

Entrées et sorties

Sortie

Entrée

Fichiers

Fichiers binaires

Fichiers binaires — introduction

À la différence des fichiers textes qui sont constitués de caractères (et sont donc potentiellement lisibles par un être humain directement), les **fichiers binaires** contiennent des suites de bits **non formatées**.

Fichiers binaires — introduction

À la différence des fichiers textes qui sont constitués de caractères (et sont donc potentiellement lisibles par un être humain directement), les **fichiers binaires** contiennent des suites de bits **non formatées**.

Par exemple, si `ft` pointe vers un fichier texte, l'instruction

```
fprintf(ft, "%d", 261)
```

place les caractères `'2'`, `'6'` et `'1'` dans le fichier en question.

Fichiers binaires — introduction

À la différence des fichiers textes qui sont constitués de caractères (et sont donc potentiellement lisibles par un être humain directement), les **fichiers binaires** contiennent des suites de bits **non formatées**.

Par exemple, si `ft` pointe vers un fichier texte, l'instruction

```
fprintf(ft, "%d", 261)
```

place les caractères '2', '6' et '1' dans le fichier en question. Ceci écrit donc les trois octets `00110010`, `00110110` et `00110001`, obtenus en interprétant en base deux les valeurs ASCII de ces caractères décimaux.

Fichiers binaires — introduction

À la différence des fichiers textes qui sont constitués de caractères (et sont donc potentiellement lisibles par un être humain directement), les **fichiers binaires** contiennent des suites de bits **non formatées**.

Par exemple, si `ft` pointe vers un fichier texte, l'instruction

```
fprintf(ft, "%d", 261)
```

place les caractères '2', '6' et '1' dans le fichier en question. Ceci écrit donc les trois octets `00110010`, `00110110` et `00110001`, obtenus en interprétant en base deux les valeurs ASCII de ces caractères décimaux.

En revanche, si `fb` pointe vers un fichier binaire, on emploiera l'instruction d'écriture `fwrite` et, si `v` est une variable entière de valeur `261`,

```
fwrite(&v, sizeof(int), 1, fb)
```

écrit dans le fichier les 4 (= `sizeof(int)`) octets `00000000`, `00000000`, `00000001`, `00000101` à la suite (développement en base deux de la valeur).

Fichiers binaires — introduction

À la différence des fichiers textes qui sont constitués de caractères (et sont donc potentiellement lisibles par un être humain directement), les **fichiers binaires** contiennent des suites de bits **non formatées**.

Par exemple, si `ft` pointe vers un fichier texte, l'instruction

```
fprintf(ft, "%d", 261)
```

place les caractères '2', '6' et '1' dans le fichier en question. Ceci écrit donc les trois octets `00110010`, `00110110` et `00110001`, obtenus en interprétant en base deux les valeurs ASCII de ces caractères décimaux.

En revanche, si `fb` pointe vers un fichier texte, on emploiera l'instruction d'écriture `fwrite` et, si `v` est une variable entière de valeur `261`,

```
fwrite(&v, sizeof(int), 1, fb)
```

écrit dans le fichier les 4 (= `sizeof(int)`) octets `00000000`, `00000000`, `00000001`, `00000101` à la suite (développement en base deux de la valeur).

Ainsi, on écrit/lit dans/depuis un fichier binaire directement des valeurs, sans se soucier du format à adopter.

Ouverture de fichiers en mode binaire

On utilise les modes d'ouverture habituels avec un **b** en plus pour signaler l'ouverture en binaire.

- ▶ **"rb"** : lecture binaire seule.
- ▶ **"wb"** : écriture binaire seule. Si le fichier n'existe pas, il est créé.
- ▶ **"ab"** : écriture binaire en ajout. Permet d'écrire dans le fichier en partant de la fin. Si le fichier n'existe pas, il est créé.
- ▶ **"rb+"** : lecture binaire et écriture binaire .
- ▶ **"wb+"** : lecture binaire et écriture binaire avec suppression préalable du contenu du fichier. Si le fichier n'existe pas, il est créé.
- ▶ **"ab+"** : lecture binaire et écriture binaire en ajout. Permet de lire et d'écrire dans le fichier en partant de la fin. Si le fichier n'existe pas, il est créé.

Écriture dans un fichier binaire

On utilise la fonction

```
int fwrite(void *ptr, int taille, int nb, FILE *f);
```

pour **écrire** dans un fichier pointé par **f** ouvert en **mode binaire**.

Écriture dans un fichier binaire

On utilise la fonction

```
int fwrite(void *ptr, int taille, int nb, FILE *f);
```

pour **écrire** dans un fichier pointé par **f** ouvert en **mode binaire**.

Cette fonction écrit les **nb** valeurs de taille **taille** octets pointées par le pointeur **ptr** dans le fichier pointé par **f**.

Écriture dans un fichier binaire

On utilise la fonction

```
int fwrite(void *ptr, int taille, int nb, FILE *f);
```

pour **écrire** dans un fichier pointé par **f** ouvert en **mode binaire**.

Cette fonction écrit les **nb** valeurs de taille **taille** octets pointées par le pointeur **ptr** dans le fichier pointé par **f**.

Cette fonction renvoie le nombre de valeurs écrites.

Exemple d'utilisation de `fwrite`

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

Exemple d'utilisation de fwrite

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

```
typedef struct {  
    unsigned char rouge;  
    unsigned char bleu;  
    unsigned char vert;  
} Couleur;
```

Exemple d'utilisation de fwrite

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

```
#include <stdio.h>

typedef struct {
    unsigned char rouge;
    unsigned char bleu;
    unsigned char vert;
} Couleur;

int main() {
    FILE *f;
    Couleur coul;
    coul.rouge = 120; coul.bleu = 200; coul.vert = 12;

    return 0;
}
```

Exemple d'utilisation de fwrite

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

```
#include <stdio.h>

typedef struct {
    unsigned char rouge;
    unsigned char bleu;
    unsigned char vert;
} Couleur;

int main() {
    FILE *f;
    Couleur coul;
    coul.rouge = 120; coul.bleu = 200; coul.vert = 12;

    f = fopen("fic", "wb");

    return 0;
}
```

Exemple d'utilisation de fwrite

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

```
#include <stdio.h>

typedef struct {
    unsigned char rouge;
    unsigned char bleu;
    unsigned char vert;
} Couleur;

int main() {
    FILE *f;
    Couleur coul;
    coul.rouge = 120; coul.bleu = 200; coul.vert = 12;

    f = fopen("fic", "wb");
    fwrite(&coul, sizeof(Couleur), 1, f);

    return 0;
}
```

Exemple d'utilisation de fwrite

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

```
#include <stdio.h>

typedef struct {
    unsigned char rouge;
    unsigned char bleu;
    unsigned char vert;
} Couleur;

int main() {
    FILE *f;
    Couleur coul;
    coul.rouge = 120; coul.bleu = 200; coul.vert = 12;

    f = fopen("fic", "wb");
    fwrite(&coul, sizeof(Couleur), 1, f);
    fclose(f);

    return 0;
}
```

Exemple d'utilisation de fwrite

Ce programme écrit en mode binaire dans le fichier `fic` la valeur d'une variable d'un type structuré.

```
#include <stdio.h>

typedef struct {
    unsigned char rouge;
    unsigned char bleu;
    unsigned char vert;
} Couleur;

int main() {
    FILE *f;
    Couleur coul;
    coul.rouge = 120; coul.bleu = 200; coul.vert = 12;

    f = fopen("fic", "wb");
    fwrite(&coul, sizeof(Couleur), 1, f);
    fclose(f);

    return 0;
}
```

Remarque : il n'a pas été nécessaire ici de définir un format d'écriture (ce qui aurait été imposé en cas d'écriture dans un fichier texte).

Lecture dans un fichier binaire

On utilise la fonction

```
int fread(void *ptr, int taille, int nb, FILE *f);
```

pour lire dans un fichier pointé par `f` ouvert en mode binaire.

Lecture dans un fichier binaire

On utilise la fonction

```
int fread(void *ptr, int taille, int nb, FILE *f);
```

pour lire dans un fichier pointé par `f` ouvert en mode binaire.

Cette fonction lit `nb` valeurs de taille `taille` octets dans le fichier pointé par `f` et les place à l'adresse pointée par le pointeur `ptr`.

Lecture dans un fichier binaire

On utilise la fonction

```
int fread(void *ptr, int taille, int nb, FILE *f);
```

pour lire dans un fichier pointé par `f` ouvert en mode binaire.

Cette fonction lit `nb` valeurs de taille `taille` octets dans le fichier pointé par `f` et les place à l'adresse pointée par le pointeur `ptr`.

Cette fonction renvoie le nombre de valeurs lues.

Exemple d'utilisation de fread

Ce programme écrit en mode binaire dans le fichier `fic` un tableau d'entiers et le lit.

Exemple d'utilisation de fread

Ce programme écrit en mode binaire dans le fichier `fic` un tableau d'entiers et le lit.

```
#include <stdio.h>

int main() {
    FILE *f;
    int i, tab_1[12], tab_2[12];

    for (i = 0 ; i < 12 ; ++i)
        tab_1[i] = i;

    return 0;
}
```

Exemple d'utilisation de fread

Ce programme écrit en mode binaire dans le fichier `fic` un tableau d'entiers et le lit.

```
#include <stdio.h>

int main() {
    FILE *f;
    int i, tab_1[12], tab_2[12];

    for (i = 0 ; i < 12 ; ++i)
        tab_1[i] = i;

    f = fopen("fic", "wb");

    return 0;
}
```

Exemple d'utilisation de fread

Ce programme écrit en mode binaire dans le fichier `fic` un tableau d'entiers et le lit.

```
#include <stdio.h>

int main() {
    FILE *f;
    int i, tab_1[12], tab_2[12];

    for (i = 0 ; i < 12 ; ++i)
        tab_1[i] = i;

    f = fopen("fic", "wb");
    fwrite(tab_1, sizeof(int), 12, f);

    return 0;
}
```

Exemple d'utilisation de fread

Ce programme écrit en mode binaire dans le fichier `fic` un tableau d'entiers et le lit.

```
#include <stdio.h>

int main() {
    FILE *f;
    int i, tab_1[12], tab_2[12];

    for (i = 0 ; i < 12 ; ++i)
        tab_1[i] = i;

    f = fopen("fic", "wb");
    fwrite(tab_1, sizeof(int), 12, f);
    fclose(f);

    return 0;
}
```

Exemple d'utilisation de fread

Ce programme écrit en mode binaire dans le fichier `fic` un tableau d'entiers et le lit.

```
#include <stdio.h>

int main() {
    FILE *f;
    int i, tab_1[12], tab_2[12];

    for (i = 0 ; i < 12 ; ++i)
        tab_1[i] = i;

    f = fopen("fic", "wb");
    fwrite(tab_1, sizeof(int), 12, f);
    fclose(f);

    f = fopen("fic", "rb");

    return 0;
}
```

Exemple d'utilisation de fread

Ce programme écrit en mode binaire dans le fichier `fic` un tableau d'entiers et le lit.

```
#include <stdio.h>

int main() {
    FILE *f;
    int i, tab_1[12], tab_2[12];

    for (i = 0 ; i < 12 ; ++i)
        tab_1[i] = i;

    f = fopen("fic", "wb");
    fwrite(tab_1, sizeof(int), 12, f);
    fclose(f);

    f = fopen("fic", "rb");
    fread(tab_2, sizeof(int), 12, f);

    return 0;
}
```

Exemple d'utilisation de fread

Ce programme écrit en mode binaire dans le fichier `fic` un tableau d'entiers et le lit.

```
#include <stdio.h>

int main() {
    FILE *f;
    int i, tab_1[12], tab_2[12];

    for (i = 0 ; i < 12 ; ++i)
        tab_1[i] = i;

    f = fopen("fic", "wb");
    fwrite(tab_1, sizeof(int), 12, f);
    fclose(f);

    f = fopen("fic", "rb");
    fread(tab_2, sizeof(int), 12, f);
    fclose(f);

    return 0;
}
```

Exemple d'utilisation de fread

Ce programme écrit en mode binaire dans le fichier `fic` un tableau d'entiers et le lit.

```
#include <stdio.h>

int main() {
    FILE *f;
    int i, tab_1[12], tab_2[12];

    for (i = 0 ; i < 12 ; ++i)
        tab_1[i] = i;

    f = fopen("fic", "wb");
    fwrite(tab_1, sizeof(int), 12, f);
    fclose(f);

    f = fopen("fic", "rb");
    fread(tab_2, sizeof(int), 12, f);
    fclose(f);

    return 0;
}
```

Remarque : il n'a pas été nécessaire de se baser sur un format pour la lecture (ce qui aurait été imposé en cas de lecture depuis un fichier texte).

Plan

Types

- Notion de type

- Types scalaires

- Types construits

Plan

Types

Notion de type

Types scalaires

Types construits

Un **type** peut être vu comme un ensemble (fini ou infini) de valeurs.

Types

Un **type** peut être vu comme un ensemble (fini ou infini) de valeurs.

Dire qu'une variable **x** est de type **T** signifie que la valeur de **x** est dans **T**.

Types

Un **type** peut être vu comme un ensemble (fini ou infini) de valeurs.

Dire qu'une variable **x** est de type **T** signifie que la valeur de **x** est dans **T**.

Il existe deux sortes de types :

1. les **types scalaires**, qui sont des types atomiques et définis à l'avance dans le langage ;

Types

Un **type** peut être vu comme un ensemble (fini ou infini) de valeurs.

Dire qu'une variable **x** est de type **T** signifie que la valeur de **x** est dans **T**.

Il existe deux sortes de types :

1. les **types scalaires**, qui sont des types atomiques et définis à l'avance dans le langage ;
2. les **types composites**, qui sont des assemblages de types scalaires ou de types composites par le biais des constructions **struct**, **enum** ou tableau.

Type d'une variable

Le type d'une variable indique comment **interpréter** la zone mémoire qui lui est attribuée ainsi que sa **taille**.

Type d'une variable

Le type d'une variable indique comment **interpréter** la zone mémoire qui lui est attribuée ainsi que sa **taille**.

L'opérateur `sizeof` permet de connaître la taille en octets d'un type. On peut aussi l'appliquer à une valeur. P.ex., `sizeof(int)` et `sizeof(33)` valent `4`.

Type d'une variable

Le type d'une variable indique comment **interpréter** la zone mémoire qui lui est attribuée ainsi que sa **taille**.

L'opérateur `sizeof` permet de connaître la taille en octets d'un type. On peut aussi l'appliquer à une valeur. P.ex., `sizeof(int)` et `sizeof(33)` valent `4`.

```
int t;  
char c;  
  
printf("%d ", sizeof(int));  
printf("%d ", sizeof(t));  
printf("%d ", sizeof(31));  
  
printf("%d ", sizeof(char));  
printf("%d ", sizeof(c));
```

```
printf("%d\n", sizeof('a'));
```

Ceci affiche `4 4 4 1 1 4`.

L'expression `sizeof('a')` vaut `4`.
En effet, même si `'a'` est un caractère, c'est avant tout un entier.
La conversion est implicite.

Type d'une variable

Le type d'une variable indique comment **interpréter** la zone mémoire qui lui est attribuée ainsi que sa **taille**.

L'opérateur `sizeof` permet de connaître la taille en octets d'un type. On peut aussi l'appliquer à une valeur. P.ex., `sizeof(int)` et `sizeof(33)` valent 4.

```
int t;  
char c;  
  
printf("%d ", sizeof(int));  
printf("%d ", sizeof(t));  
printf("%d ", sizeof(31));  
  
printf("%d ", sizeof(char));  
printf("%d ", sizeof(c));
```

```
printf("%d\n", sizeof('a'));
```

Ceci affiche 4 4 4 1 1 4.

L'expression `sizeof('a')` vaut 4.
En effet, même si 'a' est un caractère, c'est avant tout un entier.
La conversion est implicite.

Le type d'une variable est **attribué à sa déclaration** et ne peut pas être modifié.

Types

Notion de type

Types scalaires

Types construits

Types entier

On se place sur une machine 64 bits.

Nom	Taille (octets)	Plage
<code>char</code>	1	-128 à 127
<code>short</code>	2	-32768 à 32767
<code>int</code>	4	-2^{31} à $2^{31} - 1$
<code>long</code>	8	-2^{63} à $2^{63} - 1$

Types entier

On se place sur une machine 64 bits.

Nom	Taille (octets)	Plage
<code>char</code>	1	-128 à 127
<code>short</code>	2	-32768 à 32767
<code>int</code>	4	-2^{31} à $2^{31} - 1$
<code>long</code>	8	-2^{63} à $2^{63} - 1$

Chacun de ces types peut être précédé de `unsigned` pour faire en sorte de ne représenter que des entiers positifs. On a ainsi les plages suivantes :

Nom	Plage
<code>unsigned char</code>	0 à 255
<code>unsigned short</code>	0 à 65535
<code>unsigned int</code>	0 à $2^{32} - 1$
<code>unsigned long</code>	0 à $2^{64} - 1$

Entiers non signés

Dans le cas où l'on a besoin de représenter uniquement des valeurs entières positives, on utilisera les versions non signées des types entiers.

Entiers non signés

Dans le cas où l'on a besoin de représenter uniquement des valeurs entières positives, on utilisera les versions non signées des types entiers.

Quelques avantages de ce procédé :

1. possibilité de représenter des entiers plus grands;
2. gain de lisibilité du programme.

Entiers non signés

Dans le cas où l'on a besoin de représenter uniquement des valeurs entières positives, on utilisera les versions non signées des types entiers.

Quelques avantages de ce procédé :

1. possibilité de représenter des entiers plus grands;
2. gain de lisibilité du programme.

Attention : les instructions

```
unsigned int i;  
for (i = 8 ; i >= 0 ; --i) {  
    ...  
}
```

produisent une boucle infinie. En effet, `i` étant non signé, il est toujours positif et donc la condition `i >= 0` est toujours vraie.

Constantes entières

Il existe plusieurs manières d'exprimer des **constantes entières** :

- ▶ en base dix : 0, 29, -322, ...

Constantes entières

Il existe plusieurs manières d'exprimer des **constantes entières** :

- ▶ en base dix : 0, 29, -322, ...
- ▶ en octal : 01, 0145, -01234567, ...

Constantes entières

Il existe plusieurs manières d'exprimer des **constantes entières** :

- ▶ en base dix : 0, 29, -322, ...
- ▶ en octal : 01, 0145, -01234567, ...
- ▶ en hexadécimal : 0x1, 0x5555FFFF, -0x98879AFA, ...

Constantes entières

Il existe plusieurs manières d'exprimer des **constantes entières** :

- ▶ en base dix : `0`, `29`, `-322`, ...
- ▶ en octal : `01`, `0145`, `-01234567`, ...
- ▶ en hexadécimal : `0x1`, `0x5555FFFF`, `-0x98879AFA`, ...
- ▶ par un caractère : `'a'`, `'9'`, `'*'`, `'\n'`, ...

Constantes entières

Il existe plusieurs manières d'exprimer des **constantes entières** :

- ▶ en base dix : `0`, `29`, `-322`, ...
- ▶ en octal : `01`, `0145`, `-01234567`, ...
- ▶ en hexadécimal : `0x1`, `0x5555FFFF`, `-0x98879AFA`, ...
- ▶ par un caractère : `'a'`, `'9'`, `'*'`, `'\n'`, ...

Un entier peut être représenté par un caractère car tout caractère est représenté par son code ASCII (qui est un entier compris entre `0` et `127`).

Constantes entières

Il existe plusieurs manières d'exprimer des **constantes entières** :

- ▶ en base dix : `0`, `29`, `-322`, ...
- ▶ en octal : `01`, `0145`, `-01234567`, ...
- ▶ en hexadécimal : `0x1`, `0x5555FFFF`, `-0x98879AFA`, ...
- ▶ par un caractère : `'a'`, `'9'`, `'*'`, `'\n'`, ...

Un entier peut être représenté par un caractère car tout caractère est représenté par son code ASCII (qui est un entier compris entre `0` et `127`).

Attention : ne pas confondre les caractères chiffres avec les entiers (l'entier `'1'` vaut `49` et non pas `1`).

Types flottant

On se place sur une machine 64 bits.

Nom	Taille (octets)	Valeur absolue maximale
<code>float</code>	4	3.40282×10^{38}
<code>double</code>	8	1.79769×10^{308}
<code>long double</code>	16	1.18973×10^{4932}

Le fichier d'en-tête `float.h` contient des constantes donnant d'autres renseignements sur les types flottant.

Danger des types flottant

```
float x = 10000001.0;  
printf("%f\n", x);
```

Ces instructions affichent, de manière attendue,
10000001.000000.

Danger des types flottant

```
float x = 10000001.0;  
printf("%f\n", x);
```

Ces instructions affichent, de manière attendue,
10000001.000000.

```
float x = 100000001.0;  
printf("%f\n", x);
```

En revanche, ces instructions affichent, de manière inattendue,
100000000.000000.

Danger des types flottant

```
float x = 10000001.0;  
printf("%f\n", x);
```

Ces instructions affichent, de manière attendue,
10000001.000000.

```
float x = 100000001.0;  
printf("%f\n", x);
```

En revanche, ces instructions affichent, de manière inattendue,
100000000.000000.

Les nombres flottants sont représentés de manière **approchée**.

Comme ces exemples le montrent, même certains entiers, représentables de manière exacte par des types entier, ne le sont pas par des types flottant.

Danger des types flottant : une solution partielle

Les types flottant présentent divers désavantages par rapport aux types entier :

1. **représentation non exacte** des nombres;
2. opérations arithmétiques beaucoup **moins efficaces**.

Danger des types flottant : une solution partielle

Les types flottant présentent divers désavantages par rapport aux types entier :

1. **représentation non exacte** des nombres;
2. opérations arithmétiques beaucoup **moins efficaces**.

Pour ces raisons, **il est recommandé de ne jamais utiliser de types flottant.**

Danger des types flottant : une solution partielle

Les types flottant présentent divers désavantages par rapport aux types entier :

1. **représentation non exacte** des nombres;
2. opérations arithmétiques beaucoup **moins efficaces**.

Pour ces raisons, **il est recommandé de ne jamais utiliser de types flottant.**

Solution partielle : on représente par l'entier $x \times 10^k$ tout nombre x qui dispose de $k \geq 0$ chiffres (en base dix) après la virgule, k étant fixé.

Danger des types flottant : une solution partielle

Les types flottant présentent divers désavantages par rapport aux types entier :

1. **représentation non exacte** des nombres;
2. opérations arithmétiques beaucoup **moins efficaces**.

Pour ces raisons, **il est recommandé de ne jamais utiliser de types flottant**.

Solution partielle : on représente par l'entier $x \times 10^k$ tout nombre x qui dispose de $k \geq 0$ chiffres (en base dix) après la virgule, k étant fixé.

P.ex., si l'on a besoin de manipuler des nombres à $k := 2$ chiffres après la virgule, les nombres 0.15 et 331.9 sont respectivement représentés par les entiers 15 et 33190.

Opérations sur les types scalaires

Les valeurs d'un type scalaire (entier ou flottant) forment un **ensemble totalement ordonné** : étant donné deux valeurs, il est toujours possible de les comparer. On utilise pour cela les **opérateurs relationnels**

`==, !=, <=, >=, <, >`.

Opérations sur les types scalaires

Les valeurs d'un type scalaire (entier ou flottant) forment un **ensemble totalement ordonné** : étant donné deux valeurs, il est toujours possible de les comparer. On utilise pour cela les **opérateurs relationnels**

`==, !=, <=, >=, <, >`.

Il est possible de mélanger des comparaisons de valeurs de types entier et de types flottant. Dans ce cas, les entiers sont convertis implicitement en une valeur de type flottant avant d'effectuer la comparaison.

Opérations sur les types scalaires

Les valeurs d'un type scalaire (entier ou flottant) forment un **ensemble totalement ordonné** : étant donné deux valeurs, il est toujours possible de les comparer. On utilise pour cela les **opérateurs relationnels**

`==, !=, <=, >=, <, >`.

Il est possible de mélanger des comparaisons de valeurs de types entier et de types flottant. Dans ce cas, les entiers sont convertis implicitement en une valeur de type flottant avant d'effectuer la comparaison.

Sur des variables de type scalaire sont définis les **opérateurs arithmétiques**

`+, -, *, /, ++, --`.

Les opérateurs `++` et `--` servent à additionner ou à retrancher de 1 la valeur des variables sur lesquels ils sont appliqués.

Plan

Types

Notion de type

Types scalaires

Types construits

Types structurés

La syntaxe

```
typedef struct ALIAS {  
    TYPE_1 CHAMP_1;  
    TYPE_2 CHAMP_2;  
    ...  
} NOM;
```

permet de **déclarer un type structuré** **NOM**, constitué des **champs** **CHAMP_1**, **CHAMP_2**, L'**alias** **ALIAS** est facultatif.

Types structurés

La syntaxe

```
typedef struct ALIAS {  
    TYPE_1 CHAMP_1;  
    TYPE_2 CHAMP_2;  
    ...  
} NOM;
```

permet de **déclarer un type structuré** `NOM`, constitué des **champs** `CHAMP_1`, `CHAMP_2`, L'**alias** `ALIAS` est facultatif.

C'est un **amalgame** de types.

Types structurés

La syntaxe

```
typedef struct ALIAS {  
    TYPE_1 CHAMP_1;  
    TYPE_2 CHAMP_2;  
    ...  
} NOM;
```

permet de **déclarer un type structuré** `NOM`, constitué des **champs** `CHAMP_1`, `CHAMP_2`, ... L'**alias** `ALIAS` est facultatif.

C'est un **amalgame** de types.

P.ex.,

```
typedef struct {  
    int x;  
    int y;  
} Couple;
```

déclare un type structuré `Couple`
qui permet de représenter des
couples d'entiers.

Types structurés

Si `x` est une variable d'un type structuré `T` contenant le champ `ch`, on **accède** à ce champ par la syntaxe

`x.ch`

Types structurés

Si `x` est une variable d'un type structuré `T` contenant le champ `ch`, on **accède** à ce champ par la syntaxe

```
x.ch
```

Si `adr_x` est une adresse sur une variable de type `T`, on accède à ce même champ par la syntaxe

```
adr_x->ch
```

Types structurés

Si `x` est une variable d'un type structuré `T` contenant le champ `ch`, on **accède** à ce champ par la syntaxe

```
x.ch
```

Si `adr_x` est une adresse sur une variable de type `T`, on accède à ce même champ par la syntaxe

```
adr_x->ch
```

Cette syntaxe est un raccourci pour

```
(*adr_x).ch
```

Types structurés

Si `x` est une variable d'un type structuré `T` contenant le champ `ch`, on **accède** à ce champ par la syntaxe

`x.ch`

Si `adr_x` est une adresse sur une variable de type `T`, on accède à ce même champ par la syntaxe

`adr_x->ch`

Cette syntaxe est un raccourci pour

`(*adr_x).ch`

P.ex., les trois suites d'instructions suivantes sont équivalentes :

Couple `*c;`

...

`c->x = c->x + 1;`

Couple `*c;`

...

`*(c).x = c->x + 1;`

Couple `*c;`

...

`c->x = (*c).x + 1;`

Opérations sur les types structurés

Les **opérateurs relationnels** ne sont pas définis sur les types structurés.

Il est donc impossible de tester si deux variables d'un même type structuré sont égales au moyen de l'opérateur `==`. Il faut tester l'égalité de chacun des champs qui les constituent.

Opérations sur les types structurés

Les **opérateurs relationnels** ne sont pas définis sur les types structurés.

Il est donc impossible de tester si deux variables d'un même type structuré sont égales au moyen de l'opérateur `==`. Il faut tester l'égalité de chacun des champs qui les constituent.

En revanche, l'**opérateur d'affectation** `=` est compatible avec les types structurés.

Opérations sur les types structurés

Les **opérateurs relationnels** ne sont pas définis sur les types structurés.

Il est donc impossible de tester si deux variables d'un même type structuré sont égales au moyen de l'opérateur `==`. Il faut tester l'égalité de chacun des champs qui les constituent.

En revanche, l'**opérateur d'affectation** `=` est compatible avec les types structurés.

```
typedef struct {
    char nom[32];
    char prenom[32];
    int age;
} Personne;
...
Personne p1, p2;
scanf(" %s", p1.nom);
scanf(" %s", p1.prenom);
p1.age = 30;
p2 = p1;
```

L'affectation en dernière ligne fait en sorte que tous les champs de `p2` contiennent les mêmes valeurs que ceux de `p1`.

Opérations sur les types structurés

Les **opérateurs relationnels** ne sont pas définis sur les types structurés.

Il est donc impossible de tester si deux variables d'un même type structuré sont égales au moyen de l'opérateur `==`. Il faut tester l'égalité de chacun des champs qui les constituent.

En revanche, l'**opérateur d'affectation** `=` est compatible avec les types structurés.

```
typedef struct {
    char nom[32];
    char prenom[32];
    int age;
} Personne;
...
Personne p1, p2;
scanf(" %s", p1.nom);
scanf(" %s", p1.prenom);
p1.age = 30;
p2 = p1;
```

L'affectation en dernière ligne fait en sorte que tous les champs de `p2` contiennent les mêmes valeurs que ceux de `p1`.

Il y a recopie des tableaux statiques `p1.nom` et `p1.prenom` dans `p2.nom` et `p2.prenom`.

Ce phénomène va être étudié en détail plus loin.

Types énumérés

La syntaxe

```
typedef enum {  
    ENU_1,  
    ENU_2,  
    ...  
} NOM;
```

permet de **déclarer un type énuméré** **NOM**, constitué des **énumérateurs** **ENU_1, ENU_2, ...** (Attention, on utilise des **,** et non pas des **;**.)

Types énumérés

La syntaxe

```
typedef enum {  
    ENU_1,  
    ENU_2,  
    ...  
} NOM;
```

permet de **déclarer un type énuméré** **NOM**, constitué des **énumérateurs** **ENU_1**, **ENU_2**, ... (Attention, on utilise des **,** et non pas des **;**.)

Une valeur de ce type prend pour valeur exactement un des énumérateurs qui le constituent.

Types énumérés

La syntaxe

```
typedef enum {  
    ENU_1,  
    ENU_2,  
    ...  
} NOM;
```

permet de **déclarer un type énuméré** **NOM**, constitué des **énumérateurs** **ENU_1**, **ENU_2**, ... (Attention, on utilise des **,** et non pas des **;**.)

Une valeur de ce type prend pour valeur exactement un des énumérateurs qui le constituent.

P.ex.,

```
typedef enum {  
    FAUX,  
    VRAI  
} Boolean;
```

est un type qui permet de représenter des booléens.

Une valeur de type **Boolean** est soit **FAUX**, soit **VRAI**.

Types énumérés

```
typedef enum {
    LUNDI,      /* = 0 */
    MARDI,      /* = 1 */
    MERCREDI,   /* = 2 */
    JEUDI,      /* = 3 */
    VENDREDI,   /* = 4 */
    SAMEDI,     /* = 5 */
    DIMANCHE    /* = 6 */
} Jour;
...
printf("%d\n", MERCREDI);
```

Les énumérateurs sont des **expressions entières**. Leur valeur est déterminée par leur ordre de déclaration dans le type.

Ces instructions affichent **2**. En effet, **LUNDI** vaut **0** car il est le 1^{er} énumérateur déclaré et les valeurs des suivants s'incrémentent selon leur ordre de déclaration.

Types énumérés

```
typedef enum {
    LUNDI,      /* = 0 */
    MARDI,      /* = 1 */
    MERCREDI,   /* = 2 */
    JEUDI,      /* = 3 */
    VENDREDI,   /* = 4 */
    SAMEDI,     /* = 5 */
    DIMANCHE    /* = 6 */
} Jour;
...
printf("%d\n", MERCREDI);
```

```
typedef enum {
    LA = 0,
    SI = 2,
    DO,      /* = 3 */
    RE = 5,
    MI = 7,
    FA = 8,
    SOL = 10
} Note;
```

Les énumérateurs sont des **expressions entières**. Leur valeur est déterminée par leur ordre de déclaration dans le type.

Ces instructions affichent **2**. En effet, **LUNDI** vaut **0** car il est le 1^{er} énumérateur déclaré et les valeurs des suivants s'incrémentent selon leur ordre de déclaration.

Il est possible de spécifier manuellement les valeurs des énumérateurs avec la syntaxe **ENU = VAL** où **ENU** est un énumérateur et **VAL** une constante entière.

Si une valeur n'est pas spécifiée, elle est déduite de la précédente en l'incrémentant.

Types énumérés

L'utilisation de branchement `switch` est particulièrement adaptée pour traiter une variable d'un type énuméré.

Types énumérés

L'utilisation de branchement `switch` est particulièrement adaptée pour traiter une variable d'un type énuméré.

```
Note note;

scanf(" %d", &note);

switch (note) {
    case LA : printf("A"); break;
    case SI : printf("B"); break;
    case DO : printf("C"); break;
    case RE : printf("D"); break;
    case MI : printf("E"); break;
    case FA : printf("F"); break;
    case SOL : printf("G");break;
    default : printf(
        "%d non note", note);
}
```

Ces instructions lisent une valeur entière (représentant une `Note`) sur l'entrée standard et l'affichent (en notation internationale).

Types énumérés

L'utilisation de branchement `switch` est particulièrement adaptée pour traiter une variable d'un type énuméré.

```
Note note;

scanf(" %d", &note);

switch (note) {
    case LA : printf("A"); break;
    case SI : printf("B"); break;
    case DO : printf("C"); break;
    case RE : printf("D"); break;
    case MI : printf("E"); break;
    case FA : printf("F"); break;
    case SOL : printf("G");break;
    default : printf(
        "%d non note", note);
}
```

Ces instructions lisent une valeur entière (représentant une `Note`) sur l'entrée standard et l'affichent (en notation internationale).

Remarque : une variable d'un type énuméré peut prendre comme valeur n'importe quel entier. Ceci explique la présence de la clause `default`.

Types énumérés

L'utilisation de branchement `switch` est particulièrement adaptée pour traiter une variable d'un type énuméré.

```
Note note;

scanf(" %d", &note);

switch (note) {
    case LA : printf("A"); break;
    case SI : printf("B"); break;
    case DO : printf("C"); break;
    case RE : printf("D"); break;
    case MI : printf("E"); break;
    case FA : printf("F"); break;
    case SOL : printf("G"); break;
    default : printf(
        "%d non note", note);
}
```

Ces instructions lisent une valeur entière (représentant une `Note`) sur l'entrée standard et l'affichent (en notation internationale).

Remarque : une variable d'un type énuméré peut prendre comme valeur n'importe quel entier. Ceci explique la présence de la clause `default`.

L'intérêt de l'utilisation des types énumérés est principalement **sémantique** : un programme qui les utilise est plus facile à lire et à maintenir.

Opérations sur les types énumérés

Contrairement aux types structurés, il est possible de comparer les éléments d'un type énuméré au moyen des **opérateurs relationnels**.

Ceci est une conséquence du fait que les énumérateurs sont des entiers.

```
printf("%d\n", SOL == SOL);    affiche 1  
printf("%d\n", SOL == FA);    affiche 0  
printf("%d\n", SI <= RE);     affiche 1
```

Opérations sur les types énumérés

Contrairement aux types structurés, il est possible de comparer les éléments d'un type énuméré au moyen des **opérateurs relationnels**.

Ceci est une conséquence du fait que les énumérateurs sont des entiers.

```
printf("%d\n", SOL == SOL);    affiche 1  
printf("%d\n", SOL == FA);    affiche 0  
printf("%d\n", SI <= RE);    affiche 1
```

De même, l'**opérateur d'affectation** = est compatible avec les types énumérés.

Opérations sur les types énumérés

Contrairement aux types structurés, il est possible de comparer les éléments d'un type énuméré au moyen des **opérateurs relationnels**.

Ceci est une conséquence du fait que les énumérateurs sont des entiers.

```
printf("%d\n", SOL == SOL);    affiche 1  
printf("%d\n", SOL == FA);    affiche 0  
printf("%d\n", SI <= RE);     affiche 1
```

De même, l'**opérateur d'affectation** = est compatible avec les types énumérés.

L'**opérateur de taille** `sizeof` renvoie 4 sur les valeurs d'un type énuméré. C'est la taille occupée par le type `int`.

Types structurés

- Déclaration et initialisation

- Affectation et comparaison

- Dans les fonctions

- Alignement en mémoire

Types structurés

Déclaration et initialisation

Affectation et comparaison

Dans les fonctions

Alignement en mémoire

Déclaration de types structurés récursifs

Il est possible de **déclarer des types structurés récursifs** en faisant usage de l'**alias** et du mot clé **struct** :

```
1 typedef struct _Liste {
2     int e;
3     struct _Liste *s;
4 } Liste;
```

Ceci fonctionne car la taille d'un pointeur vers une valeur de type **T** est connue et indépendante de la nature de **T**.

Attention, la déclaration

```
1 typedef struct _Liste {
2     int e;
3     struct _Liste s;
4 } Liste;
```

n'est pas valide car le **champ récursif** n'est pas un **pointeur**.

Le système ne peut pas connaître pas la taille de ce champ.

Déclaration de types structurés mutuellement récursifs

Il est possible de déclarer des types structurés mutuellement récursifs :

```
1 typedef struct _Flip {
2     int a;
3     int b;
4     struct _Flop *suiv;
5 } Flip;
6
7 typedef struct _Flop {
8     double x;
9     struct _Flip *suiv;
10 } Flop;
```

Dessiner un exemple de variable de type Flip.

Initialisation d'une variable d'un type structuré

Il est possible d'**initialiser les champs** d'une variable d'un type structuré au moment de sa **déclaration**.

On utilise pour cela l'opérateur d'affectation = avec comme valeur droite les valeurs des champs à affecter dans des accolades et séparées par des virgules.

Par exemple,

```
1 typedef struct {
2     char c;
3     int a;
4     double b;
5 } Triplet;
6 ...
7 Triplet tr = {'h', 55, 214.35};
```

Déclare, en l'initialisant, la variable `tr`.

Dessiner le contenu de `tr`.

Types structurés

Déclaration et initialisation

Affectation et comparaison

Dans les fonctions

Alignement en mémoire

Affectation de variables d'un type structuré

Considérons le code

```
1 typedef struct {
2     int a;
3     float f;
4 } X;
5 ...
6 X v1, v2;
7 v1.a = 10;
8 v1.f = 3.14;
9 v2 = v1;
10 v2.a = 20;
```

Dessiner le contenu de v1 et v2 aux l. 6, 8, 9 et 10.

Observation : l'affectation recopie les champs d'une variable d'un type scalaire.

Affectation de variables d'un type structuré

Considérons le code

```
1  typedef struct {
2      X x;
3      char t[3];
4  } Y;
5  ...
6  Y v1, v2;

7  v1.x.a = 10;
8  v1.x.f = 3.14;
9  v1.t = {'a', 'b', 'c'};
10 v2 = v1;
11 v2.x.f = 1.8;
12 v2.t[0] = 'g';
```

Dessiner le contenu de v1 et v2 aux l. 9, 10 et 12.

Observation : l'affectation recopie les champs d'une variable d'un type structuré de manière **récurive** et les tableaux statiques.

Affectation de variables d'un type structuré

Considérons le code

```
1  typedef struct {
2      char *t;
3      int n;
4  } T;
5  ...
6  T v1, v2;
7  v1.t = (char *)
8      malloc(sizeof(char) * 3);
9  v1.n = 3;
10 v1.t[0] = 'a';
11 v1.t[1] = 'b';
12 v1.t[2] = 'c';
13 v2 = v1;
14 v2.n = 2;
15 v2.t[0] = 'g';
```

Dessiner le contenu de v1 et v2 aux l. 12, 13 et 15.

Observation : l'affectation ne recopie pas les tableaux dynamiques. Seule l'adresse d'un tableau dynamique est recopiée. C'est une **copie de surface**.

Affectation de variables d'un type structuré

Règle générale : pour chaque déclaration d'un type structuré **X**, on définit (dans le même module) une fonction de prototype

```
void copier_X(const X *v1, X *v2);
```

qui **copie en profondeur** les champs de **v1** dans les champs de **v2**.

Par exemple, la définition du type **T** précédent s'accompagne de la définition de la fonction

```
1 void copier_T(const T *v1, T *v2) {
2     int i;
3     assert(v1 != NULL);
4     assert(v2 != NULL);
5     v2->n = v1->n;
6     v2->t = (char *) malloc(sizeof(char) * v1->n);
7     if (v2->t == NULL) exit(EXIT_FAILURE);
8     for (i = 0 ; i < v1->n ; ++i)
9         v2->t[i] = v1->t[i];
10 }
```

Il est possible de munir cette fonction du mécanisme habituel de gestion d'erreurs.

Comparaison de variables d'un type structuré

Considérons le code

```
1  typedef struct {
2      int a;
3      int b;
4  } A;
5  ...
6  A v1, v2;
7  ...
8  if (v1 == v2) {...}
9  ...
10 if (v1 != v2) {...}
```

Ce code est incorrect (il ne compile pas).

Le compilateur n'accepte pas la comparaison de variables d'un type structuré.

```
invalid operands to binary == (have 'A' and 'A')
```

```
invalid operands to binary != (have 'A' and 'A')
```

Comparaison de variables d'un type structuré

Règle générale : pour chaque déclaration d'un type structuré X , on définit (dans le même module) deux fonctions de prototypes

```
int sont_ega_X(const X *v1, const X *v2);  
int sont_dif_X(const X *v1, const X *v2);
```

qui **testent l'égalité** et **l'inégalité** entre $v1$ et $v2$.

Par exemple, la définition du type A précédent s'accompagne de la définition des fonctions

```
1 int sont_ega_A(A *v1, A *v2) {           7 int sont_dif_A(A *v1, A *v2) {  
2     assert(v1 != NULL);                 8     assert(v1 != NULL);  
3     assert(v2 != NULL);                 9     assert(v2 != NULL);  
4     return (v1->a == v2->a)            10    return !sont_ega_A(v1, v2);  
5         && (v1->b == v2->b);           11 }  
6 }
```

Attention : si X est composé d'un champ qui est un type structuré Y , il faut appeler dans `sont_ega_X` la fonction de comparaison `sont_ega_Y`.

Destruction de variables d'un type structuré

Règle générale : pour chaque déclaration d'un type structuré **X**, on définit (dans le même module) une fonction de prototype

```
void detruire_X(X *v);
```

qui libère l'espace mémoire adressé par **v**.

Par exemple, la déclaration du type **B** suivant s'accompagne de la définition de la fonction

```
1 typedef struct {
2     int *tab;
3     int n;
4 } B;
5
6 void detruire_B(B *v) {
7     assert(v != NULL);
8     free(v->tab);
9     *v = NULL;
10 }
```

Attention : si **X** est composé d'un champ qui est un type structuré **Y**, il faut appeler dans `detruire_X` la fonction de comparaison `detruire_Y`.