monoïdes

En maths, un **monoïde** est un triplet $(\mathcal{M}, \bullet, \mathbf{1})$ où

1. \mathcal{M} est un ensemble ;
2. \bullet est une application $\bullet : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$ associative, c.-à-d., pour tous $x, y, z \in \mathcal{M}$, on a $(x \bullet y) \bullet z = x \bullet (y \bullet z)$;
3. $\mathbf{1} \in \mathcal{M}$ est un élément unitaire, c.-à-d., pour tout $x \in \mathcal{M}$, on a $x \bullet \mathbf{1} = x = \mathbf{1} \bullet x$.

Exemples :

- ▶ $(\mathbb{N}, +, 0)$ est le **monoïde additif** des entiers naturels ;
- ▶ $(\mathbb{N}, \times, 1)$ est le **monoïde multiplicatif** des entiers naturels ;

Monoïdes

En maths, un **monoïde** est un triplet $(\mathcal{M}, \bullet, \mathbf{1})$ où

1. \mathcal{M} est un ensemble ;
2. \bullet est une application $\bullet : \mathcal{M} \times \mathcal{M} \rightarrow \mathcal{M}$ associative, c.-à-d., pour tous $x, y, z \in \mathcal{M}$, on a $(x \bullet y) \bullet z = x \bullet (y \bullet z)$;
3. $\mathbf{1} \in \mathcal{M}$ est un élément unitaire, c.-à-d., pour tout $x \in \mathcal{M}$, on a $x \bullet \mathbf{1} = x = \mathbf{1} \bullet x$.

Exemples :

- ▶ $(\mathbb{N}, +, 0)$ est le **monoïde additif** des entiers naturels ;
- ▶ $(\mathbb{N}, \times, 1)$ est le **monoïde multiplicatif** des entiers naturels ;
- ▶ $(\{a, b\}^*, \cdot, \epsilon)$ est le **monoïde libre** sur les lettres a et b. Ses éléments sont les chaînes de caractères composées de a et de b. Son produit \cdot est la concaténation et son unité ϵ est la chaîne vide.

Implantation de monoïdes

On se donne les deux objectifs suivants :

Implantation de monoïdes

On se donne les deux objectifs suivants :

1. implanter une **structure de donnée générique** pour **représenter des monoïdes** dont la nature des éléments n'est pas connue à l'avance ;

Implantation de monoïdes

On se donne les deux objectifs suivants :

1. implanter une **structure de donnée générique** pour **représenter des monoïdes** dont la nature des éléments n'est pas connue à l'avance ;
2. implanter une **fonction générique** qui **élève à la puissance $n \geq 0$** un élément x d'un monoïde.

Structure de donnée générique de monoïde

D'après la définition mathématique d'un monoïde, on aboutit à la définition suivante :

```
typedef struct {  
    void *unite;  
    void *(*produit)(void *, void *);  
} Monoide;
```

Structure de donnée générique de monoïde

D'après la définition mathématique d'un monoïde, on aboutit à la définition suivante :

```
typedef struct {  
    void *unite;  
    void *(*produit)(void *, void *);  
} Monoide;
```

On utilise des pointeurs génériques `void *` pour les éléments du monoïde.

Le champ `unite` est un pointeur générique vers l'unité du monoïde.

Le champ `produit` est un pointeur de fonction vers une fonction qui réalise le produit du monoïde.

Définition du monoïde additif des entiers naturels

On commence par définir une fonction qui réalise le produit de $(\mathbb{N}, +, 0)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *add(void *a, void *b) {
```

```
}
```

Définition du monoïde additif des entiers naturels

On commence par définir une fonction qui réalise le produit de $(\mathbb{N}, +, 0)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *add(void *a, void *b) {  
    int *res;  
  
    return (void *) res;  
}
```

Définition du monoïde additif des entiers naturels

On commence par définir une fonction qui réalise le produit de $(\mathbb{N}, +, 0)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *add(void *a, void *b) {  
    int *res;  
    res = (int *) malloc(sizeof(int));  
  
    return (void *) res;  
}
```

Définition du monoïde additif des entiers naturels

On commence par définir une fonction qui réalise le produit de $(\mathbb{N}, +, 0)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *add(void *a, void *b) {  
    int *res;  
    res = (int *) malloc(sizeof(int));  
    *res = *((int *) a) + *((int *) b);  
    return (void *) res;  
}
```

Définition du monoïde additif des entiers naturels

On commence par définir une fonction qui réalise le produit de $(\mathbb{N}, +, 0)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *add(void *a, void *b) {  
    int *res;  
    res = (int *) malloc(sizeof(int));  
    *res = *((int *) a) + *((int *) b);  
    return (void *) res;  
}
```

Ensuite, on crée le monoïde de la manière suivante :

```
Monoide m_add;  
int zero = 0;  
m_add.unite = &zero;  
m_add.produit = &add;
```

Définition du monoïde multiplicatif des entiers naturels

On commence par définir une fonction qui réalise le produit de $(\mathbb{N}, \times, 1)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *mul(void *a, void *b) {
```

```
}
```

Définition du monoïde multiplicatif des entiers naturels

On commence par définir une fonction qui réalise le produit de $(\mathbb{N}, \times, 1)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *mul(void *a, void *b) {  
    int *res;  
  
    return (void *) res;  
}
```

Ensuite, on crée le monoïde de la manière suivante :

```
Monoide m_mul;  
int un = 1;  
m_mul.unite = &un;  
m_mul.produit = &mul;
```

Définition du monoïde multiplicatif des entiers naturels

On commence par définir une fonction qui réalise le produit de $(\mathbb{N}, \times, 1)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *mul(void *a, void *b) {  
    int *res;  
    res = (int *) malloc(sizeof(int));  
  
    return (void *) res;  
}
```

Ensuite, on crée le monoïde de la manière suivante :

```
Monoide m_mul;  
int un = 1;  
m_mul.unite = &un;  
m_mul.produit = &mul;
```

Définition du monoïde multiplicatif des entiers naturels

On commence par définir une fonction qui réalise le produit de $(\mathbb{N}, \times, 1)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *mul(void *a, void *b) {  
    int *res;  
    res = (int *) malloc(sizeof(int));  
    *res = *((int *) a) * *((int *) b);  
    return (void *) res;  
}
```

Ensuite, on crée le monoïde de la manière suivante :

```
Monoide m_mul;  
int un = 1;  
m_mul.unite = &un;  
m_mul.produit = &mul;
```

Définition du monoïde libre sur a et b

On commence par définir une fonction qui réalise le produit de $(\{a, b\}^*, \cdot, \epsilon)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *conc(void *u, void *v) {
```

```
}
```

Définition du monoïde libre sur a et b

On commence par définir une fonction qui réalise le produit de $(\{a, b\}^*, \cdot, \epsilon)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *conc(void *u, void *v) {  
    char *res;  
  
    return (void *) res;  
}
```

Définition du monoïde libre sur a et b

On commence par définir une fonction qui réalise le produit de $(\{a, b\}^*, \cdot, \epsilon)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *conc(void *u, void *v) {
    char *res;
    int i, lu, lv;

    return (void *) res;
}
```

Définition du monoïde libre sur a et b

On commence par définir une fonction qui réalise le produit de $(\{a, b\}^*, \cdot, \epsilon)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *conc(void *u, void *v) {
    char *res;
    int i, lu, lv;
    lu = strlen((char *) u);
    lv = strlen((char *) v);

    return (void *) res;
}
```

Définition du monoïde libre sur a et b

On commence par définir une fonction qui réalise le produit de $(\{a, b\}^*, \cdot, \epsilon)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *conc(void *u, void *v) {
    char *res;
    int i, lu, lv;
    lu = strlen((char *) u);
    lv = strlen((char *) v);
    res = (char *) malloc(sizeof(char) * (1 + lu + lv));

    return (void *) res;
}
```

Définition du monoïde libre sur a et b

On commence par définir une fonction qui réalise le produit de $(\{a, b\}^*, \cdot, \epsilon)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *conc(void *u, void *v) {
    char *res;
    int i, lu, lv;
    lu = strlen((char *) u);
    lv = strlen((char *) v);
    res = (char *) malloc(sizeof(char) * (1 + lu + lv));
    for (i = 0 ; i < lu ; ++i) res[i] = ((char *) u)[i];
    for (i = 0 ; i < lv ; ++i) res[lu + i] = ((char *) v)[i];

    return (void *) res;
}
```

Définition du monoïde libre sur a et b

On commence par définir une fonction qui réalise le produit de $(\{a, b\}^*, \cdot, \epsilon)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *conc(void *u, void *v) {
    char *res;
    int i, lu, lv;
    lu = strlen((char *) u);
    lv = strlen((char *) v);
    res = (char *) malloc(sizeof(char) * (1 + lu + lv));
    for (i = 0 ; i < lu ; ++i) res[i] = ((char *) u)[i];
    for (i = 0 ; i < lv ; ++i) res[lu + i] = ((char *) v)[i];
    res[lu + lv] = '\0';
    return (void *) res;
}
```

Définition du monoïde libre sur a et b

On commence par définir une fonction qui réalise le produit de $(\{a, b\}^*, \cdot, \epsilon)$ et dont le type de retour et la signature respectent ceux du champ `produit` :

```
void *conc(void *u, void *v) {
    char *res;
    int i, lu, lv;
    lu = strlen((char *) u);
    lv = strlen((char *) v);
    res = (char *) malloc(sizeof(char) * (1 + lu + lv));
    for (i = 0 ; i < lu ; ++i) res[i] = ((char *) u)[i];
    for (i = 0 ; i < lv ; ++i) res[lu + i] = ((char *) v)[i];
    res[lu + lv] = '\0';
    return (void *) res;
}
```

Ensuite, on crée le monoïde de la manière suivante :

```
Monoide m_lib;                               m_lib.unite = &epsilon;
char epsilon[1] = "";                          m_lib.produit = &conc;
```

Fonction générique d'exponentiation

```
void *puissance(Monoide m, void *x, int n) {
```

```
}
```

Fonction générique d'exponentiation

```
void *puissance(Monoide m, void *x, int n) {  
    void *tmp;
```

```
}
```

Fonction générique d'exponentiation

```
void *puissance(Monoide m, void *x, int n) {  
    void *tmp;  
  
    if (n == 0)  
        return m.unite;  
  
}
```

Fonction générique d'exponentiation

```
void *puissance(Monoide m, void *x, int n) {  
    void *tmp;  
  
    if (n == 0)  
        return m.unite;  
    tmp = puissance(m, x, n / 2);  
  
}
```

Fonction générique d'exponentiation

```
void *puissance(Monoide m, void *x, int n) {  
    void *tmp;  
  
    if (n == 0)  
        return m.unite;  
    tmp = puissance(m, x, n / 2);  
    if (n % 2 == 0)  
        return m.produit(tmp, tmp);  
  
}
```

Fonction générique d'exponentiation

```
void *puissance(Monoide m, void *x, int n) {  
    void *tmp;  
  
    if (n == 0)  
        return m.unite;  
    tmp = puissance(m, x, n / 2);  
    if (n % 2 == 0)  
        return m.produit(tmp, tmp);  
    return m.produit(x, m.produit(tmp, tmp));  
}
```

Fonction générique d'exponentiation

```
void *puissance(Monoide m, void *x, int n) {  
    void *tmp;  
  
    if (n == 0)  
        return m.unite;  
    tmp = puissance(m, x, n / 2);  
    if (n % 2 == 0)  
        return m.produit(tmp, tmp);  
    return m.produit(x, m.produit(tmp, tmp));  
}
```

Cette fonction renvoie un pointeur générique sur une variable contenant le résultat de la valeur à l'adresse `x` élevée à la puissance `n`, pour le produit spécifié par le monoïde `m`.

Fonction générique d'exponentiation

```
void *puissance(Monoide m, void *x, int n) {  
    void *tmp;  
  
    if (n == 0)  
        return m.unite;  
    tmp = puissance(m, x, n / 2);  
    if (n % 2 == 0)  
        return m.produit(tmp, tmp);  
    return m.produit(x, m.produit(tmp, tmp));  
}
```

Cette fonction renvoie un pointeur générique sur une variable contenant le résultat de la valeur à l'adresse `x` élevée à la puissance `n`, pour le produit spécifié par le monoïde `m`.

Cette fonction utilise l'algorithme dit d'**exponentiation rapide**.

Application de la fonction d'exponentiation

```
int res, val = 5;  
res = *((int *) puissance(m_add, &val, 3));  
printf("%d\n", res);
```

Ceci affiche 5^3 dans le monoïde additif, c'est à dire 15.

Application de la fonction d'exponentiation

```
int res, val = 5;
res = *((int *) puissance(m_add, &val, 3));
printf("%d\n", res);
```

Ceci affiche 5^3 dans le monoïde additif, c'est à dire 15.

```
int res, val = 5;
res = *((int *) puissance(m_mul, &val, 3));
printf("%d\n", res);
```

Ceci affiche 5^3 dans le monoïde multiplicatif, c'est à dire 125.

Application de la fonction d'exponentiation

```
int res, val = 5;
res = *((int *) puissance(m_add, &val, 3));
printf("%d\n", res);
```

Ceci affiche 5^3 dans le monoïde additif, c'est à dire 15.

```
int res, val = 5;
res = *((int *) puissance(m_mul, &val, 3));
printf("%d\n", res);
```

Ceci affiche 5^3 dans le monoïde multiplicatif, c'est à dire 125.

```
char *res, ch[3] = "ab";
res = (char *) puissance(m_lib, &ch, 4);
printf("%s\n", res);
```

Ceci affiche $(ab)^4$ dans le monoïde libre, c'est à dire abababab.