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;
}
```

Plan

Plan

Types

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.

Plan

Types entier

On se place sur une machine 64 bits.

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

Types entier

On se place sur une machine 64 bits.

Nom	Taille (octets)	Plage
`char`	1	-128 à 127
`short`	2	-32768 à 32767
`int`	4	-2^{31} à $2^{31}-1$
`long`	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
`unsigned char`	0 à 255
`unsigned short`	0 à 65535
`unsigned int`	0 à $2^{32}-1$
`unsigned long`	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
`float`	4	3.40282×10^{38}
`double`	8	1.79769×10^{308}
`long double`	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 $k \geqslant 0$ 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 $k \geqslant 0$ 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 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`.

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
} Booleen;
```

est un type qui permet de représenter des booléens.

Une valeur de type `Booleen` 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 1er 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 1er 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`.

Plan

Plan

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`.

Plan

Renvoi d'une variable d'un type structuré

Le code

```
1  typedef struct {
2      int x;
3      int y;
4  } Couple;
5
6
```

```
7  Couple twist(Couple c) {
8      Couple res;
9      res.x = c.y;
10     res.y = c.x;
11     return res;
12 }
```

est correct (`twist` renvoie le couple obtenu par échange des coordonnées de celui passé en argument).

`twist` renvoie une variable d'un type structuré.

Cependant, il n'est pas efficace car, à chaque appel de fonction

```
d = twist(c);
```

la variable `res`, qui vit dans la pile, doit être recopiée.

Paramètre variable d'un type structuré

Le code

```
1  typedef struct {
2      int tab1[2048];
3      int tab2[2048];
4  } DeuxTab;
5
```

```
6  int prem_egaux(DeuxTab x) {
7      return x.tab1[0]
8          == x.tab2[0];
9  }
```

est correct (`prem_egaux` teste si les premières cases des tableaux sont égales).

`prem_egaux` est paramétrée par une variable d'un type structuré.

Cependant, il n'est pas efficace car à chaque appel de fonction

```
prem_egaux(y);
```

les champs de l'**argument** `y` sont recopiés dans le **paramètre** `x`.

Passage par adresse *vs* passage par valeur

Soit une fonction `fct` paramétrée par une variable `x` d'un type structuré `T`.

Il est d'usage courant d'adopter la convention suivante :

- si les champs de `x` doivent être modifiés par la fonction, alors on recourt à un **passage par adresse**

```
... fct(T *x, ...) { ... }
```

- si les champs de `x` ne doivent pas être modifiés par la fonction, alors on recourt à un **passage par valeur**

```
... fct(T x, ...) { ... }
```

Cette conception est erronée car il est possible de « modifier » une variable d'un type structuré passée par valeur à une fonction.

Passage par adresse *vs* passage par valeur

Considérons en effet le code suivant :

```
1  tyepdef struct {
2      int *tab;
3      int n;
4  } Tab;
5
```

```
6  void init(Tab t, int k) {
7      int i;
8      for (i = 0 ; i < t.n ; ++i)
9          t.tab[i] = k;
10 }
```

Chaque appel de fonction

```
init(s, r);
```

provoque la recopie de trois valeurs (ce qui est encore acceptable) mais « modifie » les valeurs pointées par le champ `tab` de `s`, malgré le passage par valeur.

Conclusion : écrire des fonctions avec passage par valeur des paramètres d'un type structuré ne présente que des désavantages.

Variables d'un type structuré dans les fonctions

En résumé, on adopte les deux règles suivantes :

1. une fonction ne renvoie jamais de valeur d'un type structuré ;
2. tous les paramètres d'un type structuré sont passés par adresse dans une fonction.

Ainsi, un prototype de fonction habituel est

```
int fct(T *x, E1 e1, ..., EN en, S1 *s1, ..., SM *sm);
```

où

- le type de retour est `int` (renvoi d'un code d'erreur) ;
- `x` est l'adresse d'une variable d'un type structuré `T` ;
- `e1`, ..., `en` sont les entrées de la fonction (adresses ou non) ;
- `s1`, ..., `sm` sont les sorties de la fonction (qui sont des adresses).

Variables d'un type structuré dans les fonctions

P.ex., voici le nécessaire pour calculer la somme pondérée de deux points selon les conventions établies :

```
1  typdef struct {
2      float x;
3      float y;
4  } Point;
5
6  void somme_points(const Point *p1, const Point *p2,
7          float coeff1, float coeff2,
8          Point *res) {
9
10     assert(p1 != NULL);
11     assert(p2 != NULL);
12     assert(res != NULL);
13
14     res->x = coeff1 * p1->x + coeff2 * p2->x;
15     res->y = coeff1 * p1->y + coeff2 * p2->y;
16 }
```

Plan

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 récursive et les tableaux statiques.