Programmation en C

Perfectionnement à la programmation en C

Henri Derycke

henri.derycke@univ-eiffel.fr
https://igm.univ-mlv.fr/~derycke/PC

Université Gustave Eiffel
IGM, bureau 4B165

à partir du cours de Samuele Giraudo

2024–2025

Introduction

L'objectif de ce module est d'approfondir les concepts de base et d'approcher certains concepts plus avancés de la programmation en C.

Celui-ci est organisé en trois axes.

1. **Axe 1** : **écrire un projet maintenable**.

   Bonnes habitudes de programmation, documentation, pré-assertions, gestion des erreurs, programmation modulaire, compilation.

2. **Axe 2** : **comprendre les mécanismes de base**.

   Pointeurs, allocation dynamique, types, types structurés, entrées/sorties.

3. **Axe 3** : **utiliser quelques techniques avancées**.

   Opérateurs bit à bit, mémoïsation, génération aléatoire, pointeurs de fonction, généricité.

Contenu du cours

Axe 1.

1. Bases
2. Habitudes
3. Modules
4. Compilation

Axe 2.

5. Allocation dynamique
6. Entrées et sorties
7. Types
8. Types structurés

Axe 3.

9. Opérateurs
10. Pointeurs de fonction
11. Mémoïsation
12. Génération aléatoire

Pré-requis

Ce cours demande les pré-requis suivants :

- des bases en **programmation générale** (notion de programme, d'instruction, de compilation) ;
- des bases en **programmation impérative** (notion de variable, de structures de contrôle, de fonction) ;
- des bases en **algorithmique** (manipulation de structures de données élémentaires comme les tableaux, les chaînes de caractères, les listes) ;
- des bases en **programmation en C** (écriture de fonctions, manipulation de tableaux, de chaînes de caractères, connaissance des types numériques, des types structurés, notion d'allocation dynamique, de pointeur).

Quelques dates

Le langage C est apparu en 1972 dans les laboratoires Bell. Il a été créé par **D. Ritchie** et **K. Thompson** au même moment que l'apparition des systèmes UNIX.

Il est influencé par le langage B qui, contrairement au C, ne possédait pas de système de type.

Quelques dates sur l'évolution de la spécification du langage :

- 1978 : première description complète du langage (traditionnel) ;
- 1989/1990 : **publication de la norme ANSI C** (ou C89) et C90 ;
- 1999 : évolution du langage C99 ;
- 2011 : nouvelle version du standard C11 ;
- 2018 : nouvelle version du standard C17 (ou C18) ;
- 2023 : nouvelle version du standard C23.

Quelques caractéristiques

Le C est un **langage de bas niveau** : il offre la possibilité de se placer « proche » de la machine. Il permet en effet de manipuler finement des données en mémoire, des adresses et des suites de bits.

Ce langage a été initialement pensé pour la **conception de systèmes d'exploitation**.

Il est cependant suffisamment expressif pour s'adapter à un large éventail d'utilisations.

Il se situe à la croisée des chemins du monde des langages de programmation : beaucoup de langages modernes sont traduits en C pour être compilés.

Axe 1 : écrire un projet maintenable

1. Bases
2. Habitudes
3. Modules
4. Compilation

1. Bases

Plan

1. Bases

La notion de programme

Un programme en C est avant tout un texte (contenu dans un **fichier**) qui suit certaines règles (dites de **syntaxe**).

C'est une collection

- de déclarations de fonctions ;
- de définitions de fonctions ;
- de déclarations de types ;
- de déclarations de variables globales ;
- de commandes pré-processeur.

Tout programme possède une **fonction principale** nommée `main`. C'est par elle que commence l'exécution du programme. On appelle ceci le point d'entrée du programme.

Compiler un programme

Compiler un programme signifie **traduire** le fichier le contenant en un langage compréhensible et exécutable par la machine cible.

On compile un fichier `Prog.c` par la commande

```
gcc -o NOM Prog.c
```

Ceci produit un exécutable nommé `NOM`.

On compilera obligatoirement avec les **options** `-std=c17`, `-pedantic` et `-Wall` au moyen de la commande

```
gcc -o NOM -std=c17 -pedantic -Wall Prog.c
```

- `-std=c17 -pedantic` : empêche un programme non compatible avec la norme C17 de compiler ;
- `-Wall` : active tous les messages d'avertissement.

D'autres options seront utiles par la suite notamment à des fins de débogage.

Plan

1. Bases

Expressions

Une expression est définie récursivement comme étant soit

1. une constante ;
2. une variable ;
3. une combinaison d'expressions et d'opérateurs ;
4. un appel de fonction qui renvoie une valeur, c.-à-d., de type de retour autre que `void`.

Une expression n'est donc rien d'autre qu'un **assemblage de symboles** qui vérifie des règles **syntaxiques** et **sémantiques**.

Toute expression possède une **valeur** et un **type**. Le processus qui consiste à déterminer la valeur d'une expression se nomme l'évaluation.

Expressions

– Exemples –

Les entités suivantes sont des expressions :

- `0`

  Valeur : `0`, type : **`int`**

- `'a'`

  Valeur : `97`, type : **`int`**

- `"abc"`

  Valeur : `"abc"`, type : **`char *`**

- `x`

  Valeur : `x`, type : le type de la variable `x`

- `a == 2`

  Valeur : `0` ou `1`, type : **`int`**

- `b = 3`

  Valeur : `3`, type : **`int`**

- `f(5)`

  Valeur : la valeur renvoyée par `f(5)`, type : le type de retour de `f`

- `(a == 0) && (b >= 3)`

  Valeur : `0` ou `1`, type : **`int`**

- `a > 'a' ? a : -a`

  Valeur : `a` ou `-a`, type : le type de `a`

- `1. + 7 * 2`

  Valeur : `15.`, type : **`double`**

Instructions

Une instruction est définie récursivement comme étant soit :

1. une expression `E;` terminée par un point-virgule ;
2. un bloc `{I1 I2 ...Ik}` où `I1`, `I2`, …, `Ik` sont des instructions ;
3. une conditionnelle `if (E) I1 else I2`, où `E` est une expression et `I1` et `I2` sont des instructions ;
4. toute autre construction similaire (**`switch`**, **`while`**, **`do while`**, **`for`**) ;
5. une déclaration `T X;` de variable, où `T` est un type et `X` est un identificateur.

À la différence des expressions, c'est souvent l'**effet** d'une instruction qui est sa raison d'être (et non plus uniquement sa valeur ou son type).

En effet, une instruction peut p.ex. afficher un élément, allouer une zone mémoire, lire dans un fichier, *etc.* La valeur de l'expression sous-jacente à l'instruction est d'importance moindre dans certains cas.

Instructions

– Exemples –

Les entités suivantes sont des instructions :

- `;` (instruction vide)
- `1;`
- `a = b;`
- `while (1) a += 1;`
- `printf("abc\n");`
- `{ a++; b--; }`
- `malloc(64);`
- `int a;`
- `int tab[64];`
- `tab[3] = 9;`
- `tab[3];`
- `return 31;`

Expressions à effet secondaire

Une **expression** est à effet secondaire (ou encore, improprement à « effet de bord ») si son évaluation modifie la mémoire.

– Exemples –

- `0`,
- `'c'`,
- `(a + 21) << 3`,
- `tab[8]`,

sont des expressions qui ne sont pas à effet secondaire.

– Exemples –

En revanche,

- `a = 31`,
- `printf("abc\n")`,
- `char c`,
- `malloc(64)`,

sont des expressions à effet secondaire (noter que la déclaration de variable n'est pas une expression mais une instruction).

Expressions à effet secondaire

En règle générale, ce sont les éléments suivants dans les expressions qui produisent des effets secondaires :

- les affectations ;
- les allocations de mémoire ;
- la sollicitation au système de fichiers.

En revanche, les éléments suivants dans les expressions ne produisent pas d'effet secondaire :

- lectures de constantes et de variables ;
- calculs arithmétiques, logiques ou bit à bit ;
- comparaisons.

Plan

1. Bases

Blocs

Un bloc est une suite d'instructions délimitée par des accolades. Il est constitué d'une partie consacrée à la déclaration de variables et d'une partie consacrée aux instructions.

```
{
    D
    I
}
```

Ici, `D` est une section de **déclarations** et `I` est une section d'**instructions**.

Les parties `D` et/ou `I` peuvent être vides.

Sachant qu'un bloc est une suite d'instructions, il est possible de placer un bloc dans la section d'instructions d'un bloc et ainsi d'**imbriquer** plusieurs blocs.

Blocs

– Exemple –

```c
int main() {
    int b;
    scanf("%d", &b);
    {
        int a;
        a = 1 + b;
    }
}
```

La partie coloriée est un bloc au sein d'une fonction `main`.

Il y a des règles concernant la visibilité des variables (que nous verrons plus loin).

– Exemple –

```c
{
    int a;
    int b;
    scanf(" %d", &a);
    {
        printf("%d\n", a);
    }
    scanf(" %d", &b);
}
```

Ceci est un bloc constitué d'une section de déclarations et d'une section d'instructions.

Cette dernière partie contient un bloc (en couleur) dont la section de déclarations est vide.

Opérateur de test `if`

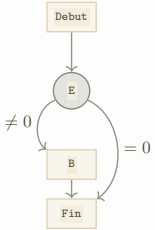
L'opérateur de test se décline en deux versions :

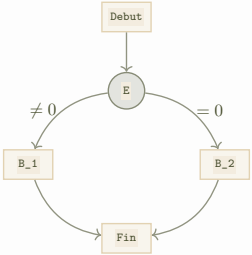
```
if (E)
    B
```

```
if (E)
    B_1
else
    B_2
```

Ici, `E` est une expression booléenne : elle est considérée comme **fausse** si elle s'évalue en zéro et comme **vraie** sinon. De plus, `B`, `B_1` et `B_2` sont des blocs d'instructions.

Diagrammes d'exécution :





Instruction de branchement `switch`

L'instruction de branchement admet la syntaxe

```
switch (E) {
    case E_1 : I_1
    case E_2 : I_2
    ...
    case E_N : I_N
    default : I_D
}
```

Ici, `E`, `E_1`, `E_2`, …, `E_N` sont des expressions **constantes** qui s'évaluent en des **entiers**.

Les `I_1`, `I_2`, …, `I_N` et `I_D` sont des suites d'instructions.

L'exécution de cette instruction se passe ainsi. Soit `i` le plus petit entier (s'il existe) tel que les évaluations de `E` et `E_i` sont égales. Les instructions `I_i`, …, `I_N` ainsi que `I_D` sont exécutées.

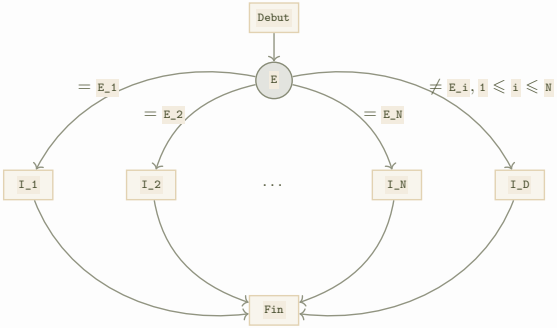
La ligne `default : I_D` est facultative. Si elle est présente, `I_D` est toujours exécutée.

Instruction de branchement `switch`

On s'impose de terminer chaque `I_j` et `I_D` par le mot-clé **`break`**.

Ceci permet de n'exécuter que le `I_i` tel que les évaluations de `E` et `E_i` sont égales. Dans ce cas, `I_D` n'est exécuté que si aucun des `E_i` ne l'a été.

On obtient ainsi le diagramme d'exécution



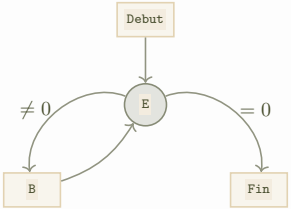
Instruction itérative `while`

L'instruction itérative **`while`** admet la syntaxe

```
while (E)
    B
```

Ici, `E` est une expression booléenne et `B` est un bloc d'instructions.

Diagramme d'exécution :



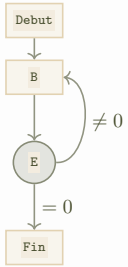
Instruction itérative `do while`

L'instruction itérative **`do while`** admet la syntaxe

```
do
    B
while (E);
```

Ici, `E` est une expression booléenne et `B` est un bloc d'instructions.

Diagramme d'exécution :



Instruction itérative `for`

L'instruction itérative `for` admet la syntaxe

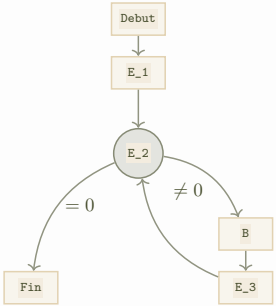
```
for (E_1 ; E_2 ; E_3)
    B
```

ou

```
for (D_1 ; E_2 ; E_3)
    B
```

Ici, `E_2` est une expression booléenne (le test), `E_1` (l'initialisation) et `E_3` (l'incrémentation) sont des expressions, `D_1` est une déclaration et `B` est un bloc d'instructions.

Diagramme d'exécution :



Instruction itérative `for`

– Exemple –

```
for (Cell *x = lst ; x != NULL; x = x->suiv) {
    afficher(x->elem);
}
```

Parcourt une liste simplement chaînée et l'affiche. La variable `x` est un pointeur sur la cellule courante.

– Exemple –

```
for (i = 0 ; i < 100 && test(i) ; ++i) {
    /* utilisation de i... */
}
if (i < 100)
    printf("%d", i);
```

Affiche la première valeur inférieure à 100 ne passant pas un test.

– Exemple –

```
for ( ; *str != '\0'; str++)) {
    putchar(*str);
}
```

Affiche la chaîne de caractères `str`. Il n'y a pas d'initialisation. L'incrémentation fait pointer `str` sur le prochain caractère.

– Exemple –

```
for ( ; *str != '\0'; putchar(*str++)) {
    /* Rien. */
}
```

Affiche la chaîne de caractères `str`. Il n'y a pas d'initialisation. L'affichage est réalisé comme effet secondaire de l'incrémentation.

Instructions de court-circuit

Les instructions de court-circuit sont `break` et `continue`.

L'instruction `break` permet à l'exécution de sortir d'une structure `switch`, `while`, `do while` ou `for`. L'exécution continue alors aux instructions qui suivent cette structure.

L'instruction `continue` permet à l'exécution de sauter à la fin du bloc `B` d'une structure `while`, `do while` ou `for`. L'exécution continue alors à l'évaluation de l'expression test.

Instructions de court-circuit

Dans une boucle `for`, l'instruction `continue` fait que l'expression d'incrémentation est tout de même évaluée.

– Exemple –

```
for (int i = 1 ; i <= 7 ; ++i) {
    if (i == 4)
        continue;
    printf("%d ", i);
}
```

L'exécution de ces instructions produit sept tours de boucle et l'affichage `1 2 3 5 6 7`. Lorsque `i == 4`, le `continue` relance l'exécution à l'incrémentation du `for`.

Ce n'est pas le cas pour le `break`.

– Exemple –

```
for (int i = 1 ; i <= 7 ; ++i) {
    if (i == 4)
        break;
    printf("%d ", i);
}
```

L'exécution de ces instructions produit quatre tours de boucle et l'affichage `1 2 3`. Lorsque `i == 4`, le `break` fait sortir de la boucle et `i` vaut `4`.

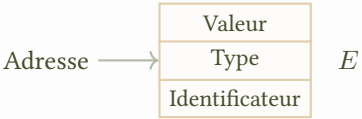
Plan

1. Bases

Variables

Une variable est une entité constituée des cinq éléments suivants :

1. un identificateur ;
2. un type ;
3. une valeur ;
4. une adresse ;
5. une portée lexicale.



Intuitivement, c'est une boîte (un objet) qui peut contenir une valeur et qui dispose d'un nom (identificateur).

Une boîte ne peut contenir que des valeurs d'une certaine sorte (type).

Elle se situe de plus à un endroit bien précis dans la mémoire (adresse) et elle n'est visible par son nom qu'à partir de certains endroits du code (portée lexicale).

Identificateurs de variable

L'identificateur d'une variable est un mot commençant par une lettre ou bien `'_'`, suivi par un nombre arbitraire de lettres, chiffres ou `'_'`.

De plus, aucun identificateur ne peut-être un mot réservé du langage. En voici la liste complète :

auto	**break**	**case**	**char**	**const**	**continue**	**default**	**do**
double	**else**	**enum**	**extern**	**float**	**for**	**goto**	**if**
int	**long**	**register**	**return**	**short**	**signed**	**sizeof**	**static**
struct	**switch**	**typedef**	**union**	**unsigned**	**void**	**volatile**	**while**

L'identificateur d'une variable est **attribué à sa déclaration**.

Objet et valeur d'une variable

L'objet d'une variable est la raison pour laquelle celle-ci existe. Son rôle premier est en effet de contenir la valeur de la variable.

La valeur d'une variable n'est **pas attribuée à sa déclaration** (elle contient à ce moment là une valeur mais il ne faut rien supposer dessus).

On accède à la valeur d'une variable par son identificateur.

On modifie le contenu d'une variable par une **affectation**. L'occurrence de l'identificateur de la variable se trouve dans ce cas à gauche de l'opérateur `=`.

– Exemple –

```
int num;

num = 23;
num = num + 32;
```

L'occurrence de `num` en l. 2 est située à gauche du `=` : il s'agit d'une affectation. Il y en a deux en ligne 3 : la 1re permet de modifier et la 2 de lire sa valeur.

Adresse d'une variable

L'adresse d'une variable est une valeur entière spécifiant la position de la variable en mémoire.

L'adresse d'une variable est **attribuée à sa déclaration** par le système à l'**exécution**. Elle ne peut pas être choisie par le programmeur ni être modifiée.

On accède à l'adresse d'une variable par son identificateur précédé de l'opérateur `&`.

– Exemple –

```
int num;

printf("%p\n", &num);
```

Une 1re exécution de ces instructions affiche `0x7fff6a3014fc`. Une 2e affiche `0x7fffbdc357dc`. L'adresse de `num` varie d'une exécution à l'autre.

Portée lexicale d'une variable et variables locales

La portée lexicale d'une variable désigne la **zone du programme** dans laquelle la variable peut être utilisée par son nom.

Elle dépend de l'endroit dans lequel elle a été déclarée.

Sa portée lexicale s'étend aux instructions qui sont situées après sa déclaration dans le plus petit bloc d'instructions qui la contient.

– Exemple –

```
void f(int x) {
    int a, b;
    ...
}



int g(int y, int z) {
    int c;
    ...
}
```

La portée lexicale des variables `a` et `b` s'étend aux instructions du corps de la fonction `f`. Elle ne s'étend pas aux instructions du corps de `g`. Les variables `a` et `b` sont des variables locales à la fonction `f` et invisibles ailleurs.

Le **paramètre** `x` de `f` a pour portée lexicale uniquement le corps de `f`.

Portée lexicale d'une variable et variables locales

– Exemple –

```
...
int f() {...}
...
int taille = 31;
...
int g() {...}
...
int main() {...}
```

La portée lexicale de la variable `taille` s'étend à tout ce qui suit sa déclaration dans le programme. Elle est donc visible dans les fonctions `g` et `main` mais pas dans `f`. Étant donné qu'elle est déclarée en dehors de toute fonction, elle est qualifiée de variable globale.

Attention : l'utilisation de variables globales n'est ni élégante ni indispensable. Elle est également source d'erreurs car il est souvent difficile de comprendre un programme les utilisant.

Elle est bannie pour ces raisons.

On préfère utiliser des définitions préprocesseur pour représenter leur valeur.

Portée lexicale d'une variable et blocs d'instructions

– Exemple –

```
{
    int a;
    a = 15;
    printf("%d", a);
}
printf("%d", a);
```

Il y a erreur de compilation : la portée lexicale de la variable `a` s'étend de la l. 2 à la l. 4. Elle n'est pas visible à la l. 6. Il n'existe pas de variable identifiée par `a` lorsque la l. 7 est évaluée.

– Exemple –

```
{
    int a;
    a = 15;
    printf("%d", a);
}
{
    printf("%d", a);
}
```

De la même manière que dans l'exemple précédent, l'occurrence du symbole `a` dans le second bloc (l. 7) n'est pas résolue. Elle se situe dans un bloc qui n'est pas contenu par celui où le symbole `a` est déclaré.

Portée lexicale d'une variable et blocs d'instructions

– Exemple –

```
int a;
a = 10;
{
    printf("%d ", a);
}
printf("%d\n", a);
```

Ces instructions produisent l'affichage `10 10`. En effet, la variable `a` est visible dans le bloc d'instructions dans lequel elle est définie, ainsi que dans les blocs d'instructions qui se trouvent à l'intérieur.

– Exemple –

```
int a;
a = 10;
{
    int a;
    a = 20;
    printf("%d ", a);
}
printf("%d\n", a);
```

Ces instructions produisent l'affichage `20 10`. La variable identifiée par `a` dans le bloc d'instructions de la l. 3 à la l. 7 est celle déclarée en l. 4. La variable identifiée par `a` hors de ce bloc d'instructions est celle déclarée en l. 1.

Portée lexicale d'une variable et blocs d'instructions

– Exemples –

Comparons les instructions

```
int a;
a = 10;
for (int a = 0; a < 4; a++) {
    printf("%d ", a);
}
printf("%d\n", a);
```

```
int a;
a = 10;
for (a = 0; a < 4; a++) {
    printf("%d ", a);
}
printf("%d\n", a);
```

Dans le cas de gauche, l'affectation `a = 0` n'a d'effet que sur la variable `a` déclarée à l'occasion de la boucle `for`. Ceci affiche `0 1 2 3 10`.

En revanche, les instructions de droite affichent `0 1 2 3 4` car il n'y a pas de déclaration de `a` dans le bloc. L'affectation et les incrémentations modifient la variable `a` déclarée en l. 1.

Plan

1. Bases

Fonctions

Une fonction est constituée

1. d'un identificateur (qui suit les mêmes contraintes que ceux des variables) ;
2. d'une signature (la liste de ses paramètres et de leurs types) ;
3. d'un type de retour (le type de la valeur renvoyée par la fonction) ;
4. d'instructions (qui forment le corps de la fonction).

La ligne constituée du type de retour, de l'identificateur et de la signature d'une fonction est son prototype.

– Exemple –

```
int produit(int a, int b, int c) {
    return a * b * c;
}
```

est une fonction d'identificateur `produit`, de signature `(int a, int b, int c)` et de type de retour `int`.

Définition *vs* déclaration

La définition d'une fonction consiste à fournir tous ses constituants.

La déclaration d'une fonction consiste à fournir son **prototype**.

Déclarer une fonction est utile si l'on souhaite s'en servir avant de l'avoir définie.

– Exemple –

```c
#include <stdio.h>

/* Declarations. */
void flop(int nb);
void flip(int nb);

/* Definitions. */
int main() {
    flip(10);
    return 0;
}
```

```c
void flip(int nb) {
    if (nb >= 1) {
        printf("flip\n");
        flop(nb - 1);
    }
}
```

```c
void flop(int nb) {
    if (nb >= 1) {
        printf("flop\n");
        flip(nb - 1);
    }
}
```

Paramètres *vs* arguments

Il faut faire attention à bien distinguer les notions de paramètre et d'argument qui sont deux choses différentes.

On considère la fonction

```
int produit(int a, int b, int c) {
    return a * b * c;
}
```

Les symboles `a`, `b` et `c` de son prototype sont ses paramètres.

Lors de l'appel

```
produit(15, num, -3);
```

les expressions `15`, `num` et `-3` sont les arguments de l'appel.

Aide-mémoire : **p**aramètre ↔ **p**rototype ; **a**rgument ↔ **a**ppel.

Portée lexicale des paramètres

Les variables locales d'une fonction ont pour portée lexicale la fonction elle-même (déjà mentionné).

Il en est de même pour ses **paramètres** : leur portée lexicale est la fonction elle-même. On peut voir la déclaration des paramètres d'une fonction dans son en-tête comme une déclaration de variable.

– Exemple –

```
int doubl(int a) {
    return 2 * a;
}

int main() {
    int a;
    a = 10;
    a = doubl(a + 1);
    return 0;
}
```

Il y plusieurs occurrences du symbole `a`.

Celui déclaré dans l'en-tête de `doubl` a une portée lexicale qui s'étend de la l. 1 à la l. 2.

Celui déclaré dans le `main` a pour portée lexicale le `main` tout entier.

Ce sont des variables différentes.

Pile

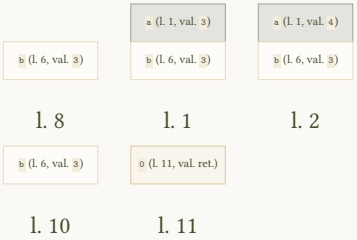
Lors de l'appel d'une fonction, les valeurs de ses arguments sont **recopiées** dans une zone de la mémoire appelée pile.

Conséquence très importante : toute modification des paramètres dans une fonction ne modifie pas les valeurs des arguments avec lesquels elle a été appelée.

– Exemple –

```
void incr(int a) {
    a = a + 1;
}

int f() {
    int b;

    b = 3;
    incr(b);
    printf("%d\n", b);
    return 0;
}
...
f();
```

L'appel à `f` en l. 14 produit les configurations de pile



Pile

Les **variables locales** d'une fonction (c.-à-d. les variables déclarées dans le corps de la fonction) se situent dans la **pile**.

Après avoir appelé une fonction, c.-à-d. juste après avoir renvoyé la valeur de retour, la pile se trouve dans le même état qu'avant l'appel.

Conséquence très importante : toute variable locale à une fonction est non seulement invisible mais n'existe plus en mémoire hors de la fonction et après son appel.

Pile

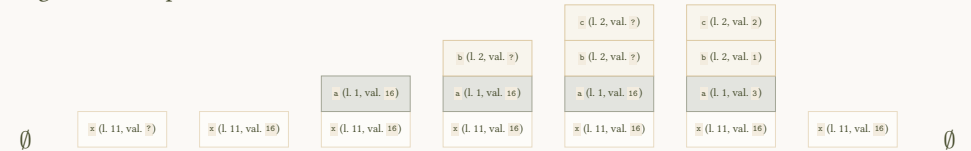
– Exemple –

```
1  void fct_1(int a) {
2      int b, c;
3      b = 1;
4      c = 2;
5      a = b + c;
6  }
```

```
10 void fct_2() {
11     int x;
12     x = 16;
13     fct_1(x);
14     printf("%d\n", x);
15 }
```

```
20 ...
21 fct_2();
22 ...
```

Configurations de pile :



Pile et fonctions récursives

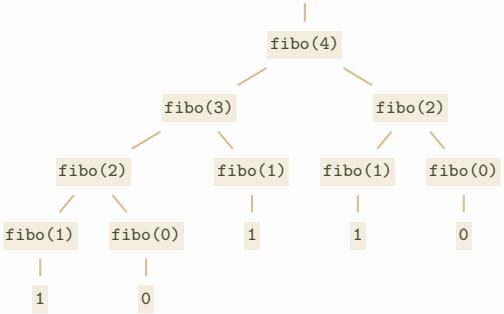
Soit la fonction

```
int fibo(int a) {
    if (a <= 1)
        return a;
    return fibo(a - 1) + fibo(a - 2);
}
```

et la suite d'instructions

```
int x;
x = fibo(4);
```

On représente son exécution par un **arbre des appels** :



Fonctions à effet secondaire

Une **fonction** est à effet secondaire s'il existe au moins un jeu d'arguments qui fait que l'évaluation de l'appel à la fonction sur ce jeu d'arguments modifie la mémoire par rapport à son état d'avant l'appel.

– Exemple –

```
int f(int a, int b) {
    return 21 * a + b;
}
```

Cette fonction n'est pas à effet secondaire. Elle renvoie une valeur sans modifier la mémoire.

– Exemple –

```
float double_val(float *x) {
    *x = 2 * (*x);
    return *x;
}
```

Cette fonction est à effet secondaire puisqu'elle modifie une zone de la mémoire (celle à l'adresse spécifiée par son argument).

Fonctions à effet secondaire

– Exemple –

```
int g(char c) {
    int b;
    b = 5;
    return b + c;
}
```

Cette fonction n'est pas à effet secondaire. La déclaration (et l'affectation) de `b` reste locale à la fonction. Après tout appel à `g`, la variable `b` n'existe plus.

– Exemple –

```
char *allouer(int n) {
    char *res;
    res = (char *) malloc(sizeof(char) * n);
    return res;
}
```

Cette fonction est à effet secondaire. Elle réserve en effet, par l'appel interne à la fonction `malloc`, une zone de la mémoire, ce qui modifie son état.

– Exemple –

```
int h(int a, int b) {
    if (a * b == 0)
        printf("z\n");
    return a - b;
}
```

Cette fonction est à effet de secondaire. En effet, l'appel à `h` avec, p.ex., les arguments `1` et `0` provoque un affichage sur la sortie standard, modifiant l'état de la mémoire.

Plan

1. Bases

Commandes préprocesseur

Une commande préprocesseur (ou directive préprocesseur) est une ligne qui commence par `#`.

Le préprocesseur est une unité qui intervient lors de la compilation. Son rôle est de traiter les commandes préprocesseur.

Il fonctionne en **construisant une nouvelle version** du programme en **remplaçant** chaque commande préprocesseur par des expressions en C adéquates.

Il existe plusieurs sortes de commandes préprocesseur :

- les inclusions de fichiers ;
- les définitions de symboles ;
- les macro-instructions à paramètres ;
- les macro-instructions de contrôle de compilation.

Inclusions de fichiers

La commande préprocesseur

```
#include <NOM.h>
```

permet d'**inclure** le fichier `NOM.h` dans le programme pour bénéficier des fonctionnalités qu'il apporte.

Le préprocesseur résout cette commande en recopiant le contenu de `NOM.h` à l'endroit où elle est invoquée.

Il est possible d'enchaîner les inclusions :

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

Habituellement, les inclusions sont réalisées au début du programme.

Définitions de symboles

La commande préprocesseur

```
#define SYMB EXP
```

permet de **définir un alias** `SYMB` pour l'expression `EXP`. Ceci autorise à faire référence à l'expression `EXP` par l'intermédiaire du symbole `SYMB`.

Le préprocesseur résout tout invocation `SYMB` en la remplaçant par `EXP`.

– Exemple –

À gauche (resp. à droite), des instructions avant (resp. après) le passage du préprocesseur :

```
#define NB 5
#define CHAINE "cba\n"
...
for (int i = 1 ; i <= NB ; ++i) {
    printf("%s", CHAINE);
}
```

```
/* Rien. */
/* Rien. */
...
for (int i = 1 ; i <= 5 ; ++i) {
    printf("%s", "cba\n");
}
```

Par convention, tout alias est constitué de lettres majuscules, de chiffres ou de tirets bas.

Macro-instructions à paramètres

La commande préprocesseur

```
#define SYMB(P1, P2, ..., Pn) EXP
```

permet de définir une **macro-instruction à paramètres** `SYMB`. Ceci autorise à faire référence à l'expression `EXP` par l'intermédiaire du symbole `SYMB` paramétrable par des paramètres `P1`, `P2`, …, `Pn`.

Le préprocesseur résout toute invocation `SYMB(A1, A2, ..., An)` en la remplaçant par l'expression obtenue en substituant `Ai` à toute occurrence du paramètre `Pi` dans `EXP`.

– Exemple –

À gauche (resp. à droite), des instructions avant (resp. après) le passage du préprocesseur :

```
#define MAX(a, b) a > b ? a : b
...
int x;
x = MAX(10, 14);
```

```
/* Rien. */
...
int x;
x = 10 > 14 ? 10 : 14;
```

Macro-instructions à paramètres

Problème : étudions comment le préprocesseur transforme les instructions suivantes :

```
#define CARRE(a) a * a
...
x = 2;
y = 3;
z = CARRE(x + y);
```

La l. 5 est remplacée par `z = x + y * x + y;`.

Ainsi, la valeur `2 + 3 * 2 + 3 = 11` est affectée à `z` au lieu de `25` comme attendu.

Solution : il faut placer des parenthèses autour des paramètres des macro-instructions à paramètres :

```
#define CARRE(a) (a) * (a)
```

De cette façon, `CARRE(x + y)` est remplacée par `(x + y) * (x + y)` comme désiré.

Macro-instructions à paramètres

Problème : étudions comment le préprocesseur transforme les instructions suivantes :

```
#define DOUBLE(a) (a) + (a)
...
x = 3;
z = 5 * DOUBLE(x);
```

La l. 4 est remplacée par `z = 5 * (x) + (x);`.

Ainsi, la valeur `5 * 3 + 3 = 18` est affectée à `z` au lieu de `30` comme attendu.

Solution : il faut placer des parenthèses autour de l'expression toute entière :

```
#define DOUBLE(a) ((a) + (a))
```

De cette façon, `5 * DOUBLE(x)` est remplacée par `5 * ((x) + (x))` comme désiré.

Macro-instructions à paramètres

Attention, l'usage de macro-instructions à paramètres peut provoquer des **évaluations multiples** d'une même expression.

Considérons par exemple `#define MAX(a, b) ((a) > (b) ? (a) : (b))` et son emploi dans

```
res = MAX(f(t1), f(t2));
```

où `res` est une variable entière, `f` est une fonction qui renvoie un entier et `t1` et `t2` sont des arguments acceptés par `f`.

Cette dernière instruction est remplacée par le préprocesseur en

```
res = ((f(t1)) > (f(t2)) ? (f(t1)) : (f(t2)));
```

Problème : ceci réalise un appel

- à `f` avec l'argument `t1` (ce qui est normal) ;
- à `f` avec l'argument `t2` (ce qui est normal) ;
- à `f` avec l'argument `t1` ou `t2` (ce qui est de trop).

Macro-instructions de contrôle de compilation

Les **macro-instructions de contrôle de compilation** permettent d'ignorer, lors de la compilation, une partie du programme.

Ceci est utile pour **sélectionner** les parties à prendre en compte dans un programme, sans avoir à les (dé)commenter.

On dispose ainsi des constructions

```
#ifdef SYMB
...
#endif
```

```
#ifndef SYMB
...
#endif
```

```
#ifdef SYMB
...
#else
...
#endif
```

À gauche, le code `...` n'est considéré que si l'alias `SYMB` est défini.

Au centre, le code `...` n'est considéré que si l'alias `SYMB` n'est pas défini.

Macro-instructions de contrôle de compilation

– Exemple –

```
#include <stdio.h>

#define GAUCHE_DROITE

#ifdef GAUCHE_DROITE

int rechercher(char *tab,
        int n, char x) {
    int i;
    for (i = 0 ; i < n ; ++i)
        if (tab[i] == x)
            return i;
    return -1;
}
```

```
#else

int rechercher(char *tab,
        int n, char x) {
    int i;
    for (i = n - 1 ; i >= 0 ; --i)
        if (tab[i] == x)
            return i;
    return -1;
}

#endif
```

```
int main() {
    int res;
    char tab[] = "chaine de test";

    res = rechercher(tab, 14, 't');
    printf("%d\n", res);
}
```

Ici, on donne deux algorithmes pour localiser la première occurrence d'une lettre dans un tableau : de la gauche vers la droite, ou bien de la droite vers la gauche.

Ce programme affiche `10` ; si on renomme ou supprime `GAUCHE_DROITE` (en l. 3), il affiche `13`.

2. Habitudes

Plan

2. Habitudes

Mise en page d'un programme

Pour écrire un programme de valeur, il faut soigner les points suivants :

1. l'indentation ;
2. l'organisation des espaces autour des unités lexicales ;
3. le choix des identificateurs ;
4. la documentation.

Pour les deux premiers points, on pourra s'aider du formatage automatique de l'éditeur de code utilisé ou d'outils de formatage de code tel que `clang-format`.

Indentation

L'indentation consiste à disposer des caractères blancs au début de certaines lignes d'un programme.

Contrairement au Python, celle-ci ne modifie pas le comportement d'un programme.

L'objectif est d'augmenter la lisibilité.

Règle : on place **quatre espaces** (pas de tabulation) avant chaque instruction d'un bloc et on incrémente cet espacement de quatre en quatre en fonction de la profondeur des blocs.

– Exemples –

```
/* Correct. */
a = 8;
if (b >= 0) {
    printf("%d\n", a);
    a = 0;
    for (i = 0 ; i <= b ; ++i)
        a /= 2;
}
```

```
/* Incorrect. */
a = 8;
if (b >= 0) {
  printf("%d\n", a);
  a = 0;
  for (i = 0 ; i <= b ; ++i)
    a /= 2;
}
```

Organisation des blocs

Nous avons vu qu'un bloc des une suite d'instructions délimitée par des accolades. Il se trouve attaché à une instruction de branchement ou de boucle. Il peut aussi être indépendant.

On **revient à la ligne** après une **accolade ouvrante**.

L'**accolade fermante** se trouve horizontalement au **niveau du début** de la ligne qui contient l'accolade ouvrante.

– Exemples –

```
/* Correct. */
if (valeur >= 1) {
    valeur -= 1;
}
```

```
/* Incorrect. */
if (valeur >= 1)
{
    valeur -= 1;
}
```

– Exemples –

```
/* Correct. */
valeur = 1;
{
    int a;
    valeur = 10;
}
```

```
/* Incorrect. */
valeur = 1; {
    int a;
    valeur = 10;
}
```

Organisation des espaces

On place une **espace avant et après** chaque opérateur.

– Exemple –

```
/* Correct. */
a = b * 2 + 5;
```

```
/* Incorrect. */
a = b*2 + 5;
```

On utilise les règles habituelles de **typographie** pour l'usage des **virgules**.

– Exemple –

```
/* Correct. */
f(a, b, c, 16);
```

```
/* Incorrect. */
f(a,b,c,16);
```

On ne place **pas d'espace** après une **parenthèse ouvrante** ou avant une **parenthèse fermante**.

– Exemple –

```
/* Correct. */
a = (f(a, 3) + a) * 2;
```

```
/* Incorrect. */
a = (f( a, 3) + a ) * 2;
```

Choix des identificateurs

Les **identificateurs** doivent à la fois **renseigner sur le rôle** des entités auxquelles ils appartiennent (variables, paramètres, fonctions, modules, *etc.*) et être **concis**.

– Exemples –

```
/* Identificateur non explicite. */
v
```

```
/* Identificateur trop long. */
valeur_choisie_pour_le_nombre_parties
```

```
/* Identificateur acceptable. */
nb_parties
```

On fixe la langue au **français** pour leur construction.

Choix des identificateurs

- **Identificateurs de variables**.

  Les seuls identificateurs d'une lettre autorisés sont `i`, `j`, `k`, *etc.*, pour les indices de boucles. Les majuscules sont interdites dans les identificateurs. Dans un identificateur composé de plusieurs mots, ces derniers sont séparés par des sous-tirets.

  – Exemples –

  ```
  /* Correct. */
  nb_parties
  ```

  ```
  /* Incorrect. */
  nbParties
  ```

- **Identificateurs de fonctions**. Mêmes règles que pour les identificateurs de variables.

- **Identificateurs des macro-instructions**.

  Ils sont écrits exclusivement en majuscules et, s'ils sont composés de plusieurs mots, ils sont séparés par des sous-tirets.

  – Exemple –

  ```
  #define TAILLE_MAX 1024
  #define DEBUG
  ```

- **Identificateurs de types**.

  La première lettre commence par une majuscule et, s'ils sont composés de plusieurs mots, chaque première lettre de chaque mot commence par une majuscule.

  – Exemple –

  ```
  typedef ... Liste;
  typedef ... ArbreBinaire;
  ```

Documentation

Un programme est documenté par des commentaires. Ce sont des **phrases**, placées entre `/*` et `*/`, ou à la suite d'un `//`.

On documente chaque **fichier de programme**, au tout début, par

- les prénoms et noms des auteurs ;
- la date de création (au format jour-mois-année) ;
- la date de modification (au format précédent).

– Exemple –

```
/* Auteur : A. M. Turing
 * Creation : 23-06-1912
 * Modification : 07-06-1954 */
```

On commentera le moins possible les instructions.
Il faut impérativement éviter les commentaires inutiles.

– Exemple –

```
/* Incorrect. */
//Affiche la valeur de 'a'.
printf("%d\n", a);
```

Documentation des fonctions

On documente la plupart des **fonctions** par des commentaires situés avant leur déclaration (ou leur définition).

Un commentaire de fonction explique

- le **rôle** de chaque **paramètre** ;
- ce que **renvoie** la fonction ;
- l'**effet** produit par la fonction.

– Exemples –

```c
/* Correct. */
/* Renvoie le plus grand entier parmi `a` et `b`. */
int max(int a, int b) {
    if (a >= b)
        return a;
    return b;
}
```

```c
/* Incorrect. */
/* Calcule le maximum de deux entiers. */
int max(int a, int b) {
    if (a >= b)
        return a;
    return b;
}
```

Plan

2. Habitudes

Options de compilation

Le compilateur est le principal allié du développeur. Il faut l'exploiter au mieux à l'aide des options de compilation.

On utilisera les options nous signalant les (potentiels) problèmes lors de la compilation.

- **Portabilité** : `-std=c17 -pedantic`
- **Avertissements** : `-Wall` mais aussi `-Wextra`

On ajoutera également les options permettant le diagnostique de problème à lors de l'exécution.

- **Debug** : `-g` (pour `valgrind` et `gdb`)
- **Mais aussi les sanitizers** : `-fsanitize=address`, `-fsanitize=undefined` (ou `-fsanitize=address,undefined` pour les deux)

Quelques messages d'erreur/avertissement

– Variable ou fonction non déclarée –

```
error: 'foo' undeclared (first use in this function)

warning: implicit declaration of function 'ma_fonction' [-Wimplicit-function-declaration]

main.c:(.text+0x10): undefined reference to `ma_fonction'
```

– Variable non initialisée –

```
warning: 'foo' is used uninitialized [-Wuninitialized]
```

– Conversion incorrecte de pointeurs –

```
warning: passing argument 1 of 'ma_function' from incompatible pointer type [-Wincompatible-pointer-types]
note: expected 'TypeA *' but argument is of type 'TypeB *'

warning: assignment to 'TypeA *' from incompatible pointer type 'TypeB *' [-Wincompatible-pointer-types]

warning: passing argument 1 of 'ma_fonction' makes integer from pointer without a cast [-Wint-conversion]
note: expected 'int' but argument is of type 'TypeA *'
```

Le `fprintf` comme premier outil

La premiere méthode dont on dispose pour déboguer un code est d'ajouter nous même de quoi tracer l'exécution de notre code. Pour cela, on utilise la fonction `fprintf` et le flux standard d'erreur `stderr`. [1]

```c
#include <stdio.h>
extern FILE * stderr;
int fprintf(FILE *f, char *format, ...);
```

Pour bien utiliser cette fonction, il faut identifier les portions suspectes du code et les encadrer de messages indiquant des informations qui seront potentiellement utiles au débogage.

```c
int ma_fonction(void) {
    ...
    fprintf(stderr, "%s: <Message> var=%d, ptr=%p\n", "ma_fonction", var, ptr);
    // Portion de code suspecte
    ...
    fprintf(stderr, "%s: <Message Fin>\n", "ma_fonction");
```

1. on reverra dans la suite cette fonction pour les entrées/sorties

Construction à partir du `fprintf`

Afin de mieux localiser les messages, on pourra s'aider de macro-instructions fournies par le compilateur :

- `__FILE__` est remplacé par le nom du fichier courrant
- `__LINE__` est remplacé par le numéro de la ligne courrante
- `__func__` est remplacé par le nom de la fonction courrante [2]

```
fprintf(stderr, "%s:%d:%s(): <Message> ...", __FILE__, __LINE__, __func__, ...);
```

En créant nous même une macro-instruction, on peut éviter le code redondant :

```
#define DEBUG_PRINT(fmt, ...) \
    fprintf(stderr, "%s:%d:%s(): " fmt, __FILE__, __LINE__, __func__, __VA_ARGS__)
```

– Exemple –

```
14. void ma_fonction(int arg) {
15.     DEBUG_PRINT("Arguments: arg=%d\n", arg);
16.     ...
```

Le code précédent produit le message suivant dans le flux d'erreur :

```
main.c:15:ma_fonction(): Arguments: arg=12
```

2. il s'agit en fait d'une variable locale et non d'une macro

Utilisation de gdb

gdb est un débogueur et permet de voir ce qu'il se passe lors de l'exécution d'un programme. Il permet une exécution pas à pas.

Pour l'utiliser convenablement il faut compiler nos programmes avec l'option `-g`. Puis lancer ce dernier avec la commande `$ gdb ./mon_programme`.

On entre alors dans un environnement interactif permettant différentes commandes :

- `(gdb) run` : lance l'exécution du programme
- `(gdb) break ma_fonction / fichier.c:<numéro de ligne>` : définit un point d'arrêt à la première instruction de la fonction ou à l'instruction spécifiée
- `(gdb) backtrace` affiche la pile d'appel en cours
- `(gdb) continue` : poursuit l'exécution du programme
- `(gdb) print <variable> / <expression>` : affiche la valeur de l'expression indiquée
- `(gdb) next` exécute la ligne de code suivante
- `(gdb) step` idem, en dans la fonction en cas d'appel

Utilisation de `valgrind`

Depuis 2002, valgrind est un outil permettant (notamment) de tracer l'utilisation de la mémoire d'un programme.

Pour l'utiliser convenablement il faut compiler nos programmes avec l'option `-g`. Puis lancer ce dernier avec la commande `$ valgrind ./mon_programme`.

– Exemple –

Le code suivant provoque une erreur de segmentation.

```
int main(int argc, char *argv[]) {
  return argv[argc][0];
}
```

```
$ valgrind -q ./mon_programme
==13277== Invalid read of size 1
==13277==     at 0x10914B: main (prog.c:2)
==13277==   Address 0x0 is not stack'd, malloc'd or (recently) free'd
==13277==
==13277== Process terminating with default action of signal 11 (SIGSEGV)
```

`valgrind` permet d'obtenir directement la ligne à l'origine de l'erreur.

Les *sanitizers*

Depuis 2012, les *sanitizers* sont un nouvel ensemble d'outils permettant la détection de bugs via l'**instrumentation** de code lors de la compilation. L'exécutable produit à la compilation intègre alors des fonctionnalités de diagnostique détectant notamment erreurs d'allocation dynamique, dépassement de capacité ou fuite de mémoire.

Pour l'utiliser les *sanitizers* il faut compiler avec les options `-g -fsanitize=address,undefined`. Il suffit ensuite de lancer le programme normalement.

– Exemple –

Le code suivant ne libère pas la mémoire avant de terminer.

```c
#include <stdlib.h>
#include <limits.h>
int main(void) {
    int size = INT_MAX;
    char *tab = malloc(size * size);
    return tab[0];
}
```

```
$ ./mon_programme
/path/prog.c:5:26: runtime error: signed integer overflow:
    2147483647 * 2147483647 cannot be represented in type 'int'
=================================================================
==6009==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 1 byte(s) in 1 object(s) allocated from:
    #0 0x7fe3020dd78f in malloc (/usr/lib/libasan.so.8+0xdd78f) ...
    #1 0x559bd2ad11eb in main /path/prog.c:6
    ...
```

Plan

2. Habitudes

Détecter les erreurs

Certains appels à des fonctions peuvent mal se passer.

Ceci peut provoquer une **erreur** provenant principalement

- du **système** (par exemple, lorsqu'il ne parvient pas à satisfaire un `malloc` pour cause de la configuration de la mémoire) ;
- de l'**utilisateur** (par exemple, lorsqu'il communique au programme des données inattendues) ;
- du **programmeur** (par exemple, lorsqu'il appelle une fonction avec des arguments inadéquats qui rendent le calcul impraticable).

Il est nécessaire de

1. pouvoir détecter les erreurs ;
2. pouvoir remédier aux erreurs quand elles surviennent.

Pour cela, nous allons augmenter l'écriture de fonctions de **mécanismes de gestion d'erreur**.

Mécanisme de gestion d'erreurs

On écrira la plupart des fonctions selon le schéma suivant :

- le type de retour est `int` et la valeur de retour, le code d'erreur, renseigne si l'exécution de la fonction s'est bien déroulée ;
- la (ou les) valeur(s) « renvoyée(s) » par la fonction se fait par un (des) passage(s) par adresse.

Schématiquement, une telle fonction admet ainsi le prototype

```
int FCT(T1 E1, ..., TN EN,   U1 S1, ..., UK SK);
```

dit prototype standard où

- les `Ei` sont les paramètres d'entrée ;
- les `Ti` sont des types (potentiellement des adresses) ;
- les `Sj` sont les paramètres de sortie ;
- les `Uj` sont des types (potentiellement des adresses).

Mécanisme de gestion d'erreurs

Le **code d'erreur** d'une fonction ayant un prototype standard suit la spécification suivante :

- une **valeur négative ou nulle** renseigne qu'une erreur s'est produite. En général, la fonction renvoie `0` dans ce cas. On peut être plus précis et exprimer une erreur particulière en renvoyant des autres valeurs négatives.
- une **valeur strictement positive** signifie que l'exécution de la fonction s'est bien déroulée. En général, la fonction renvoie `1` dans ce cas. On peut être plus précis et exprimer une information particulière en renvoyant des autres valeurs strictement positives.

Attention : il y a des fonctions de la bibliothèque standard qui ne suivent pas cette spécification.

De notre côté, nous allons la suivre à la lettre dans les fonctions que nous écrirons.

Remarque : le fonctionnement des codes d'erreurs de chaque fonction écrite doit être spécifié dans sa documentation.

Exemple 1 de gestion d'erreur

– Exemple –

Considérons la fonction

```
int division(float x, float y, float *res) {
    if (y == 0)
        return 0;
    *res = x / y;
    return 1;
}
```

Les entrées sont les flottants `x` et `y`. La sortie est `res` ; c'est une adresse qui pointera sur le résultat de la division de `x` par `y`.

La valeur de retour est un **code d'erreur** : il vaut `0` lorsque la division ne peut pas être calculée (`y` nul) et vaut `1` sinon.

On remarque que l'on ne modifie pas `*res` lorsque le calcul ne peut pas être réalisé.

Exemple 2 de gestion d'erreur

– Exemple –

Considérons la fonction

```
int nb_min_maj(char *chaine, int *res_min, int *res_maj) {
    int i;
    *res_min = 0;
    *res_maj = 0;
    i = 0;
    while (chaine[i] != '\0') {
        if (('a' <= chaine[i]) && (chaine[i] <= 'z'))
            *res_min += 1;
        else if (('A' <= chaine[i]) && (chaine[i] <= 'Z'))
            *res_maj += 1;
        else
            return 0;
        i += 1;
    }
    return i;
}
```

L'entrée est la chaîne de caractères `chaine`. Les sorties sont `res_min` et `res_maj` ; ces adresses pointeront sur le nombre de minuscules et de majuscules dans `chaine`.

La valeur de retour est un **code d'erreur** : il vaut `0` si un caractère non alphabétique apparaît dans `chaine` et vaut la longueur de `chaine` sinon.

Fonctions classiques à gestion d'erreurs

La bibliothèque standard du C contient beaucoup de fonctions à gestion d'erreurs. Par exemple :

- `printf` renvoie le nombre de caractères écrits (sans compter `'\0'`) ;
- `scanf` renvoie le nombre d'affectations réalisées lors de la lecture. La valeur `EOF` est renvoyée si une erreur de lecture a lieu ;
- `malloc` renvoie un pointeur vers la zone de la mémoire allouée. Lorsque l'allocation échoue, la valeur `NULL` est renvoyée ;
- `fopen` renvoie un pointeur sur le fichier ouvert. Lorsque l'ouverture échoue, la valeur `NULL` est renvoyée ;
- `fclose` renvoie `0` si la fermeture du fichier s'est bien déroulée (attention à ce cas particulier). Lorsque la fermeture échoue, la valeur `EOF` est renvoyée.

Remarque : certaines de ces fonctions ont une gestion d'erreurs encore plus sophistiquée et modifient des variables globales comme `errno` (de l'en-tête `errno.h`) pour renseigner précisément sur l'erreur survenue.

Emploi des fonctions à gestion d'erreurs

Les fonctions à gestion d'erreur renvoient des entiers. De ce fait, leur **valeur de retour** est une **expression booléenne**.

Nous pouvons donc combiner l'appel d'une fonction à gestion d'erreurs avec un test pour traiter l'erreur éventuelle.

– Exemples –

En utilisant les deux fonctions précédentes :

```
int a, b;
...
if (division(8, a, &b) == 0) {
    /* Traitement de l'erreur lors
     * de la division par zero. */
}
/* Instructions suivantes. */
```

```
int a, b;
...
if (nb_min_maj("UnDeuxTrois", &a, &b) == 0) {
    /* Traitement de l'erreur lorsque
     * la chaine de caracteres contient
     * des caracteres non alphabetiques. */
}
/* Instructions suivantes. */
```

Interruption de l'exécution

Dans certains cas où une erreur survient, celle-ci peut être irrécupérable. Il faut donc **interrompre l'exécution** du programme. On utilise pour cela la fonction

```
void exit(int status);
```

de `stdlib.h`, appelée avec l'argument `EXIT_FAILURE` (constante qui vaut `1`).

– Exemple –

```
/* Allocation dynamique. */
tab = (int *) malloc(sizeof(int) * 1024);

/* Verification de son succes. */
if (NULL == tab)
    /* Sur son echec, on interrompt
     * l'execution immediatement. */
    exit(EXIT_FAILURE);
/* Instructions suivantes. */
```

Important : on tâchera d'utiliser ce mécanisme d'arrêt exclusivement dans la fonction `main`. Ailleurs, il faut plutôt renvoyer un code d'erreur plutôt que d'interrompre ainsi l'exécution.

Plan

2. Habitudes

Assertions d'entrée

Lors d'un appel à une fonction, certains arguments peuvent être dans un état incohérent.

Au lieu de gérer ces cas de figure par l'usage de codes d'erreur, il est possible de tester l'état des arguments.

Une pré-assertion (ou assertion d'entrée) est un test réalisé dans une fonction pour vérifier si elle est appelée avec des arguments adéquats.

On utilise la fonction

```
void assert(int expr);
```

du fichier d'en-tête `assert.h`. Elle fonctionne de la manière suivante :

- lorsque l'assertion `expr` est **fausse**, l'exécution du programme est **interrompue** et diverses informations utiles sont affichées ;
- lorsque `expr` est **vraie**, l'exécution **se poursuit**.

Exemple 1 de fonction avec pré-assertions

– Exemple –

Considérons la fonction

```
void afficher_tab(int tab[], int nb) {
    assert(tab != NULL);
    assert(nb >= 0);

    for (int i = 0 ; i < nb ; ++i)
        printf("%d\n", tab[i]);
}
```

Elle possède deux pré-assertions :

1. la première teste si le tableau `tab` est bien un pointeur valide (différent de `NULL`) ;
2. la seconde teste si la taille `nb` donnée est bien positive.

Conception de pré-assertions

Il est important de **munir ses fonctions de pré-assertions** les plus précises et complètes possibles. Quelques règles :

- la condition testée ne doit dépendre que des arguments d'une fonction (elle ne dépend pas de données apprises à l'exécution) ;
- la condition testée doit être la plus atomique possible.

– Exemples –

```
/* Correct. */
assert(nb >= 0);
assert(nb <= 1024);
```

```
/* Incorrect (quand on debute, mais correct apres). */
assert((0 <= nb) && (nb <= 1024));
```

- Pour les concevoir, il faut imaginer les pires cas possibles à capturer qui peuvent survenir (par exemple, pointeurs nuls, quantités négatives, chaînes de caractères vides, *etc.*).
- Elles sont situées juste après les déclarations de variables dans le corps d'une fonction.

Redondance nécessaire des pré-assertions

Considérons les fonctions

```
int div(int a, int b) {
    assert(b != 0);
    return a / b;
}
```

```
int somme_div(int a, int b) {
    return div(a, b) + div(b, a + 1);
}
```

Raisonnement : il n'y a pas de pré-assertion dans `somme_div` mais cela n'est pas grave car les cas problématiques sont capturés par `div` qui contient une pré-assertion.

Ceci est une **fausse bonne idée** : chaque fonction doit faire ses propres pré-assertions. Toute erreur doit être capturée le plus en amont possible.

La bonne version de `somme_div` consiste à capturer les mauvaises valeurs possibles de ses arguments de la manière suivante :

```
int somme_div(int a, int b) {
    assert(b != 0);
    assert(a + 1 != 0);
    return div(a, b) + div(b, a + 1);
}
```

Exemple 2 de fonction avec pré-assertion

La fonction `nb_min_maj`, à code d'erreur, doit être pourvue de pré-assertions :

```
int nb_min_maj(char *chaine, int *res_min, int *res_maj) {
    int i;

    assert(chaine != NULL);
    assert(res_min != NULL);
    assert(res_maj != NULL);

    /* Suite inchangee. */
}
```

On observe que le **mécanisme de gestion d'erreurs** par valeur de retour teste des **comportements** incohérents complexes qui se déroulent à l'**exécution**, tandis que le mécanisme de pré-assertion permet de capturer des erreurs évidentes lisibles directement sur les arguments.

Les pré-assertions pour corriger un programme

Considérons le programme

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

int div(int a, int b) {
    assert(b != 0);

    return a / b;
}

int main() {
    int a;

    a = div(17, 0);
    printf("%d\n", a);

    return 0;
}
```

La compilation `clang -std=c17 -pedantic -Wall -o Prgm Prgm.c` donne l'exécutable `Prgm`.

Son exécution `./Prgm` est interrompue en l.5. Elle produit la réponse

```
Prgm: Prgm.c:5: div: Assertion ‘b != 0’ failed.
Aborted (core dumped)
```

On récolte la précieuse information sur le numéro de ligne de la pré-assertion non satisfaite qui produit l'arrêt précipité de l'exécution.

3. Modules

Plan

3. Modules

Modules

Modulariser un projet signifie le découper de manière cohérente en plusieurs parties plus petites.

Un module est un ensemble de données et d'instructions qui permettent de gérer une partie bien ciblée d'un projet.

Il existe deux manières de concevoir un projet :

1. programmer dans un unique fichier contenant tout le code nécessaire ;
2. programmer dans divers fichiers qui fractionnent le projet en plusieurs sous-parties.

À partir de maintenant, on adoptera la 2e manière.

Il reste à savoir comment **découper un projet de manière cohérente** et comment **utiliser les outils offerts par le langage** pour gérer ce découpage.

Avantages offerts par la modularité

La modularité, illustration du principe stratégique universel

« *diviser pour régner* »,

offre les avantages suivants.

1. La **lisibilité** du code est accrue, ainsi que la facilité de son **entretien**.
2. Permet de **regrouper** les types et les fonctions selon leurs objectifs.
3. Il devient possible de **réutiliser** dans un nouveau projet un module créé dans un projet antérieur.
4. Permet de **cacher des fonctions** (notion de fonctions privées).
5. Facilite le **travail en équipe**.
6. Permet de rendre la **compilation localisée** (compilation module par module).

3. Modules

Spécification d'un projet

Considérons le **projet spécifié** de la manière suivante :

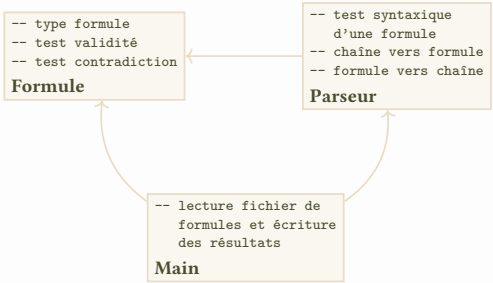
- le but est de fournir un programme qui permet de décider si des formules logiques sans quantificateur sont valides ou contradictoires.
- La syntaxe d'une formule est la suivante : on dispose du jeu de formules atomiques a, b, …, z et on écrit les formules de manière infixe et totalement parenthésée. Par exemple, la formule $(A \to (B \vee \neg C)) \wedge A$ s'écrit `((a IMP (b OU (NON c))) ET a)`.
- L'interaction utilisateur / programme se fait de la manière suivante :
  1. l'utilisateur fournit un fichier en entrée contenant une formule par ligne ;
  2. le programme produit un fichier en sortie contenant, ligne par ligne, la réponse `erreur` si la formule correspondante est syntaxiquement erronée ou bien `valide`, `contradictoire` ou `rien` selon la nature de la formule.

Découpage du projet

Il y a deux parties bien distinctes dans ce projet :

1. la **représentation des formules** et le test de validité / contradiction ;
2. la **gestion syntaxique des formules** (lecture / écriture d'une formule dans un fichier).

Ces deux parties dictent le découpage suivant :



Toute flèche $A \longrightarrow B$ signifie que le module A dépend du module B.

Plan

3. Modules

Composition d'un module

Un module est composé de deux fichiers :

1. un fichier d'en-tête d'extension `.h` ;
2. un fichier source d'extension `.c`.

Les noms de ces deux fichiers sont les mêmes (extension mise à part).

Seul le module principal est constitué d'un seul fichier source `Main.c`.

Par exemple, le projet précédent est constitué des fichiers
`Formule.h`, `Formule.c`, `Parseur.h`, `Parseur.c` et `Main.c`.

Fichiers d'en-tête

Un fichier d'en-tête contient des déclarations de types et de fonctions (prototypes).

Les prototypes qui y figurent sont ceux des fonctions que l'on souhaite rendre visibles (utilisables) par d'autres modules.

– Exemple –

Un en-tête possible du module `Formule` est

```
/* Formule.h */

typedef struct {
    ...
} Form;

int est_valide(Form *f);
int est_contra(Form *f);
```

Fichiers d'en-tête

Les fichiers d'en-tête sont ceux que le programmeur regarde en premier pour connaître le **rôle d'un type ou d'une fonction**.

De ce fait, c'est dans les fichiers d'en-tête que l'**on commente chaque type et fonction** pour préciser leur rôle.

– Exemple –

L'en-tête du module `Formule` devrait être de la forme

```
/* Formule.h */

/* Representation des formules logiques sans
 * quantificateur. */
typedef struct {
    ...
} Form;

/* Renvoie '1' si la formule pointee par 'f'
 * est valide et '0' sinon. */
int est_valide(Form *f);
...
```

Fichiers sources

Un fichier source contient une implantation de l'en-tête du module auquel il appartient.

Il contient de ce fait les définitions de fonctions déclarées dans l'en-tête.

– Exemple –

Une implantation possible du module `Formule` est

```c
/* Formule.c */

...
int est_valide(Form *f) {
    int i, j;
    assert(f != NULL);
    ...
}

int est_contra(Form *f) {
    ...
}
```

L'ordre de définition des fonctions n'est pas important.
Nous en verrons la raison dans la suite.

Fichiers sources

Il faut impérativement que toutes les fonctions déclarées dans le fichier d'en-tête du module soient définies dans le fichier source correspondant.

Il est en revanche possible de définir dans un fichier source des fonctions qui ne sont pas déclarées dans l'en-tête correspondant.

Ainsi,

- une fonction définie dans un fichier source mais pas dans l'en-tête correspondant s'appelle fonction privée ;
- l'intérêt des fonctions privées est d'être des fonctions outils dont le champ d'application est **local au module** dans lequel elles sont définies.

  On ne souhaite pas les rendre utilisables en dehors.

Fichiers sources et fonctions privées

Une **fonction privée** se définit avec le mot clé `static` (à ne pas confondre avec le `static` pour la déclaration de variables).

– Exemple –

Nous pouvons avoir besoin d'une fonction privée appartenant au module `Formule` qui permet de compter le nombre d'occurrences d'un atome dans une formule.

On la définit alors dans `Formule.c` par

```
static int nb_occ(Form *f, char atome) {
    ...
    assert(f != NULL);
    assert(('a' <= atome) && (atome <= 'z'));
    ...
}
```

La **portée lexicale** de cette fonction s'étend à tout ce qui suit sa définition dans le fichier `Formule.c`. Elle est invisible ailleurs.

Fichiers sources et fonctions privées

C'est un contresens (excusable) que de définir une fonction dans un fichier source sans le mot clé `static` et sans l'avoir déclarée dans le fichier d'en-tête.

C'est aussi un contresens (inexcusable) que de définir dans un fichier source une fonction déclarée dans le fichier d'en-tête avec `static`.

Ainsi, pour résumer, toute fonction définie dans un fichier source est

1. soit déclarée dans le fichier d'en-tête ;
2. soit non déclarée dans le fichier d'en-tête mais définie par `static`.

Allure d'un projet

Il y a deux manières d'**organiser** un projet en termes de fichiers et de répertoires :

1. la 1re consiste à regrouper l'ensemble des fichiers d'en-tête et des fichiers sources dans un même répertoire. Il y figure donc un nombre impair de fichiers (le module principal et les paires en-tête / source pour chaque module) ;

2. la 2e consiste à séparer les fichiers du projet en deux répertoires frères, `include` et `src`, le premier contenant les fichiers d'en-tête et l'autre, les fichiers sources et le module principal du projet.

Les deux sont correctes. La 1re organisation a le mérite d'être plus simple. La 2e a le mérite d'être plus structurée mais en contrepartie, les chemins d'inclusions sont plus complexes.

3. Modules

Inclusion de modules

Pour utiliser un module `Module` dans un fichier `F`, on doit l'y inclure.

On utilise pour cela dans `F` la commande pré-processeur

```
#include "Module.h"
```

Celle-ci sera remplacée par le pré-processeur par le contenu de `Module.h`.

Cette commande peut se trouver

- dans un fichier source pour bénéficier des fonctions définies et des types déclarés par le module ;
- dans un fichier d'en-tête pour bénéficier des types déclarés par le module.

Inclusion de modules

Si `A` est un module définissant une fonction `f`, pour utiliser `f` dans un fichier `B.c` situé dans le même répertoire que `A.c` et `A.h`, on écrit

```
/* B.c */

#include "A.h"
...
```

Il est possible d'inclure à la suite plusieurs modules dans un même fichier :

```
/* Fichier.c ou Fichier.h */

#include "A.h"
#include "B.h"
#include "C.h"
...
```

De cette manière, `Fichier.c` ou `Fichier.h` bénéficie de tout ce qui est déclaré et défini dans les modules `A`, `B` et `C`.

Attention : les modules inclus ne doivent pas déclarer / définir des éléments ayant un même identificateur.

Inclusion de modules

Le fichier incluant n'a **accès** qu'au fichier d'en-tête du module, et donc qu'aux **déclarations effectuées**.

Le fichier incluant n'a pas besoin de connaître l'implantation du module.

– Exemple (début) –

Considérons le module `Couple` défini par

```c
/* Couple.h */

typedef int Couple[2];

int est_zero(Couple c);

void afficher(Couple c);
```

```c
/* Couple.c */
...
int est_zero(Couple c) {
    return (c[0] == 0) && (c[1]== 0);
}

void afficher(Couple c) {
    printf("(%d, %d)", c[0], c[1]);
}
```

Inclusion de modules

– Exemple (suite et fin) –

Supposons que l'on ait besoin d'inclure le module `Couple` dans un fichier `Fichier.c`.

Le pré-processeur aura donc l'effet suivant sur `Fichier.c` :

```c
/* Fichier.c avant la passe du
 * pre-processeur */

#include "Couple.h"
...

if (!est_zero(c))
    afficher(c);
...
```

```c
/* Fichier.c apres la passe
 * du pre-processeur */

typedef int Couple[2];
int est_zero(Couple c);
void afficher(Couple c);
...

if (!est_zero(c))
    afficher(c);
...
```

`Fichier.c` a besoin uniquement de connaître les types de retour et les signatures des fonctions qu'il invoque (connus à leur déclaration). Il n'a à ce stade pas besoin de connaître les définitions de ces fonctions.

Création complète d'un module

Pour créer un module `A`, il faut inclure son fichier d'en-tête `A.h` dans son fichier source `A.c`.

```
/* A.h */

...
```

```
/* A.c */

#include "A.h"
...
```

De cette manière,

- d'une part, `A.c` a accès aux types et aux prototypes de fonctions déclarés dans `A.h` ;
- d'autre part, cela permet d'implanter les fonctions déclarées dans `A.h` sans contrainte d'ordre.

Création complète d'un module

Considérons la situation suivante :

```
/* A.h */
...
#include "C.h"
...
```

```
/* A.c */
#include "A.h"
...
```

```
/* B.h */
...
#include "C.h"
...
```

```
/* B.c */
#include "B.h"
...
```

```
/* C.h */
...
int f();
...
```

```
/* C.c */
#include "C.h"
...
```

```
/* D.c */
#include "A.h"
#include "B.h"
...
```

Le pré-processeur transforme ces fichiers en

```
/* A.h */
...
int f();
...
```

```
/* A.c */
/* Copie A.h */
...
```

```
/* B.h */
...
int f();
...
```

```
/* B.c */
/* Copie B.h */
...
```

```
/* C.h */
...
int f();
...
```

```
/* C.c */
/* Copie C.h */
...
```

```
/* D.c */
int f();
int f();
...
```

Problème : le contenu de `C.h` est copié deux fois dans `D.c`. Ceci n'est pas accepté par le compilateur car il y a **multiple déclaration** d'un même symbole (`f` ici).

Création complète d'un module

La parade consiste à **inclure** un fichier d'en-tête de **manière conditionnelle** : on procède à l'inclusion que s'il n'a pas déjà été inclus.

On utilise pour cela les macro-instructions de contrôle de compilation `#ifndef` et `#endif` ainsi que `#define`.

Le schéma général est

```
/* A.h */

#ifndef A_H
#define A_H

    /* Declaration de types */
      ...

    /* Declaration de fonctions */
      ...

#endif
```

Ainsi, lors d'une inclusion de `A.h`, le pré-processeur vérifie si la macro `A_H` n'existe pas.

- Si elle n'existe pas, alors on la définit (`#define A_H`) et le contenu du module est pris en compte ;
- sinon, cela signifie que le contenu a déjà été pris en compte. Celui-ci n'est pas repris en compte une 2^e fois.

Squelette d'un module

Pour résumer, tous les modules doivent avoir le squelette suivant :

```
/* A.h */

#ifndef A_H
#define A_H

    /* Inclusions eventuelles de modules */
    ...

    /* Definitions eventuelles de macros */
    ...

    /* Declarations eventuelles de types */
    ...

    /* Declarations eventuelles de fonctions */
    ...

#endif
```

```
/* A.c */

#include "A.h"

/* Inclusions eventuelles de modules */
...

/* Definitions eventuelles de fonctions privees */
...

/* Definitions de toutes les fonctions declarees
 * dans le fichier d'en-tete */
...
```

Exemple du module `Parseur`

– Exemple –

```c
/* Parseur.h */

#ifndef PARSER_H
#define PARSER_H

#include "Formule.h"

    /* Convertit la chaine de caracteres `ch` sensee
     * representer une formule en une variable de type
     * `Form`, qui va etre ecrite dans `f`. Renvoie `1`
     * si `ch` represente bien une formule et `0` sinon. */
    int chaine_vers_form(Form *f, char *ch);

#endif
```

```c
/* Parseur.c */

#include "Parseur.h"

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int chaine_vers_form(Form *f, char *ch) {
    ...
    assert(f != NULL);
    ...
}
```

Plan

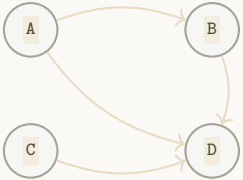
3. Modules

Graphes d'inclusions

On rappelle qu'un module `A` **dépend** d'un module `B` si le **fichier d'en-tête** de `A` inclut `B`.

Pour visualiser l'allure d'un projet, on trace son graphe d'inclusions. On représente pour cela chacun des modules qui le composent dans des cercles (sommets) et on trace des flèches (arcs) de `A` vers `B` pour tout module `A` dépendant de `B`.

– Exemple –



Ce graphe d'inclusions signifie que

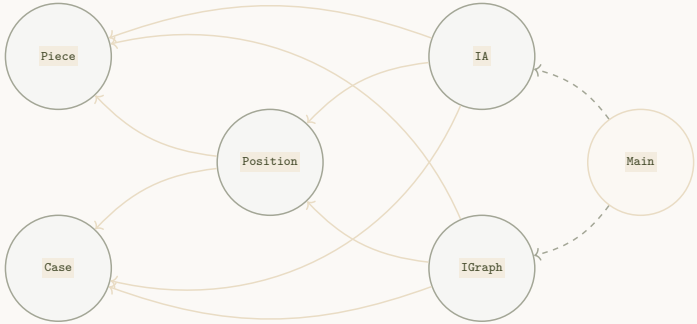
- `A.h` inclut `B.h` et `D.h` ;
- `B.h` inclut `D.h` ;
- `C.h` inclut `D.h` ;
- `D.h` n'inclut rien.

On ne mentionne pas dans les graphes d'inclusions les inclusions aux fichiers d'en-tête standards (`stdio.h`, `stdlib.h`, `assert.h`, *etc.*).

Graphe d'inclusions et fichier principal

– Exemple –

Le graphe d'inclusions d'un projet consistant à faire jouer l'ordinateur aux échecs contre un humain peut être le suivant :



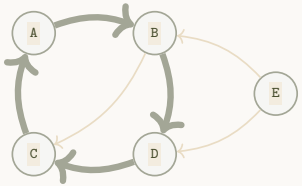
Tout projet contient un fichier source `Main.c`, **le fichier principal** du projet, où figure la fonction `main` (le point d'entrée de l'exécution du programme). Celui-ci apparaît dans le graphe d'inclusions.

Inclusions circulaires

Les graphes d'inclusions permettent d'avoir une vision globale de l'architecture d'un projet.

Ils permettent aussi de mettre en évidence des problèmes de conception et notamment les problèmes d'**inclusion circulaire**. Ce type de problème s'observe par la présence d'un **cycle** dans le graphe d'inclusions.

– Exemple –



Règle importante : il ne doit jamais y avoir de cycle dans le graphe d'inclusions d'un projet. S'il y a un cycle, c'est que le projet est **mal découpé en modules**.

Limiter les inclusions

La plupart des inclusions circulaires peuvent être évitées en **réduisant au maximum les inclusions** de modules dans les **fichiers d'en-tête** en les faisant plutôt si possible dans les fichiers sources.

– Exemple (début) –

Supposons que l'on dispose d'un module `Tri` qui permet de trier des tableaux génériques (nous aborderons plus loin ce concept de généricité). Il est de la forme :

```
/* Tri.h */
#ifndef TRI_H
#define TRI_H

    void trier_tab(void **t, int n,
        int (*est_inf)(void *, void *));
    ...

#endif
```

```
/* Tri.c */
#include "Tri.h"
...
void trier_tab(void **t, int n,
    int (*est_inf)(void *, void *)) {
    ...
}
...
```

Limiter les inclusions

– Exemple (suite et fin) –

On souhaite maintenant écrire un module `TabInt` pour gérer des tableaux d'entiers.

On n'écrira pas

```c
/* TabInt.h */
#ifndef TAB_INT_H
#define TAB_INT_H

#include "Tri.h"

    typedef struct {int n; int *tab;} TabInt;
    int trier_tab_int(TabInt *t);

#endif
```

```c
/* TabInt.c */
#include "TabInt.h"

int trier_tab_int(TabInt *t) {
    /* Util. de 'trier_tab' */
}
```

mais plutôt

```c
/* TabInt.h */
#ifndef TAB_INT_H
#define TAB_INT_H

    typedef struct {int n; int *tab;} TabInt;
    int trier_tab_int(TabInt *t);

#endif
```

```c
/* TabInt.c */
#include "TabInt.h"

#include "Tri.h"

int trier_tab_int(TabInt *t) {
    /* Util. de 'trier_tab' */
}
```

Dépendances étendues

Nous serons amenés dans la suite (dans le cadre de la compilation séparée) — étant donné un fichier `A.c` — à considérer l'ensemble des fichiers d'en-tête qu'il inclut.

Si `A.c` inclut un fichier d'en-tête `B.h`, alors le module `A` dépend de manière étendue au module `B`.

Le graphe d'inclusions étendu d'un projet consiste en le graphe d'inclusions du projet dans lequel sont ajoutées des **flèches en pointillés** pour symboliser les dépendances étendues.

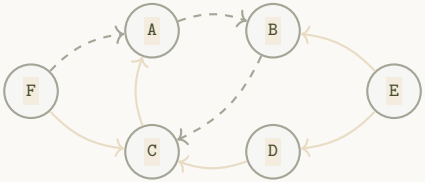
Note 1. : les flèches qui partent du module principal `Main` représentent des inclusions étendues.

Note 2. : les cycles dans lesquels intervient au moins une flèche en pointillés ne posent pas de problème de structure du projet.

Dépendances étendues

– Exemple –

Le graphe d'inclusions étendu



nous renseigne sur le fait que

- `A.h` n'inclut rien ;
- `A.c` inclut `B.h` ;
- `B.h` n'inclut rien ;
- `B.c` inclut `C.h` ;
- `C.h` inclut `A.h` ;
- `C.c` n'inclut rien ;
- `D.h` inclut `C.h` ;
- `D.c` n'inclut rien ;
- `E.h` inclut `B.h` et `D.h`
- `E.c` n'inclut rien ;
- `F.h` inclut `C.h` ;
- `F.c` inclut `A.h`.

On observe que ce projet n'est pas mal structuré car il ne possède pas de cycle formé uniquement par des flèches de dépendance.

Plan

3. Modules

Erreur : le fichier d'en-tête général

Une **erreur** consiste, pour un projet donné, à développer un fichier `Types.h` et plusieurs fichiers source `F1.c`, `F2.c`, …, `Fn.c`.

Ici le fichier d'en-tête `Types.h` contient les déclarations de tous les types et fonctions nécessaires au projet et les fichiers sources `Fi.c` implantent chacun un sous-ensemble des fonctions déclarées.

Cette conception est erronée puisque :

1. il n'y a plus de notion de module (et donc tous leurs avantages relatifs sont absents) ;
2. il est impossible de réutiliser du code du projet pour un nouveau (il faudrait copier / coller les types et fonctions importantes, ce qui n'est pas abordable) ;
3. le fichier `Types.h` peut contenir des déclarations de types et de fonctions qui n'ont pas grand chose à voir.

Erreur : économie d'inclusions

Une **erreur** consiste à éviter volontairement de réaliser des inclusions de modules dans d'autres si l'inclusion est déjà réalisée de manière transitive.

– Exemple –

Plus explicitement, soient trois modules `A`, `B` et `C` tels que `B` inclut `C`, `A` inclut `B` et `A` inclut `C` :

On peut être tenté de n'inclure que `B` dans `A` et `C` dans `B` car — par transitivité — ceci entraîne que `C` est inclut dans `A`. Ceci fonctionne en pratique.

Cette conception est cependant erronée puisque :

1. savoir de quels modules dépend `A` simplement en lisant son fichier d'en-tête, sans avoir de surprise sur les modules qui peuvent être inclus de manière cachée par transitivité, est un avantage ;
2. le jour où l'on modifie `B` de sorte qu'il n'ait plus besoin de dépendre de `C` provoque le fait que `C` n'est plus inclut dans `A`, ce qui est problématique.

Habitude : un module par type

Une **bonne façon de faire** par défaut consiste à créer un module pour chaque type nécessaire à l'écriture d'un projet.

Avec ce point de vue, il y a dans chaque `A.h` une déclaration de type unique (dont le nom est `A`, celui du module) et des déclarations de fonctions qui agissent sur des éléments de type `A`.

Cette conception est correcte mais un peu limitée car

1. elle dispense d'une réflexion approfondie sur un bon découpage en modules du projet ;
2. des « types de travail » ne méritent pas d'appartenir à un module dédié ;
3. un projet compterait ainsi trop de modules.

En pratique, commencer la réflexion d'un découpage en modules d'un projet en se posant la question

« De quels types ai-je besoin ? »

fournit un point de départ efficace, à raffiner ensuite.

4. Compilation

Plan

4. Compilation

Compilation d'un projet d'un fichier

La compilation d'un projet constitué d'un **unique fichier** `Fichier.c` contenant la fonction principale `main` se fait par la commande

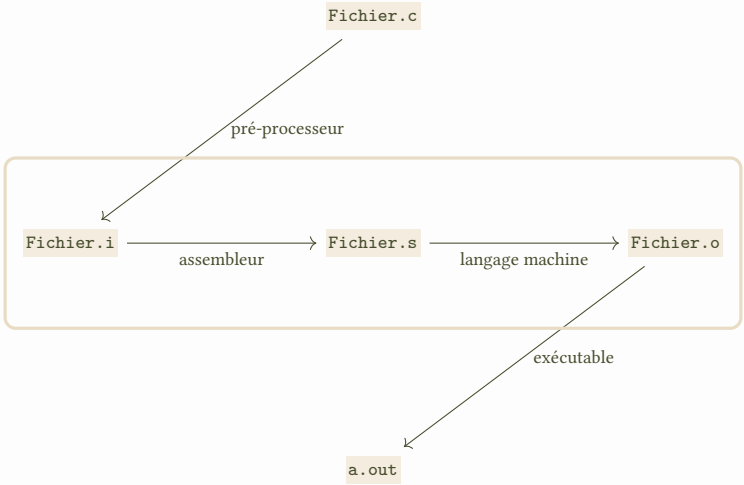
```
gcc Fichier.c
```

Cette commande réalise à la suite les étapes suivantes :

1. traitement préliminaire par le **pré-processeur** ;
2. compilation en **langage assembleur** ;
3. traduction du langage assembleur en **langage machine** ;
4. **édition des liens**.

Elle permet d'obtenir finalement un fichier **exécutable**.

Compilation d'un projet d'un fichier



Pré-processeur

Le **pré-processeur** réalise un pré-traitement du fichier source pour le rendre traduisible en langage machine.

Il procède en

1. supprimant les commentaires ;
2. incluant les fichiers d'en-tête (copie / colle les fichiers `.h` inclus) ;
3. traitant les définitions de symboles par un mécanisme de substitution (`#define`) ;
4. traitant les macro-instructions de contrôle de compilation (`#ifndef`, `#endif`, *etc.*).

Il est possible de récupérer le fichier d'extension `.i` ainsi obtenu par la commande

```
gcc -E Fichier.c >> Fichier.i
```

Compilation en assembleur

Après avoir été traité par le pré-processeur, le fichier `Fichier.i` est **traduit en assembleur**.

Il est possible de récupérer le fichier d'extension `.s` ainsi obtenu par la commande

```
gcc -S Fichier.c
```

L'assembleur est un langage très proche de la machine. Il peut se traduire assez facilement en un langage directement exécutable par le processeur.

Il existe plusieurs langages d'assemblage différents : au moins un par architecture.

Compilation en assembleur

– Exemple –

Avec le fichier `Fichier.c` suivant :

```c
/* Fichier.c */

#include <stdio.h>

int main() {
    printf("Bonjour");
    return 0;
}
```

on obtient le fichier assembleur `Fichier.s` suivant :

```
    .file "Fichier.c"
    .section .rodata
.LC0:
    .string "Bonjour"
    .text
    .globl main
    .type main, @function
main:
.LFB0:
    .cfi_startproc
    pushq %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq %rsp, %rbp
    .cfi_def_cfa_register 6
    movl $.LC0, %edi
    movl $0, %eax
    call printf
    movl $0, %eax
    popq %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size main, .-main
    .ident "GCC: (Ubuntu/Linaro 4.8.1-10 ubuntu9) 4.8.1"
    .section .note.GNU-stack,"",@progbits
```

Traduction en langage machine

Le code assembleur `Fichier.s` est traduit en **langage machine**.

On obtient ce fichier d'extension `.o` par la commande

```
gcc -c Fichier.c
```

Ce fichier s'appelle fichier objet.

Il est illisible pour un humain mais peut cependant être affiché au moyen de la commande

`od -x Fichier.o` ou bien `od -a Fichier.o`

Le langage machine est directement compris par le processeur qui peut de ce fait exécuter directement les instructions qu'il contient.

Traduction en langage machine

– Exemple (1 / 6) –

Avec le programme précédent, le contenu de `Fichier.o` est

```
0000000 del   E   L   F stx soh soh nul nul nul nul nul nul nul nul nul
0000020 soh nul   >  nul soh nul nul nul nul nul nul nul nul nul nul nul
0000040 nul nul nul nul nul nul nul nul   0 soh nul nul nul nul nul nul
0000060 nul nul nul nul   @  nul nul nul nul nul   @  nul  cr nul  nl nul
0000100   U   H  ht   e   ?  nul nul nul nul   8  nul nul nul nul   h nul
0000120 nul nul nul   8  nul nul nul nul   ]   C   B   o   n   j   o   u
0000140   r nul nul   G   C   C   :  sp   (   U   b   u   n   t   u   /
0000160   L   i   n   a   r   o  sp   4   .   8   .   1   -   1   0   u
0000200   b   u   n   t   u   9   )  sp   4   .   8   .   1 nul nul nul
0000220 dc4 nul nul nul nul nul nul nul soh   z   R  nul soh   x dle soh
0000240 esc  ff bel  bs dle soh nul nul  fs nul nul nul  fs nul nul nul
0000260 nul nul nul nul sub nul nul nul nul   A  so dle ack stx   C  cr
0000300 ack   U  ff bel  bs nul nul nul nul   .   s   y   m   t   a   b
0000320 nul   .   s   t   r   t   a   b nul   .   s   h   s   t   r   t
0000340   a   b nul   .   r   e   l   a   .   t   e   x   t nul   .   d
0000360   a   t   a nul   .   b   s   s nul   .   r   o   d   a   t   a
```

Traduction en langage machine

– Exemple (2 / 6) –

```
0000400 nul  .   c   o   m   m   e   n   t  nul  .   n   o   t   e   .
0000420  G   N   U   -   s   t   a   c   k  nul  .   r   e   l   a   .
0000440  e   h   _   f   r   a   m   e  nul nul nul nul nul nul nul nul
0000460 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
   *
0000560  sp nul nul nul soh nul nul nul ack nul nul nul nul nul nul nul
0000600 nul nul nul nul nul nul nul nul  @  nul nul nul nul nul nul nul
0000620 sub nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0000640 soh nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0000660 esc nul nul nul eot nul nul nul nul nul nul nul nul nul nul nul
0000700 nul nul nul nul nul nul nul nul dle enq nul nul nul nul nul nul
0000720  0  nul nul nul nul nul nul nul  vt  nul nul nul soh nul nul nul
0000740  bs nul nul nul nul nul nul nul can nul nul nul nul nul nul nul
0000760  &  nul nul nul soh nul nul nul etx nul nul nul nul nul nul nul
0001000 nul nul nul nul nul nul nul nul  Z  nul nul nul nul nul nul nul
0001020 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001040 soh nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
```

Traduction en langage machine

– Exemple (3 / 6) –

```
0001060  ,  nul nul nul  bs  nul nul nul etx nul nul nul nul nul nul nul
0001100 nul nul nul nul nul nul nul nul  Z  nul nul nul nul nul nul nul
0001120 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001140 soh nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001160  1  nul nul nul soh nul nul nul stx nul nul nul nul nul nul nul
0001200 nul nul nul nul nul nul nul nul  Z  nul nul nul nul nul nul nul
0001220  bs nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001240 soh nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001260  9  nul nul nul soh nul nul nul  0  nul nul nul nul nul nul nul
0001300 nul nul nul nul nul nul nul nul  b  nul nul nul nul nul nul nul
0001320  ,  nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001340 soh nul nul nul nul nul nul nul soh nul nul nul nul nul nul nul
0001360  B  nul nul nul soh nul nul nul nul nul nul nul nul nul nul nul
0001400 nul nul nul nul nul nul nul nul  so nul nul nul nul nul nul nul
0001420 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001440 soh nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001460  W  nul nul nul soh nul nul nul stx nul nul nul nul nul nul nul
```

Traduction en langage machine

– Exemple (4 / 6) –

```
0001500 nul nul nul nul nul nul nul nul dle nul nul nul nul nul nul nul
0001520   8 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001540  bs nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001560   R nul nul nul eot nul nul nul nul nul nul nul nul nul nul nul
0001600 nul nul nul nul nul nul nul nul   @ enq nul nul nul nul nul nul
0001620 can nul nul nul nul nul nul nul  vt nul nul nul  bs nul nul nul
0001640  bs nul nul nul nul nul nul nul can nul nul nul nul nul nul nul
0001660 dc1 nul nul nul etx nul nul nul nul nul nul nul nul nul nul nul
0001700 nul nul nul nul nul nul nul nul   H nul nul nul nul nul nul nul
0001720   a nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001740 soh nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0001760 soh nul nul nul stx nul nul nul nul nul nul nul nul nul nul nul
0002000 nul nul nul nul nul nul nul nul   p eot nul nul nul nul nul nul
0002020  bs soh nul nul nul nul nul nul  ff nul nul nul  ht nul nul nul
0002040  bs nul nul nul nul nul nul nul can nul nul nul nul nul nul nul
0002060  ht nul nul nul etx nul nul nul nul nul nul nul nul nul nul nul
0002100 nul nul nul nul nul nul nul nul   x enq nul nul nul nul nul nul
```

Traduction en langage machine

– Exemple (5 / 6) –

```
0002120 etb nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0002140 soh nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0002160 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0002200 nul nul nul nul nul nul nul nul soh nul nul nul eot nul  q  del
0002220 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0002240 nul nul nul nul etx nul soh nul nul nul nul nul nul nul nul nul
0002260 nul nul nul nul nul nul nul nul nul nul nul etx nul etx nul
0002300 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0002320 nul nul nul nul etx nul eot nul nul nul nul nul nul nul nul nul
0002340 nul nul nul nul nul nul nul nul nul nul nul etx nul enq nul
0002360 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0002400 nul nul nul nul etx nul bel nul nul nul nul nul nul nul nul nul
0002420 nul nul nul nul nul nul nul nul nul nul nul etx nul  bs  nul
0002440 nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
0002460 nul nul nul nul etx nul ack nul nul nul nul nul nul nul nul nul
0002500 nul nul nul nul nul nul nul nul  vt  nul nul nul dc2 nul soh nul
0002520 nul nul nul nul nul nul nul nul sub nul nul nul nul nul nul nul
```

Traduction en langage machine

– Exemple (6 / 6) –

```
0002540 dle nul nul nul dle nul nul nul nul nul nul nul nul nul nul nul
0002560 nul nul nul nul nul nul nul nul nul  F   i   c   h   i   e   r
0002600  .   c  nul  m   a   i   n  nul  p   r   i   n   t   f  nul nul
0002620 enq nul nul nul nul nul nul nul  nl nul nul nul enq nul nul nul
0002640 nul nul nul nul nul nul nul nul  si nul nul nul nul nul nul nul
0002660 stx nul nul nul  nl nul nul nul   |  del del del del del del del
0002700  sp nul nul nul nul nul nul nul stx nul nul nul stx nul nul nul
0002720 nul nul nul nul nul nul nul nul
0002730
```

Édition des liens

L'édition des liens réunit le fichier objet et le code propre aux fonctions et types de la bibliothèque standard utilisés (comme `printf`, `scanf`, *etc.*) pour produire l'exécutable complet.

Avant l'édition des liens, seuls les prototypes des fonctions sont connus du compilateur. Cela lui permet de vérifier que les types sont bien respectés.

Cependant, pour l'obtention finale de l'exécutable qui va suivre, il est nécessaire de **connaître le comportement des fonctions** (c.-à-d. leur définition).

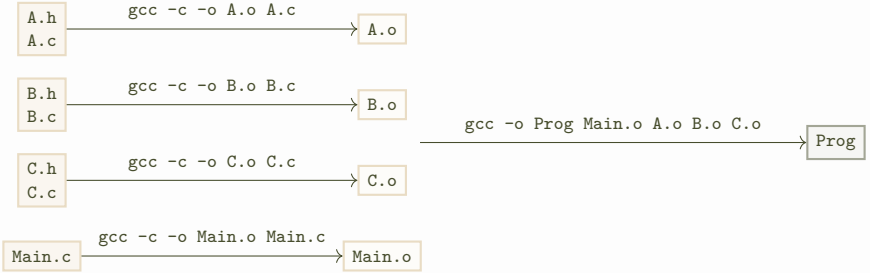
C'est précisément dans cette phase de la compilation que la **résolution des symboles** a lieu. C'est l'étape qui consiste à associer aux identificateurs de fonctions leur implantation.

Compilation d'un projet de plusieurs fichiers

On suppose que l'on travaille sur un projet constitué de trois modules `A`, `B` et `C` et d'un fichier principal `Main.c` contenant la fonction `main`.

La compilation de ce projet se réalise au moyen des étapes suivantes :

1. obtenir les fichiers objets de chaque module ;
2. obtenir le fichier objet de `Main.c` ;
3. lier les fichiers objets ainsi obtenus en un exécutable.



Création des fichiers objets

Pour compiler le projet, on commence par créer un fichier objet pour chaque module `M`. On utilise pour cela la commande

```
gcc -Wall -std=c17 -pedantic -c -o M.o M.c
```

Chaque module est ainsi **compilé séparément** et dans un **ordre quelconque**.

Symboles non résolus

Rappel : pour compiler un module `A`, il n'est pas nécessaire que `A` ait connaissance des définitions des symboles qu'il utilise.

Seules **leurs déclarations sont suffisantes**. Celles-ci se trouvent dans les fichiers d'en-tête inclus dans `A`.

On dit qu'un symbole n'est pas résolu à un stade donné de la compilation si sa définition n'est pas encore connue.

– Exemple –

```c
/* Fichier.c */

int g(int x);

int f(int x) {
    return g(x);
}
```

Ce fichier permet de produire un fichier objet sur la commande `gcc -c -o Fichier.o Fichier.c` même si le symbole `g` est non résolu pour le moment.

Sa déclaration (dans le fichier lui-même ou dans un fichier inclus) est cependant nécessaire (pour que le compilateur connaisse son prototype).

Résolution des symboles

Lors de l'édition des liens, un exécutable est créé. On utilise pour cela la commande

```
gcc -o Prgm Main.o A1.o ... An.o
```

dans le cadre d'un projet constitué des modules `A1`, …, `An` et du module principal `Main`.

Cette étape **lie à chaque symbole sa définition**.

Tous les symboles utilisés dans le projet **doivent être résolus** (sinon, un message d'erreur est produit et l'exécutable ne peut pas être construit).

Résolution des symboles — exemple

– Exemple (1 / 2) –

```c
/* A.h */
#ifndef A_H
#define A_H
    int f(int x);
#endif
```

```c
/* A.c */
#include "A.h"
int f(int x) {
    return x * x;
}
```

```c
/* B.h */
#ifndef B_H
#define B_H
    int g(int x);
#endif
```

```c
/* B.c */
#include "B.h"
#include "A.h"
int g(int x) {
    return f(x);
}
```

```c
/* Main.c */
#include "B.h"
int main() {
    g(5);
    return 0;
}
```

Pour compiler ce projet, on emploie les commandes

```
gcc -Wall -std=c17 -pedantic -c -o A.o A.c
gcc -Wall -std=c17 -pedantic -c -o B.o B.c
gcc -Wall -std=c17 -pedantic -c -o Main.o Main.c
gcc -o Prgm Main.o A.o B.o
```

L'ordre d'exécution des trois 1res commandes n'a aucune incidence sur le résultat produit.

Résolution des symboles — exemple

– Exemple (2 / 2) –

```c
/* A.h */
#ifndef A_H
#define A_H
    int f(int x);
#endif
```

```c
/* A.c */
#include "A.h"
int f(int x) {
    return x * x;
}
```

```c
/* B.h */
#ifndef B_H
#define B_H
    int g(int x);
#endif
```

```c
/* B.c */
#include "B.h"
#include "A.h"
int g(int x) {
    return f(x);
}
```

```c
/* Main.c */
#include "B.h"
int main() {
    g(5);
    return 0;
}
```

Lors de la création de `B.o`, le compilateur **ignore** ce que fait le symbole `f`. Il sait seulement (grâce à l'inclusion de `A` dans `B`) que `f` est un symbole de fonction paramétrée par un entier et renvoyant un entier et peut donc **vérifier la correspondance des types**.

C'est au moment de l'édition des liens que le compilateur va **chercher l'implantation** du symbole `f` pour créer l'exécutable de la bonne manière.

Observation importante : la compilation d'un projet à plusieurs fichiers ne dépend pas de la manière dont ses modules sont inclus les uns dans les autres. Le schéma de compilation est toujours le même.

Plan

4. Compilation

Compilation séparée — intuition

Fait 1. : pour compiler un module `A`, il n'est pas nécessaire d'avoir les fichiers objets des autres modules du projet dont `A` ne dépend pas (de manière étendue ou non).

Conséquence : si un module `B` est modifié, il n'est nécessaire de recompiler que `B` et l'ensemble des modules qui dépendent (de manière étendue) à `B`.

Fait 2. : si `A` dépend de `B` et seul le fichier source de `B` a été modifié, il n'est pas nécessaire de recompiler `A`.

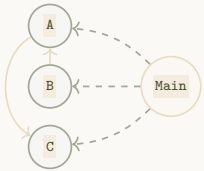
Conséquence : on ne recompile `A` que si `B.h` a été modifié.

Attention à ne pas oublier de recompiler le module principal `Main` si celui-ci dépend de manière étendue à des modules modifiés.

Compilation séparée — exemple introductif

– Exemple –

Considérons le projet suivant :



On le compile pour la 1re fois par

```
gcc ... -c -o Main.o Main.c
gcc ... -c -o A.o A.c
gcc ... -c -o B.o B.c
gcc ... -c -o C.o C.c
gcc -o Prgm Main.o A.o B.o C.o
```

Si on modifie par la suite le module `A`, il suffit d'exécuter les commandes pour mettre à jour l'exécutable du projet.

```
gcc ... -c -o A.o A.c
gcc ... -c -o B.o B.c
gcc ... -c -o Main.o Main.c
gcc -o Prgm Main.o A.o B.o C.o
```

Note : les 3 premières lignes commutent.

Il est inutile de recompiler le module `C` car il ne dépend pas (de manière étendue) au module `A`.

Schéma opérationnel de la compilation d'un projet

À l'appui de cette observation, la compilation d'un projet s'organise de la manière suivante.

(1) Pour chaque module `A` du projet :

  (a) compiler `A` si au moins l'une des conditions suivante est vérifiée :

  - `A.o` n'existe pas ;
  - `A.c` ou `A.h` ont été modifiés **après** `A.o` ;
  - il existe un module `B` dont `A` dépend (de manière étendue) et tel que `B.h` a été modifié **après** `A.o` ;

(2) si au moins un module du projet a été (re)compilé, reconstruire l'exécutable.

Nous allons utiliser l'utilitaire `make` et les fichiers `Makefile` pour rendre cette procédure automatique.

Plan

4. Compilation

Fichiers `Makefile`

L'utilitaire `make` est paramétré par un fichier dont le nom est imposé :

« `Makefile` » ou « `makefile` ».

Il doit se trouver dans le répertoire de travail (là où se trouvent les autres fichiers du projet ou au niveau des répertoires `include` et `src`).

Ce fichier fait partie intégrante du projet.

Tout fichier `Makefile` est constitué de règles. Elles sont de la forme

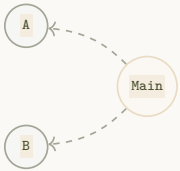
```
CIBLE: DEPENDANCES
→ COMMANDE
  ⋮
→ COMMANDE
```

Le symbole « → » désigne une tabulation.

Fichiers `Makefile` et fonctionnement

– Exemple –

Considérons le projet et le `Makefile` suivants.




```
Prog: A.o B.o Main.o
→ gcc -o Prog Main.o A.o B.o

Main.o: Main.c A.h B.h
→ gcc ... -c -o Main.o Main.c

A.o: A.c A.h
→ gcc ... -c -o A.o A.c

B.o: B.c B.h
→ gcc ... -c -o B.o B.c
```

Lorsque l'on exécute la commande `make`, `make` tente de résoudre la 1re règle (l. 1). Pour cela, il doit résoudre ses dépendances. Ensuite, il exécute les commandes de la règle si la cible n'est pas à jour.

Concrètement, pour pouvoir créer l'exécutable `Prog`, il est nécessaire que `A.o`, `B.o` et `Main.o` soient à jour. Une fois qu'ils le sont, il suffit de procéder à l'édition des liens (l. 2).

Fichiers `Makefile` et fonctionnement

Pour savoir si une cible est à jour, `make` regarde les dépendances de la règle et :

- si la dépendance est la cible d'une autre règle, `make` procède **récursivement** à sa résolution ;
- si la dépendance est le nom d'un fichier, `make` compare la **date de dernière modification** de la cible par rapport à celle du fichier.

S'il y a au moins un fichier dans les dépendances avec une date supérieure à celle de la cible, les commandes de la règle sont exécutées.

On peut imposer à `make` de commencer par la résolution de la règle dont la cible est `CIBLE` par

```
make CIBLE
```

Déclarations de types et dépendances

Soient `A` et `B` deux modules tels que `B` dépend (de manière étendue) à `A`, qu'un type `T` soit déclaré dans `A` et que `B` utilise ce type.

Toute modification de `A.h` doit être suivie d'une nouvelle compilation de `B`. En effet, si la déclaration de `T` a été modifiée, `B` doit être recompilé pour la prendre en compte.

En conséquence, dans le `Makefile` du projet doit figurer la règle

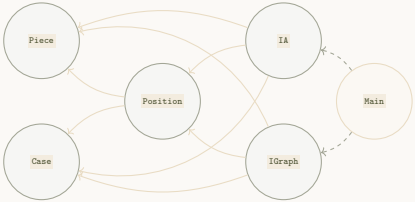
```
B.o: B.c B.h A.h
→ gcc ... -c -o B.o B.c
```

pour déclarer que la construction de `B.o` **dépend aussi** de `A.h`.

Attention : ceci ne s'applique pas aux modifications de l'implantation des fonctions de `A` dans `A.c` (comme nous l'avons déjà vu). La cible `B.o` ne dépend ainsi pas de `A.c`. Elle ne dépend que des **déclarations** du module `A` (et donc de `A.h`).

Exemple complet de `Makefile`

– Exemple –



Le `Makefile` du projet dont le graphe d'inclusions (étendues) est donné ci-contre est

```
Echecs: Main.o Piece.o Case.o Position.o IA.o IGraph.o
→ gcc -o Echecs Main.o Piece.o Case.o Position.o IA.o IGraph.o

Main.o: Main.c IA.h IGraph.h
→ gcc ... -c -o Main.o Main.c

Piece.o: Piece.c Piece.h
→ gcc ... -c -o Piece.o Piece.c
```

```
Case.o: Case.c Case.h
→ gcc ... -c -o Case.o Case.c

Position.o: Position.c Position.h Piece.h Case.h
→ gcc ... -c -o Position.o Position.c

IA.o: IA.c IA.h Piece.h Case.h Position.h
→ gcc ... -c -o IA.o IA.c

IGraph.o: IGraph.c IGraph.h Piece.h Case.h Position.h
→ gcc ... -c -o IGraph.o IGraph.c
```

Écriture de `Makefile` simples — résumé

Le `Makefile` d'un projet contenant des modules `A1`, …, `An` et un module principal `Main` est génériquement de la forme

```
NOM: Main.o A1.o ... An.o
→ gcc -o NOM Main.o A1.o ... An.o

Main.o: Main.c EXTRAmain
→ gcc -std=c17 -pedantic -Wall -c -o Main.o Main.c

A1.o: A1.c A1.h EXTRA1
→ gcc -std=c17 -pedantic -Wall -c -o A1.o A1.c

...

An.o: An.c An.h EXTRAn
→ gcc -std=c17 -pedantic -Wall -c -o An.o An.c
```

où `EXTRAmain` est la suite des noms des fichiers `.h` que `Main.c` inclut et pour tout $1 \leqslant k \leqslant n$, `EXTRAk` est la suite des noms des fichiers `.h` dont le module `Ak` dépend (de manière étendue).

Plan

4. Compilation

Variables dans les `Makefile`

Il est possible de **définir des variables** dans un `Makefile` par

```
ID=VAL
```

Ceci définit une variable identifiée par `ID`. Sa valeur est la **chaîne de caractères** `VAL`.

On accède à la valeur d'une variable identifiée par `ID` par

```
$(ID)
```

Ceci substitue à l'occurrence de `$(ID)` la chaîne de caractères qui lui a été attribuée lors de sa définition.

Variables dans les `Makefile` — exemple

Les variables permettent de factoriser les règles d'un `Makefile`.

– Exemple –

Voici un `Makefile` et sa version factorisée à l'aide des variables :

```
Main: Main.o A.o
→ gcc -o Main Main.o A.o

Main.o: Main.c
→ gcc -std=c17 -pedantic -Wall -c -o Main.o Main.c

A.o: A.c A.h
→ gcc -std=c17 -pedantic -Wall -c -o A.o A.c
```

```
CFLAGS=-std=c17 -pedantic -Wall

Main: Main.o A.o
→ gcc -o Main Main.o A.o

Main.o: Main.c
→ gcc $(CFLAGS) -c -o Main.o Main.c

A.o: A.c A.h
→ gcc $(CFLAGS) -c -o A.o A.c
```

Variables dans les `Makefile`

On utilise en général les **noms de variable usuels** suivants :

- `CFLAGS` pour les options de compilation (`CFLAGS=-std=c17 -pedantic -Wall`) ;
- `LDLIBS` pour l'inclusion de bibliothèques (`LDLIBS=-lm -lMLV`) ;
- `CC` pour le compilateur utilisé (`CC=gcc` ou bien `CC=clang`) ;
- `OPT` pour les option d'optimisation de code (`OPT=-O1` ou bien `OPT=-O2` ou encore `OPT=-O3`).

Règles courantes

Observation : la plupart des règles des `Makefile` sont sous l'une de ces deux formes :

1.
```
M.o: M.c DEP2... DEPn
→ gcc -c -o M.o M.c
```

2.
```
EXEC: DEP1 ... DEPn
→ gcc -o EXEC DEP1 ... DEPn
```

La 1re forme de règle a pour but de construire le fichier objet d'un module `M`. Dans ce cas, `DEP2`, …, `DEPn` sont les `.h` dont le module `M` dépend.

La 2e forme de règle a pour but de construire l'exécutable `EXEC` du projet. Les dépendances `DEP1`, …, `DEPn` sont dans ce cas les fichiers `.o` du projet.

Variables internes dans les `Makefile`

Il est possible de simplifier l'écriture de ces règles courantes au moyen des variables internes. Il y en a trois principales et deux secondaires :

Symbole	Ce qu'il désigne
`$@`	Nom de la cible
`$<`	Nom de la 1re dép.
`$^`	Noms de toutes les dép.
`$?`	Noms des dép. plus récentes que la cible
`$*`	Nom de la cible sans extension

– Exemples –

Sans variable interne.

```
M.o: M.c DEP2 ... DEPn
→ gcc -c -o M.o M.c
```

```
EXEC: DEP1 ... DEPn
→ gcc -o EXEC DEP1 ... DEPn
```

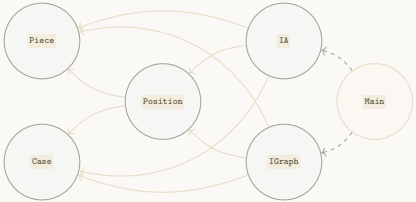
Avec variables internes.

```
M.o: M.c DEP2 ... DEPn
→ gcc -c -o $@ $<
```

```
EXEC: DEP1 ... DEPn
→ gcc -o $@ $^
```

Variables internes dans les `Makefile` — exemple

– Exemple –



L'utilisation des variables et variables internes permet de simplifier le `Makefile` de ce projet vu précédemment.

```
CC=gcc
CFLAGS=-std=c17 -pedantic -Wall
OBJ=Main.o Piece.o Case.o Position.o IA.o IGraph.o

Echecs: $(OBJ)
→ $(CC) -o $@ $^

Main.o: Main.c IA.h IGraph.h
→ $(CC) $(CFLAGS) -c -o $@ $<

Piece.o: Piece.c Piece.h
→ $(CC) $(CFLAGS) -c -o $@ $<
```

```
Case.o: Case.c Case.h
→ $(CC) $(CFLAGS) -c -o $@ $<

Position.o: Position.c Position.h Piece.h Case.h
→ $(CC) $(CFLAGS) -c -o $@ $<

IA.o: IA.c IA.h Piece.h Case.h Position.h
→ $(CC) $(CFLAGS) -c -o $@ $<

IGraph.o: IGraph.c IGraph.h Piece.h Case.h Position.h
→ $(CC) $(CFLAGS) -c -o $@ $<
```

Règles génériques

Il est possible de simplifier encore d'avantage l'écriture des `Makefile` au moyen des règles génériques.

Ce sont des règles de la forme

```
%.o: %.c
→ COMMANDE
```

où `COMMANDE` est une commande.

Cette syntaxe simule une règle

```
M.o: M.c
→ COMMANDE
```

pour chaque fichier `M.c` du projet.

Intérêt principal : l'**unique règle**

```
%.o: %.c
→ gcc $(CFLAGS) -c -o $@ $<
```

permet de **construire** chaque **fichier objet** du projet.

Règles génériques

Attention : la règle

```
%.o: %.c
→ gcc $(CFLAGS) -c -o $@ $<
```

ne prend pas en compte des dépendances des modules aux fichiers `.h` concernés.

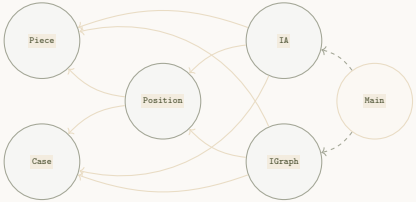
Il faut les **mentionner explicitement** de la manière suivante :

```
M.o: M.c M.h DEP1 ... DEPn
```

pour chaque module `M` du projet. `DEP1`, …, `DEPn` sont les `.h` dont le module `M` dépend.

Règles génériques — exemple

– Exemple –



L'utilisation des règles génériques permet de simplifier encore les `Makefile`.

```
CC=gcc
CFLAGS=-std=c17 -pedantic -Wall
OBJ=Main.o Piece.o Case.o Position.o IA.o IGraph.o

Echecs: $(OBJ)
→ $(CC) -o $@ $^

Main.o: Main.c IA.h IGraph.h

Piece.o: Piece.c Piece.h
```

```
Case.o: Case.c Case.h

Position.o: Position.c Position.h Piece.h Case.h

IA.o: IA.c IA.h Piece.h Case.h Position.h

IGraph.o: IGraph.c IGraph.h Piece.h Case.h Position.h

%.o: %.c
→ $(CC) $(CFLAGS) -c -o $@ $<
```

Règles de nettoyage

Règle de nettoyage :

```
clean:
→ rm -f *.o
```

Cette règle permet, lorsque l'on saisit la commande `make clean`, de supprimer les fichiers `.o` du répertoire courant.

Règle de nettoyage total :

```
mrproper: clean
→ rm -f EXEC
```

Cette règle, où `EXEC` est le nom de l'exécutable du projet, permet de supprimer tous les fichiers regénérables (c.-à-d. les fichiers `.o` et l'exécutable) à partir des fichiers `.c` et `.h` du projet.

Règles d'installation / désinstallation

Règle d'installation :

```
PREFIX = $(PWD) # dossier courant

install: EXEC
→ mkdir -p $(PREFIX)/bin
→ cp EXEC $(PREFIX)/bin/EXEC
```

Cette règle permet de compiler le projet, de placer son exécutable `EXEC` dans un répertoire `$(PREFIX)/bin`, où `PREFIX` est une variable contenant le dossier de base où installer notre programme.

Règle de désinstallation :

```
uninstall: mrproper
→ rm -f $(PREFIX)/bin/EXEC
→ rmdir --ignore-fail-on-non-empty $(PREFIX)/bin
```

Cette règle permet de supprimer les fichiers regénérables, l'exécutable du projet, ainsi que le répertoire `bin` le contenant s'il est vide.

Règle de distribution

Création d'archive :

```
ARCHIVE = mon-logiciel-v20240214

dist: clean
→ mkdir -p $(ARCHIVE)
→ cp -R Makefile Readme Main.c Piece.c Piece.h \
→ → Case.c Case.h Position.c etc $(ARCHIVE)
→ tar -czf $(ARCHIVE).tar.gz $(ARCHIVE)
→ rm -rf $(ARCHIVE)
```

Pour créer une archive propre contenant les sources d'un programme, on peut également utiliser une règle de **dist**ribution. Les fichiers à placer dans l'archive y sont listés et sont copiés dans un dossier qui servira de base pour la création de l'archive.

Plan

4. Compilation

Principes généraux

Une bibliothèque est un **regroupement de modules** offrant des fonctionnalités allant vers un même objectif.

– Exemples –

- La bibliothèque graphique MLV est un ensemble de plusieurs modules (`MLV_audio`, `MLV_keyboard`, `MLV_image`, *etc.*) réunis dans le but d'offrir des fonctions d'affichage graphique et de gérer les entrées / sorties (son, clavier, souris, *etc.*).
- La bibliothèque NCURSES offre divers outils pour concevoir des applications en mode texte.

L'intérêt des bibliothèques est double :

1. il suffit de réaliser une inclusion d'**un seul en-tête** pour bénéficier des fonctionnalités d'une bibliothèque, plutôt que des inclusions de plusieurs en-têtes sans être sûr du fichier précis à inclure ;
2. sous réserve de savoir comment créer des bibliothèques, il est possible de **partager et réutiliser** son propre code entre plusieurs de ses projets, sans avoir à le recompiler.

Bibliothèques statiques

Une bibliothèque statique est un fichier d'extension `.a`.

Lors de son utilisation, son **code est inclus dans l'exécutable** pendant l'édition des liens.

- **Avantage** : tout projet qui utilise une bibliothèque statique peut être exécuté sur une machine où la bibliothèque n'est pas installée.
- **Inconvénient** : l'exécutable est plus volumineux.

Bibliothèques dynamiques

Une bibliothèque dynamique est un fichier d'extension `.so`.

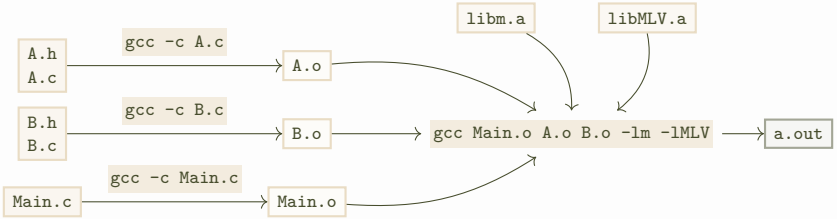
Lors de son utilisation, son code n'est pas inclus dans l'exécutable. C'est lors de l'exécution que les symboles provenant de la bibliothèque sont résolus au moyen de l'**éditeur de liens dynamique**.

- **Avantages** : l'exécutable est moins volumineux. Il n'y a pas de duplication du code de la bibliothèque sur un même système si plusieurs projets l'utilisent.
- **Inconvénient** : tout projet qui utilise une bibliothèque dynamique ne peut être exécuté que sur un système où cette dernière est installée.

Compilation d'un projet utilisant des bibliothèques

On suppose que l'on travaille sur un projet constitué de deux modules `A` et `B` et d'un fichier principal `Main.c`. Ce projet utilise deux bibliothèques statiques : `libm.a` et `libMLV.a`.

La compilation de ce projet se réalise de manière habituelle. La différence porte sur l'étape d'**édition des liens** dans laquelle il est nécessaire de **signaler les bibliothèques utilisées**.



Pour utiliser des bibliothèques qui ne sont pas situées dans le répertoire standard des bibliothèques, on précisera leur chemin `chem` lors de l'édition des liens au moyen de l'option `-Lchem`.

Compilation d'un projet utilisant des bibliothèques

Le nom d'une bibliothèque commence par `lib`. On signale l'utilisation d'une bibliothèque à l'éditeur de liens par `-lNOM` où `libNOM` est le nom de la bibliothèque.

– Exemple –

En supposant que le graphe d'inclusions (étendues) du projet précédent est celui ci-contre, un `Makefile` possible est


```
CC=gcc
CFLAGS=-std=c17 -pedantic -Wall
LDLIBS=-lm -lncurses
OBJ=Main.o A.o B.o

Projet: $(OBJ)
→ $(CC) -o $@ $^ $(LDLIBS)

Main.o: Main.c A.h B.h
A.o: A.c A.h
B.o: B.c B.h A.h

%.o: %.c
→ $(CC) $(CFLAGS) -c -o $@ $<
```

```
clean:
→ rm -f *.o

mrproper: clean
→ rm -f Projet

PREFIX = $(PWD) # dossier courant
install: EXEC
→ mkdir -p $(PREFIX)/bin
→ cp Projet $(PREFIX)/bin/Projet

uninstall: mrproper
→ rm -f $(PREFIX)/bin/Projet
→ rmdir --ignore-fail-on-non-empty $(PREFIX)/bin
```

La bibliothèque standard

La bibliothèque standard `libc.a` (`libc.so`) regroupe les vingt-quatre modules

`assert`,	`complex`,	`ctype`,	`errno`,
`fenv`,	`float`,	`inttypes`,	`iso646`,
`limits`,	`locale`,	`math`,	`setjmp`,
`signal`,	`stdarg`,	`stdbool`,	`stddef`,
`stdint`,	`stdio`,	`stdlib`,	`string`,
`tgmath`,	`time`,	`wchar`,	`wctype`.

Cette bibliothèque est **liée implicitement** lors de toute édition des liens.

Il est donc possible d'utiliser certains des modules de la bibliothèque standard simplement en les incluant (`#include <NOM.h>`), sans avoir à utiliser l'option `-lc`.

L'utilisation de certains modules doit cependant s'accompagner de l'option `-lX` (comme `-lm` pour utiliser `math.h`).

Création de bibliothèques statiques

Pour **créer une bibliothèque statique** `X`, on suit les étapes suivantes :

1. **écriture des modules** `M1`, …, `Mn` qui vont constituer la bibliothèque ;
2. **compilation des modules** et création des fichiers objets `M1.o`, …, `Mn.o` ;
3. **création de l'archive** `libX.a` par

```
ar r libX.a M1.o ... Mn.o
```

4. **génération de l'index** de la bibliothèque par

```
ranlib libX.a
```

5. (étape facultative) **écriture d'un fichier d'en-tête global**

```
/* X.h */
#ifndef X_H
#define X_H

#include "M1.h"
...
#include "Mn.h"

#endif
```

Création de bibliothèques statiques — exemple

– Exemple (début) –

On suppose que l'on a créé trois modules :

1. `Liste` pour la représentation des listes chaînées ;
2. `Arbre` pour la représentation des arbres binaires de recherche ;
3. `Tri` pour l'implantation d'algorithmes de tris de tableaux.

On souhaite regrouper ces modules en une bibliothèque nommée `algo`.

Celle-ci sera constituée de deux fichiers ;

1. `libalgo.a`, l'implantation de la bibliothèque ;
2. `Algo.h`, l'en-tête de la bibliothèque.

Création de bibliothèques statiques — exemple

– Exemple (suite et fin) –

Pour créer de la bibliothèque `algo`, on saisit les commandes

```
gcc -c Liste.c ; gcc -c Arbre.c ; gcc -c Tri.c
ar r libalgo.a Liste.o Arbre.o Tri.o
ranlib libalgo.a
```

et on écrit l'en-tête global

```c
/* Algo.h */
#ifndef ALGO_H
#define ALGO_H

#include "Liste.h"
#include "Arbre.h"
#include "Tri.h"

#endif
```

Pour utiliser la bibliothèque `algo` dans un fichier `F` d'un projet `P`, il faut inclure dans `F` son en-tête, réaliser l'édition des liens de `P` avec l'option `-lalgo` et indiquer son chemin `chem` avec l'option `-Lchem`.

Index

Il est possible de **consulter l'index** d'une bibliothèque statique `LIB` par la commande `nm -s libLIB.a`

– Exemple –

```c
/* A.h */
#ifndef A_H
#define A_H
    char h(int a);
#endif
```

```c
/* A.c */
#include "A.h"
char h(int a) {
    return a % 256;
}
```

```c
/* B.h */
#ifndef B_H
#define B_H
    typedef int S;

    int f(S s);
    char g(int a);
#endif
```

```c
/* B.c */
#include "B.h"
int f(S s) {
    return s;
}
char g(int a) {
    return a % 64;
}
```

Création :

```
gcc -c A.c ; gcc -c B.c
ar r libAB.a A.o B.o
ranlib libAB.a
```

Affichage :

```
nm -s libAB.a
```

```
Archive index:
h in A.o
f in B.o
g in B.o

A.o:
0000000000000000 T h

B.o:
0000000000000000 T f
000000000000000c T g
```

Résumé de l'axe 1

Nous avons étudié plusieurs outils et posé des conventions pour le développement de projets.

1. Au niveau de la **présentation du code** :
   - indentation ;
   - choix des identificateurs ;
   - documentation.
2. Au niveau de l'**écriture de fonctions** :
   - pré-assertions ;
   - mécanismes de gestion d'erreurs.
3. Au niveau de la **conception de projets** :
   - analyse d'une spécification ;
   - découpage en modules ;
   - compilation séparée.

Tout ce qui a été étudié dans cet axe est à appliquer dans la pratique.

Axe 2 : comprendre les mécanismes de base

5. Allocation dynamique

6. Entrées et sorties

7. Types

8. Types structurés

5. Allocation dynamique

5. Allocation dynamique

La mémoire

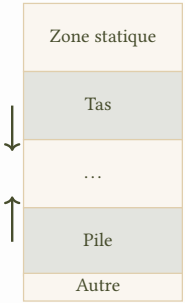
Lors de l'exécution d'un programme, une portion (de taille variable) de la mémoire lui est dédiée. Cette zone lui est exclusive. On l'appelle la mémoire.

On peut voir la mémoire comme un **très grand tableau d'octets**.

La mémoire est segmentée en plusieurs parties :

- la zone statique qui contient le code et les données statiques ;
- le tas, de taille variable au fil de l'exécution ;
- la pile, de taille variable au fil de l'exécution.

Il y a d'autres zones (non repr. ici).



Le **tas** contient les objets **alloués dynamiquement**.

La **pile** contient les objets des **variables locales** lors des appels aux fonctions.

Pointeurs

Nous avons vu que chaque variable possède une **adresse**.

Connaître l'adresse d'une variable est **suffisant pour la manipuler** (c.-à-d., lire et modifier sa valeur).

L'objet qui permet de représenter, de stocker et de manipuler des adresses est le pointeur.

Un pointeur est une entité constituée des deux éléments suivants :

1. une adresse vers une zone de la mémoire (la valeur du pointeur) ;
2. le type de la donnée se trouvant à cette adresse.

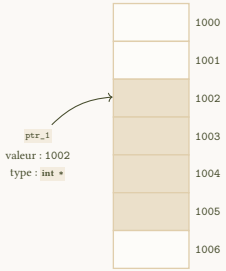
Pointeurs

Intuitivement, un pointeur est une **flèche** munie d'un mode d'emploi, le tout pointant vers une zone de la mémoire.

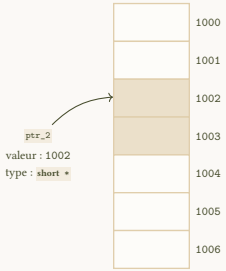
Le **mode d'emploi** décrit le type de la zone de la mémoire ainsi adressée en renseignant sur la **taille de la zone**.

– Exemples –

Si `ptr_1` est un pointeur sur une donnée de type **`int`** (4 octets) :



Si `ptr_2` est un pointeur sur une donnée de type **`short`** (2 octets) :



Déclaration de pointeurs

On **déclare** un pointeur sur une zone de la mémoire de type `T` par

```
T *ptr;
```

On **accède à la valeur** de la zone mémoire pointée par `ptr` par

```
*ptr
```

On **accède à l'adresse** de la zone mémoire pointée par `ptr` par

```
ptr
```

Attention : le même symbole `*` est utilisé pour la déclaration d'un pointeur et pour accéder à la valeur pointée. C'est une imperfection du langage C qui peut porter à confusion.

Astuce : la déclaration d'un pointeur `T *ptr;` peut se lire `T (*ptr);` c-à-d `*ptr` est de type `T`.

Manipulation de pointeurs

On suppose que `ptr` est un pointeur pointant sur une zone de la mémoire de type `T`.

Il est possible de **changer l'endroit** où `ptr` pointe par

```
ptr = ADR;
```

où `ADR` est une adresse de la mémoire de type `T`.

Il est possible d'**affecter une valeur** `VAL` de type `T` à `*ptr` par

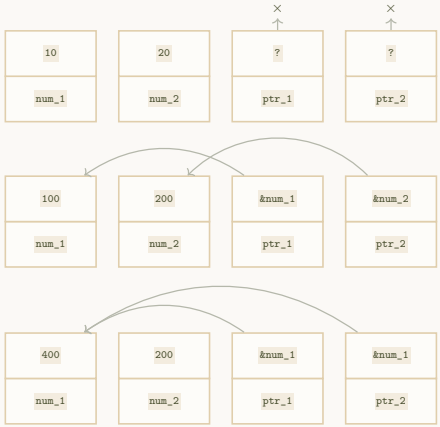
```
*ptr = VAL;
```

De cette manière, la zone mémoire d'adresse `ptr` est modifiée et devient de valeur `VAL`.

Manipulation de pointeurs — exemple

– Exemple –

```c
/* (1) */
int num1, num2;
int *ptr1, *ptr2;
num1 = 10;
num2 = 20;


/* (2) */
ptr1 = &num1;
ptr2 = &num2;
*ptr1 = 100;
*ptr2 = 200;


/* (3) */
ptr2 = ptr1;
*ptr1 = 300;
*ptr2 = 400;
```

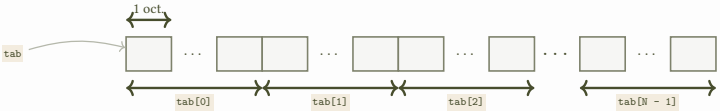


Pointeurs et tableaux statiques

Un tableau est une zone de la mémoire contenant une séquence **contigüe** d'éléments du même type. Le 1er élément occupe le début de la zone, et les autres éléments du tableau sont à des adresses plus élevées.

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

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



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

```
tab[i]
```

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

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

Pointeurs et tableaux statiques

L'expression `tab` s'évalue en **l'adresse du premier élément du tableau** (`&tab[0]`) et est donc de type `T*`. Sachant que ses éléments **sont contigus en mémoire**, la syntaxe

```
tab[i]
```

est équivalente à

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

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

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

Pointeurs et tableaux statiques

– Exemple –

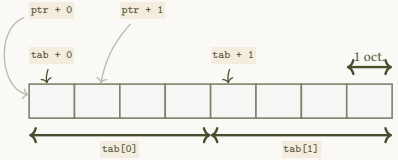
Considérons les instructions suivantes :

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

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

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

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

Leur exécution affiche

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

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



5. Allocation dynamique

Passage par valeur — rappel

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

– Exemple –

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

Ces instructions affichent `5`.

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

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

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

qui est considéré. La fonction travaille en effet avec la valeur **issue de l'évaluation** de l'expression `2 * num + 1` qui est recopiée dans la variable locale (paramètre) `nb`. Il est alors clair que cette valeur de cette expression ne peut pas être modifiée après évaluation.

Passage par adresse

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

On parle alors de passage par adresse. Il s'agit toujours d'un passage par valeur, mais dont la valeur est une adresse.

– Exemple –

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

Ces instructions affichent 6.

En ligne 6, c'est (la valeur de) l'**adresse** de `num` qui est transmise.

Celle-ci est **universelle** (visible partout).

Qualificateur `const`

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

– Exemple –

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

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

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

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

Le mot-clé **`const`** est un qualificateur.

Qualificateur `const`

Ainsi, plus généralement,

```
const T *ID
```

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

– Exemples –

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

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

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

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

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

Qualificateur `const`

Le qualificateur `const` permet quelques subtilités :

```
T *const ID
```

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

```
const T *const ID
```

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

– Exemple –

Voici les quatre cas de figure :

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

    int e;

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

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

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

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

Plan

5. Allocation dynamique

Données persistantes en mémoire

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

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

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

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

Pour allouer une portion du tas, on utilise la fonction

```
#include <stdlib.h>

void *malloc(int size);
```

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

Utilisation de `malloc`

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

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

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

Explications :

- `(T *)` sert à préciser que la zone de la mémoire à allouer est de type `T`. Cette précision, qui est une coercition, est optionnelle car `malloc` renvoie un pointeur sur une zone non typée (`void *`) dont la coercition est implicite ;
- l'argument `sizeof(T) * N` permet de demander à allouer une zone mémoire de taille `sizeof(T) * N` octets. Celle-ci pourra donc accueillir `N` valeurs de type `T`. (on peut aussi noter `sizeof(T[N])`).

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

Tests d'erreurs et `malloc`

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

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

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

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

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

`ACTION` peut être un `exit(EXIT_FAILURE);` si l'on se trouve dans le `main` ou un `return NULL;` voire un `return 0;` si l'on se trouve dans une fonction avec gestion d'erreurs.

Libération de mémoire

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

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

On l'utilise de la manière suivante :

```
free(ptr);
ptr = NULL; /* Non obligatoire. */
```

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

– Exemple –

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

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

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

La fonction `calloc`

La fonction

```
void *calloc(int length, int size);
```

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

– Exemple –

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

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

Plan

5. Allocation dynamique

Tableaux dynamiques

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

– Exemple –

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

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

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

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

```
free(tab);
```

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

Création d'un tableau dynamique à deux dimensions

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

– Exemple –

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

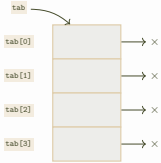
```
int **tab;
tab = (int **) malloc(sizeof(int *) * 4);
if (tab == NULL) exit(EXIT_FAILURE);
for (int i = 0 ; i < 4 ; ++i) {
    tab[i] = (int *) malloc(sizeof(int) * 3);
    if (tab[i] == NULL) exit(EXIT_FAILURE);
    for (int j = 0 ; j < 3 ; ++j)
        tab[i][j] = 0;
}
```

Cet exemple est à bien connaître et à généraliser aux dimensions supérieures.

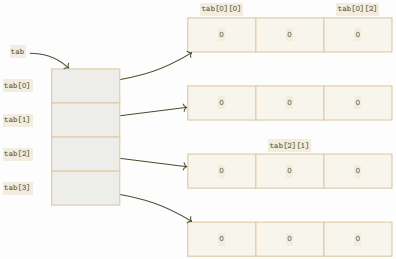
Important : ici, des `exit(EXIT_FAILURE)` sont placés pour quitter l'exécution en cas de problème. Ceci est correct uniquement si ces instructions sont dans le `main`. Si cette création est dans une fonction, un code d'erreur sera renvoyé sans interrompre l'exécution, tout en libérant la mémoire déjà allouée.

Création d'un tableau dynamique à deux dimensions

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



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



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

Destruction d'un tableau dynamique à deux dimensions

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

– Exemple –

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

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

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

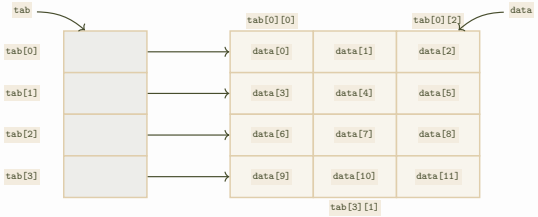
Variation de tableau dynamique à deux dimensions

Il est également possible de ne faire qu'une allocation par dimension (et donc autant de libérations).

– Exemple –

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

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



La fonction `realloc`

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

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

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

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

– Exemple –

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

affiche

```
0x1205010
0x7fb123f9d010
```

Exemple d'utilisation de `realloc`

– Exemple –

Ceci construit une chaîne de caractères, caractère par caractère, dans un tableau dynamique dont la **taille est adaptative**. La lecture s'arrête sur lecture de `'!'`.

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

Il faut ajouter la bonne gestion des erreurs en vérifiant ce qui est renvoyé par le `malloc` et les `realloc`, ainsi que par ce qui est renvoyé par le `scanf`.

6. Entrées et sorties

Plan

6. Entrées et sorties

Écriture formatée

La fonction

```
int printf(char *format, V_1, ..., V_N);
```

permet de réaliser une écriture formatée sur la sortie standard `stdout`.

Rappel : cette fonction renvoie le **nombre de caractères affichés**. Si une erreur a lieu, elle renvoie un entier négatif.

– Exemple –

```
int num;
num = printf("%s+%d\n", "test", 200);
printf("%d\n", num);
```

Ces instructions affichent

```
test+200
9
```

Indicateurs de conversion

On utilise les indicateurs de conversion suivants pour interpréter chaque valeur à écrire (ou à lire, voir la partie suivante) de manière adéquate :

Indicateur de conversion	Affichage
`d`, `i`	Entier en base dix
`u`	Entier non signé en base dix
`x`, `X`	Entier en hexadécimal
`c`	Caractère
`e`, `E`	Flottant en notation scientifique
`f`, `g`	Flottant
`s`	Chaîne de caractères
`p`	Pointeur

Caractères spéciaux

Certains caractères **ne s'affichent pas** mais produisent un **effet** sur la sortie. Ce sont des caractères spéciaux.

Caractère spécial	Rôle
`\n`	Passage à la ligne suivante
`\b`	Retour en arrière d'un caractère
`\f`	Passage à la ligne suivante avec alinéa
`\r`	Retour chariot
`\t`	Tabulation horizontale
`\v`	Tabulation verticale

Caractères de largeur

Il est possible spécifier la largeur de l'affichage.

Caractère de largeur	Rôle
`N`	Affichage sur au moins `N` caractères (ajout d'espaces)
`ON`	Affichage du nombre sur au moins `N` caractères (ajout de « `0` »)
`*`	Utilise un argument supplémentaire pour spécifier la largeur

– Exemples –

`printf("%5s%04d\n", "Ab", 72);`
affiche
`␣␣␣Ab0072`

`printf("%*s\n", 5, "Ef");`
affiche
`␣␣␣Ef`.

Caractères d'attribut

Il est possible de faire suivre le `%` d'un indicateur de conversion de caractères d'attribut pour réaliser un formatage avancé.

Caractère d'attribut	Rôle
`+`	Force l'affichage du signe d'un nombre
`␣`	Affichage une espace en plus si positif
`-`	Justifie à gauche un nombre (à droite par défaut)

– Exemples –

`printf("%+5d\n", 23);`
affiche
`␣␣+23`

`printf("%+-5d %d\n", 23, 5);`
affiche
`+23␣␣␣5`.

Écriture caractère par caractère

La fonction

```
int putchar(int c);
```

permet d'**afficher un caractère** sur la sortie standard.

Cette fonction renvoie le caractère écrit (converti en un `int`). Elle renvoie la constante `EOF` (end-of-file) si une erreur a lieu.

– Exemple –

```
int ret;
ret = putchar('a');
if (ret == EOF)
    /* Gestion de l'erreur. */
```

Ces instructions affichent `a`. Un test est réalisé pour détecter une erreur éventuelle.

Plan

6. Entrées et sorties

Lecture formatée

La fonction

```
int scanf (char *format, PTR_1, ..., PTR_N);
```

permet de réaliser une lecture formatée sur l'entrée standard `stdin`.

Cette fonction renvoie le **nombre** d'éléments lus **correctement assignés**.

– Exemples –

```
int num, ret;
char chaine[128];

chaine[0] = '\0';
ret = scanf("%d W %s", &num, chaine);
printf("%d %d %s\n", ret, num, chaine);
```

Ces instructions lisent sur l'entrée standard un entier, un caractère `'W'`, puis une chaîne de caractères.

1. `25␣W␣abc\n` → `2 25 abc`
2. `25␣W␣␣abc\n` → `2 25 abc`
3. `25Wabc\n` → `2 25 abc`
4. `25␣abc\n` → `1 25`
5. `xy␣W␣abc\n` → `0 ?`
6. `25␣w␣abc\n` → `1 25`

Exemple d'utilisation de `scanf`

– Exemple –

Ce programme, lorsque exécuté, lit sur l'entrée standard une chaîne d'au plus sept caractères (format `%Ns`), une espace, puis un entier.

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

int main() {
    char prenom[8];
    int age, ret;

    ret = scanf("%7s %d", prenom, &age);
    while (ret != 2) {
        printf("%lu\n", strlen(prenom));
        ret = scanf("%7s %d", prenom, &age);
    }
    return 0;
}
```

Bien noter que `prenom` est un tableau statique de taille `8 = 7 + 1` pour avoir assez de place pour accueillir le caractère de fin de chaîne `'\0'`.

Lecture caractère par caractère

La fonction

```
int getchar();
```

permet de **lire un caractère** sur l'entrée standard.

Cette fonction renvoie le caractère lu (converti en un `int`). Elle renvoie la constante `EOF` (end-of-file) lorsque la fin de fichier est détectée (touche `Ctrl + D`).

– Exemple –

```
int c;
while ((c = getchar()) != EOF)
    printf("%c", c);
```

Ces instructions affichent chaque caractère lu en entrée, tant que `EOF` n'est pas rencontré.

Le tampon d'entrée

Considérons les instructions

```
int c;
while ((c = getchar()) != EOF)
    printf("%c", c);
```

Il est possible de saisir plusieurs caractères avant le `'\n'` (touche `Entrée`).

Avant d'être lus par le `getchar` de la boucle, ils sont situés temporairement dans le tampon d'entrée.

Le tampon d'entrée se comporte comme un **tableau de caractères**.

Chaque appel à `getchar` considère le tampon d'entrée et

- s'il est vide, l'exécution est mise en pause jusqu'à la saisie d'au moins un caractère sur l'entrée standard suivie d'`Entrée` ;
- s'il contient au moins un élément, ce caractère est supprimé du tampon d'entrée et renvoyé, **sans mettre en pause l'exécution**.

Plan

6. Entrées et sorties

Descripteurs de fichiers et tête de lecture/écriture

Tout fichier est manipulé par un pointeur sur une variable de type `FILE`.

C'est un type structuré déclaré dans `stdio.h` qui contient diverses informations nécessaires et relatives au fichier qu'il adresse.

Toute variable de type `FILE *` est un descripteur de fichier.

La lecture/écriture dans un fichier se fait par l'intermédiaire d'une tête de lecture.

Celle-ci désigne un caractère du fichier considéré comme le **caractère observé** à un instant donné.

Les fonctions de lecture/écriture ont pour effet (en particulier) de mettre à jour cette tête de lecture en avançant dans son positionnement dans le fichier.

Tête de lecture/écriture

Il existe trois fonctions de `stdio.h` pour **déplacer manuellement la tête de lecture** ou obtenir des informations à son sujet :

- `int ftell(FILE *f);` qui renvoie l'indice de la tête de lecture du fichier pointé par `f`.

  Cette fonction est sans effet de bord ;

- `int fseek(FILE *f, int decalage, int mode);` qui décale la tête de lecture du fichier pointé par `f` de `decalage` caractères selon le mode `mode`.

  Ce paramètre explique à quoi le décalage est relatif (`SEEK_SET` : début du fichier, `SEEK_CUR` : position courante, `SEEK_END` : fin du fichier).

  Cette fonction renvoie `0` si elle s'est bien exécutée et `-1` sinon ;

- `void rewind(FILE *f)` ; place la tête de lecture au début du fichier. L'appel `rewind(f)` est ainsi équivalent à `fseek(f, 0, SEEK_SET)`.

Ouverture de fichiers

La fonction

```
FILE *fopen(const char *chemin, const char *mode);
```

permet d'**ouvrir un fichier**.

Cette fonction renvoie un descripteur de fichier sur le fichier de chemin `chemin` (relatif par rapport à l'exécutable).

La tête de lecture est positionnée sur son 1er caractère.

Le paramètre `mode` désigne le **mode d'ouverture** désiré.

Attention : `fopen` renvoie `NULL` si l'ouverture s'est mal passée. Il faut donc toujours tester la valeur de retour de `fopen`.

Modes d'ouverture

Il existe plusieurs modes d'ouverture. Chacun répond à un besoin particulier :

- `"r"` : lecture seule.
- `"w"` : écriture seule. Si le fichier n'existe pas, il est créé.
- `"a"` : écriture en ajout. Permet d'écrire dans le fichier en partant de la fin. Si le fichier n'existe pas, il est créé.
- `"r+"` : lecture et écriture.
- `"w+"` : lecture et écriture avec suppression préalable du contenu du fichier. Si le fichier n'existe pas, il est créé.
- `"a+"` : lecture et écriture 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.

Fermeture de fichiers

La fonction

```
int fclose(FILE *f);
```

permet de **fermer un fichier**.

Cette fonction permet de mettre à jour le fichier pointé par `f` de sorte que toutes les modifications effectuées soient prisent en compte. Le pointeur `f` n'est alors plus utilisable pour accéder au fichier.

Cette fonction renvoie `0` si la fermeture s'est bien déroulée et `EOF` dans le cas contraire.

Attention : toute ouverture d'un fichier lors de l'exécution d'un programme doit s'accompagner tôt ou tard de la fermeture future du fichier en question.

Écriture dans un fichier

La fonction

```
int fprintf(FILE *f, char *format, V_1, ..., V_N);
```

permet de réaliser une **écriture formatée** dans le fichier pointé par `f`.

Cette fonction se comporte comme `printf`, à la différence que cette dernière travaille sur le fichier `stdout`.

– Exemple –

```
FILE *f;
int ret;
f = fopen("fic.txt", "w");
if (f == NULL)
    /* Gestion de l'erreur. */
fprintf(f, "abc\n");
ret = fclose(f);
if (ret == EOF)
    /* Gestion de l'erreur. */
```

Ces instructions déclarent un pointeur sur le fichier `fic.txt` et écrivent `"abc\n"` dans `fic.txt`.

Lecture dans un fichier

La fonction

```
int fscanf(FILE *f, char *format, V_1, ..., V_N);
```

permet de réaliser une **lecture formatée** depuis le fichier pointé par `f`.

Cette fonction se comporte comme `scanf`, à la différence que cette dernière travaille sur le fichier `stdin`.

6. Entrées et sorties

Fichiers binaires — introduction

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

– 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).

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**.

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 –

```c
#include <stdio.h>

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

```c
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). Cependant, un format est en pratique souvent utilisé pour assurer la portabilité du fichier.

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 –

```c
#include <stdio.h>
int main() {
    FILE *f;
    int tab_1[12], tab_2[12];

    for (int 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).

7. Types

Plan

7. Types

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**.

L'opérateur `sizeof` permet de connaître la taille en octets d'un type. Il est aussi possible de l'appliquer à une valeur ou à une variable, mais nous **interdisons ces usages**.

– Exemple –

`sizeof(int)` et `sizeof(33)` valent `4`.

```
int t;
char c;

printf("%d ", sizeof(int));
printf("%d ", sizeof(t)); /* Interdit. */
printf("%d ", sizeof(31)); /* Interdit. */

printf("%d ", sizeof(char));
printf("%d ", sizeof(c)); /* Interdit. */
printf("%d\n", sizeof('a')); /* Interdit. */
```

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

7. Types

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.

Quelques avantages de ce procédé :

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

Attention : un entier non signé **ne peut jamais devenir négatif**. Ceci peut poser problème pour les conditions de sortie au sein des boucles.

– Exemple –

```
unsigned int i;

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

Ces instructions 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`, …
- 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'` a pour valeur son code ASCII qui est `49` et ne vaut donc 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

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

– Exemples –

```
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`.

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

Un phénomène similaire apparait lors de l'addition du flottant `0.1` avec le flottant `0.2` : ce n'est pas exactement le nombre `0.3` qui est obtenu.

Danger des types flottant : une solution partielle

Les types flottants présentent divers désavantages par rapport aux types entiers :

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

Pour ces raisons, il est recommandé d'utiliser des types flottants que si cela est vraiment nécessaire.

Une alternative, quand cela est possible, consister à représenter 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.

– Exemple –

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 données 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 1 à la valeur des variables sur lesquels ils sont appliqués.

Plan

7. Types

Types structurés

La syntaxe

```
struct ÉTIQUETTE {
    TYPE_1 CHAMP_1;
    TYPE_2 CHAMP_2;
    ...
};
```

permet de **déclarer un type structuré** `struct ÉTIQUETTE`, constitué des champs `CHAMP_1`, `CHAMP_2`, *etc.*

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

– Exemple –

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

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

`struct Couple couple;` déclare une variable de ce type structuré.

Types synonymes

Afin d'éviter l'écriture du redondant **struct**, on peut utiliser un **type synonyme**.

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

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

↑ L'étiquette `ÉTIQUETTE` est facultative, on parle alors de structure **anonyme**.

– Exemple –

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

déclare un type synonyme `Couple` d'un type structuré **anonyme**.

`Couple couple;` déclare une variable de ce type structuré.

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
```

– Exemple –

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.

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

– Exemple –

```
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`, *etc.*

Attention, on utilise des `,` et non pas des `;`.

Une variable de ce type peut prendre pour valeur exactement un des énumérateurs qui le constituent.

– Exemple –

```
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

– Exemple –

```
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.

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

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.

– Exemple –

```
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. Le compilateur peut fournir aider à ne pas oublier des cas.

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**.

– Exemples –

- `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`.

8. Types structurés

Plan

8. Types structurés

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.

– Exemple –

```
typedef struct {
    char c;
    int a;
    double b;
} Triplet;
...
Triplet tr1 = {'h', 55, 214.35};
// ou
Triplet tr2 = { .c = 'h', .a = 55, .b = 214.35};
```

Déclare, en l'initialisant, la variable `tr1`.

`'h'`	`55`	`214.35`
`c` : **char**	`a` : **int**	`b` : **double**
`tr1` : `Triplet`		

Déclaration de types structurés récursifs

Il est possible de **déclarer des types structurés récursifs** en faisant usage de l'étiquette et du mot clé `struct` (ou en créant le type synonyme avant la définition du type structuré) :

```
typedef struct liste {
    int e;
    struct liste *s;
} Liste;
```

```
typedef struct liste Liste;
struct liste {
    int e;
    Liste *s;
};
```

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

```
typedef struct liste {
    int e;
    struct liste s;
} Liste;
```

Attention, cette déclaration n'est pas valide car le **champ récursif** n'est pas un **pointeur**.

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

Déclaration de types structurés mutuellement récursifs

Il est possible de **déclarer des types structurés mutuellement récursifs** :

```
typedef struct flip {
    int a;
    int b;
    struct flop *s;
} Flip;

typedef struct flop {
    double d;
    struct flip *s;
} Flop;
```

– Exemple –

Une variable typique `x` de type `Flip` est de la forme

Plan

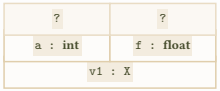
8. Types structurés

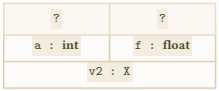
Affectation de variables d'un type structuré

Considérons le code

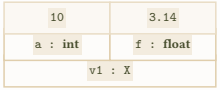
```
typedef struct {
    int a;
    float f;
} X;
...
X v1, v2;
v1.a = 10;
v1.f = 3.14;
v2 = v1;
v2.a = 20;
```

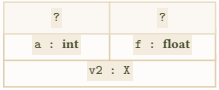
l. 6 :



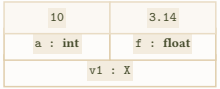


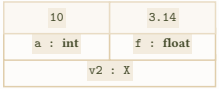
l. 8 :



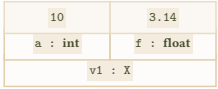


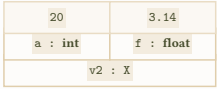
l. 9 :





l. 10 :





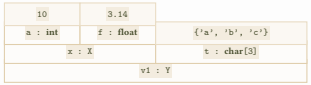
Observation : l'affectation **recopie** les champs d'une variable d'un **type scalaire**.

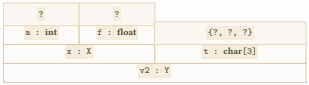
Affectation de variables d'un type structuré

Considérons le code

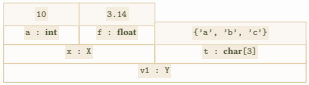
```
typedef struct {
    X x;
    char t[3];
} Y;
...
Y v1, v2;
v1.x.a = 10;
v1.x.f = 3.14;
v1.t = {'a', 'b', 'c'};
v2 = v1;
v2.x.f = 1.8;
v2.t[0] = 'g';
```

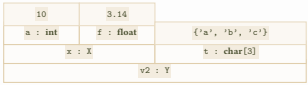
l. 9 :



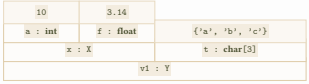


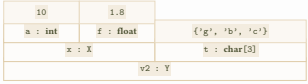
l. 10 :





l. 12 :



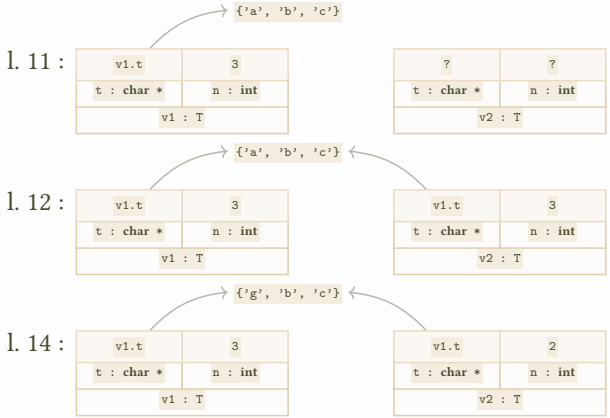


Observation : l'affectation **recopie** les champs d'une variable d'un type structuré de manière **récursive** et les **tableaux statiques**.

Affectation de variables d'un type structuré

Considérons le code

```
typedef struct {
    char *t;
    int n;
} T;
...
T v1, v2;
v1.t = malloc(3);
v1.n = 3;
v1.t[0] = 'a';
v1.t[1] = 'b';
v1.t[2] = 'c';
v2 = v1;
v2.n = 2;
v2.t[0] = 'g';
```



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

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

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

– Exemple –

La définition du type `T` précédent s'accompagne de la définition de la fonction

Cette fonction est munie du mécanisme habituel de gestion d'erreurs.

```
int copier_T(const T *v1, T *v2) {
    assert(v1 != NULL);
    assert(v2 != NULL);
    v2->n = v1->n;
    v2->t = (char *) malloc(sizeof(char) * v1->n);
    if (v2->t == NULL) return 0;
    for (int i = 0 ; i < v1->n ; ++i)
        v2->t[i] = v1->t[i];
    return 1;
}
```

Comparaison de variables d'un type structuré

Considérons le code

```
typedef struct {
    int a;
    int b;
} A;
...
A v1, v2;
...
if (v1 == v2) {...}
...
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`.

– Exemples –

La définition du type `A` précédent s'accompagne de la définition des fonctions

```
int sont_ega_A(A *v1, A *v2) {
    assert(v1 != NULL);
    assert(v2 != NULL);
    return (v1->a == v2->a) && (v1->b == v2->b);
}
```

```
int sont_dif_A(A *v1, A *v2) {
    assert(v1 != NULL);
    assert(v2 != NULL);
    return !sont_ega_A(v1, v2);
}
```

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`.

– Exemple –

La déclaration du type `B` suivant s'accompagne de la définition de la fonction

```
typedef struct {
    int *tab;
    int n;
} B;
```

```
void detruire_B(B *v) {
    assert(v != NULL);
    free(v->tab);
}
```

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

8. Types structurés

Renvoi d'une variable d'un type structuré

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

Couple twist(Couple c) {
    Couple res;
    res.x = c.y;
    res.y = c.x;
    return res;
}
```

Ce code 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, ce code n'est pas efficace car, à chaque appel de fonction

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

la variable `res`, qui est située dans la pile, doit être **recopiée**.

Paramètre variable d'un type structuré

```
typedef struct {
    int tab1[2048];
    int tab2[2048];
} DeuxTab;

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

Le code 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, ce code 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 :

```
typedef struct {
    int *tab;
    int n;
} Tab;
```

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

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 la règle générale suivante :

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

– Exemple –

Voici le nécessaire pour calculer la somme pondérée de deux points selon les conventions établies :

```
typedef struct {
    float x;
    float y;
} Point;

void somme_points(const Point *p1, const Point *p2, float coeff1, float coeff2, Point *res) {

    assert(p1 != NULL);
    assert(p2 != NULL);
    assert(res != NULL);

    res->x = coeff1 * p1->x + coeff2 * p2->x;
    res->y = coeff1 * p1->y + coeff2 * p2->y;
}
```

Résumé

Voici en résumé la bonne marche à suivre lors de la manipulation de types structurés :

1. on utilise l'**étiquette** lors de la déclaration de types structurés **récursifs** et/ou **mutuellement récursifs** ;
2. toute **déclaration d'un type structuré** s'accompagne de la définition des quatre fonctions suivantes :
   - une fonction de **copie** ;
   - une fonction de **test d'égalité** ;
   - une fonction de **test d'inégalité** ;
   - une fonction de **destruction** ;
3. on passe les **paramètres** d'un type structuré **par adresse** (ne pas oublier d'ajouter les qualificateurs `const` nécessaires).

8. Types structurés

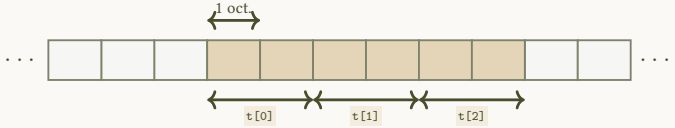
Alignement en mémoire

L'alignement en mémoire d'une donnée est la façon dont celle-ci est organisée dans la mémoire.

– Exemple –

Nous avons déjà vu que les **tableaux** de taille `n` d'éléments d'un type `T` sont organisés en un segment contigu de **`sizeof`**`(T) * n` octets.

Ainsi, un tableau `t` de `3` éléments de type **`short`** est organisé en



On peut se poser de la même manière la question de l'**alignement mémoire** des variables d'un **type structuré**.

Alignement en mémoire des variables d'un type structuré

Considérons les déclarations de types

```
typedef struct {
    short x;
    short y;
    int z;
} A;
```

```
typedef struct {
    short x;
    int z;
    short y;
} B;
```

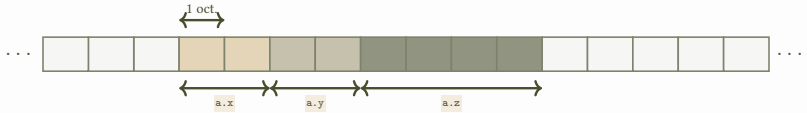
`A` et `B` sont des types structurés composés des mêmes champs. Il n'y a que l'ordre de leur déclaration qui diffère.

Cependant, l'instruction `printf("%lu %lu\n", sizeof(A), sizeof(B));` affiche `8 12`.

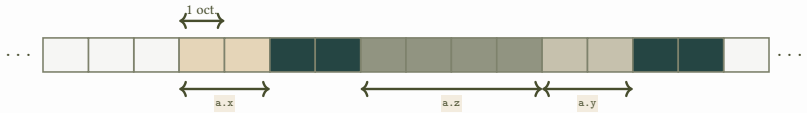
Le fait que les tailles des variables de type `A` et `B` diffèrent est dû à leur **alignements en mémoire** respectifs qui ne sont pas les mêmes.

Alignement en mémoire des variables d'un type structuré

Soit `a` une variable de type `A`. Cette variable est organisée en mémoire en



Soit `b` une variable de type `B`. Cette variable est organisée en mémoire en

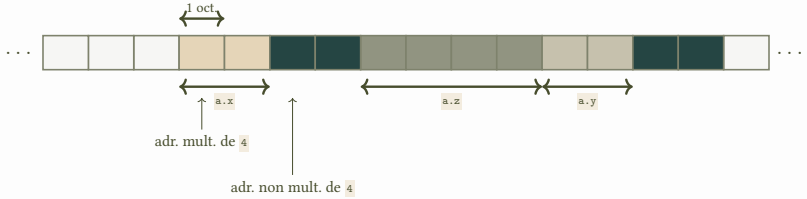


Les octets foncés intervenant dans l'alignement mémoire de `b` sont des octets de complétion.

Octets de complétion

Des octets de complétion sont introduits pour que chaque champ `c` d'une variable d'un type structuré **commence à une adresse multiple d'un entier** dépendant du type de `c`.

Dans notre exemple, en sachant que tout champ de type **short** (resp. **int**) doit commencer à une adresse multiple de `2` (resp. `4`), on explique l'alignement en mémoire de `b` précédent :



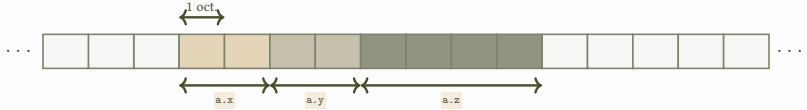
Les derniers octets de complétion sont introduits pour que les tableaux de variables de type `B` puissent être représentés en vérifiant cet alignement en mémoire.

Accès manuel aux champs

Soit `a` une variable de type `A` initialisée par

```
A a = {1000, 2000, 3000};
```

Cette variable est organisée en mémoire en



On peut accéder aux champs de `a` de la manière suivante :

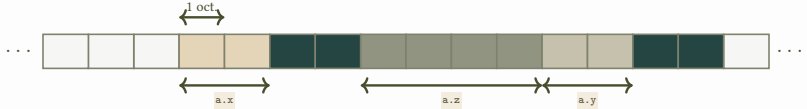
```
short x, y;
int z;
void *p;
p = &a;
x = *((short *) p); /* Equivalent à x = a.x; */
p += 2; /* p est de type void * : l'adresse p est incrementee de 2 */
y = *((short *) p); /* Equivalent à y = a.y; */
p += 2;
z = *((int *) p);   /* Equivalent à z = a.z; */
```

Accès manuel aux champs

Soit `b` une variable de type `B` initialisée par

```
B b = {1000, 2000, 3000};
```

Cette variable est organisée en mémoire en



On peut accéder aux champs de `b` de la manière suivante :

```
short x, y;
int z;
void *p;
p = &b;
x = *((short *) p); /* Equivalent à x = b.x; */
p += 4;
z = *((int *) p);   /* Equivalent à z = b.z; */
p += 4;
y = *((short *) p); /* Equivalent à y = b.y; */
```

L'option `Wpadded`

L'option du compilateur `-Wpadded` permet d'obtenir un avertissement sanctionnant la déclaration d'un type structuré nécessitant des octets de complétion.

– Exemple –

Avec le type structuré `B` défini par

```
typedef struct {
    short x;
    int z;
    short y;
} B;
```

on obtient l'avertissement

```
Prog.c:3:9: warning: padding struct to align ‘z’ [-Wpadded]
    int z;
        ^
Prog.c:5:1: warning: padding struct size to alignment boundary [-Wpadded]
 } B;
 ^
```

Alignement en mémoire — ce qu'il faut retenir

L'alignement mémoire d'une variable d'un type structuré dépend de beaucoup de paramètres, notamment :

- de l'architecture de la machine exécutant ou ayant compilé le programme ;
- du compilateur ayant compilé le programme.

Il est donc important de savoir que le calcul de la **taille** d'une variable d'un **type structuré** n'est pas immédiat et **dépend du contexte**.

Dans la pratique, on ne cherchera pas à déclarer des types structurés qui ne nécessitent pas d'octet de complétion. Au contraire : il est de loin préférable de déclarer les champs dans un ordre logique favorisant la relecture et la maintenance.

8. Types structurés

Les unions

Les unions est une variation de type structuré qui respecte les mêmes règles que les structures. Cependant il ne permet de représenter la valeur que d'**un seul** de ses membres.

La syntaxe est la même que pour les structures :

```
union ÉTIQUETTE {
    TYPE_1 CHAMP_1;
    TYPE_2 CHAMP_2;
    ...
};
```

La même zone mémoire est utilisée pour tous les membres. Sa taille correspond donc à la taille du plus grand de ses membres.

– Exemple –

```
union Nombre {
    int i;
    float f;
};
```

déclare un type structuré `union Nombre` qui permet de représenter soit un entier `int`, soit un floattant `float`.

`union Nombre nombre;` déclare une variable de ce type union.

Utilisation des unions

Comme la zone mémoire est partagée par tous ses membres, les unions doivent être utilisés correctement pour rester portable.

Le développeur garantit de lire un champ dont le type est cohérent avec la dernière donnée écrite.

```
union Nombre n;
n.i = 42;
printf("%d\n", n.i); /* ok */
printf("%f\n", n.f); /* risque */
```

Une utilisation classique combine une structure et une énumération :

– Exemple –

```
struct ABC {
    EnumType type;
    union {
        TypeA a;
        TypeB b;
        TypeC c;
    } val;
};
```

On déclare ici un type `struct ABC` qui permet de représenter soit un `TypeA`, soit un `TypeB`, soit un `TypeC`, le champ `type` indiquant quel est le type de la valeur stockée dans l'union.

Plan

8. Types structurés

Objet anonyme ou *compound literals*

En C (depuis C99), on peut créer des objets anonymes appelés « *compound literal* » [3] (littéraux composés) avec la syntaxe suivante :

```
(Type){ liste d'initialisation }
```

Dans une fonction, cette **expression** crée dans la pile un objet anonyme de type `Type` et l'initialise à partir de la liste d'initialisation. Cet objet respecte la même sémantique qu'une variable locale (portée, adresse).

– Exemple –

```
typedef struct {
    int x;
    int y;
} Couple;
Couple twist(Couple c) {
    return (Couple){
        .x = c.y,
        .y = c.x,
    };
}
```

```
typedef struct {
    int *tab;
    int taille;
} TabInt;

void detruire(TabInt *ti) {
    free(ti->tab);
    // remise à zéro
    *ti = (TabInt){ 0 };
}
```

```
typedef struct cell{
    struct cell *next;
} Cell;

int test_longueur(void) {
    Cell *list2 = &(Cell){
        .next = &(Cell){ NULL },
    };
    return longueur(list2) == 2;
}
```

3. C99 standard 6.5.2.5 Compound literals

Axe 3 : utiliser quelques techniques avancées

9. Opérateurs

10. Pointeurs de fonction

11. Mémoïsation

12. Génération aléatoire

9. Opérateurs

9. Opérateurs

Caractéristiques d'un opérateur

Un **opérateur** dispose des caractéristiques structurelles suivantes :

1. son arité, qui désigne le nombre d'opérandes sur lesquelles il agit ;
2. sa précédence, qui permet de savoir, dans une expression, dans quel ordre appliquer les différents opérateurs qui la composent ;
3. pour les opérateurs binaires (d'arité 2), son sens d'associativité, qui permet de savoir, dans une expression, dans quel sens appliquer des mêmes opérateurs qui la composent.

Précédence et associativité des opérateurs

- Considérons l'expression `3 * 2 + 1`.

  Suivant les priorités relatives des opérateurs `*` et `+`, il y deux manières de l'évaluer :

  1. `(3 * 2) + 1`, si `*` est **plus prioritaire** que `+` ;
  2. `3 * (2 + 1)`, si `+` est **plus prioritaire** que `*`.

- Considérons l'expression `4 - 3 - 2 - 1`.

  Suivant le sens d'associativité de `-`, il y a deux manières de l'évaluer :

  1. `((4 - 3) - 2) - 1`, si `-` est **associatif de gauche à droite** ;
  2. `4 - (3 - (2 - 1))`, si `-` est **associatif de droite à gauche**.

Tout ceci peut être rendu explicite par l'**utilisation de parenthèses**.

9. Opérateurs

Opérateurs de gestion de la mémoire

Op.	Rôle	Ari.	Assoc.	Opérandes
`&`	référencement	1	–	une variable
`*`	déréférencement	1	–	un pointeur
`[]`	élément d'un tableau	2	$\longrightarrow$	un pointeur et un entier
`.`	valeur d'un champ	2	$\longrightarrow$	une var. d'un type struct. et un id. de champ
`->`	valeur d'un champ	2	$\longrightarrow$	une pointeur sur une var. d'un type struct. et un id. de champ

9. Opérateurs

Opérateurs arithmétiques

Op.	Rôle	Ari.	Assoc.	Opérandes
`+`, `-`, `*`, `/`	opérations arith.	2	$\longrightarrow$	deux val. numériques
`%`	modulo	2	$\longrightarrow$	deux entiers
`+`, `-`	signe	1	–	une val. numérique
`++`, `--`	incr. / décr.	1	–	une var. d'un type numérique ou un pointeur

L'opérateur modulo

L'opérateur **modulo** `%` calcule le reste de la division euclidienne de son premier opérande par son second.

D'un point de vue mathématique, si a et b sont deux entiers, on a $a = b \times q + r$, où $0 \leqslant r \leqslant b - 1$ et q est un entier. q est le quotient et r est le reste, **toujours positif**.

Cependant, `%` peut produire des valeurs négatives, dans le cas où le premier opérande est négatif.

Solution pour un modulo qui respecte la définition mathématique :

```
int vrai_modulo(int a, int b) {
    int r;
    assert(b != 0);
    r = a % b;
    if (r < 0)
        return r + b;
    return r;
}
```

Les opérateurs d'incrémentation et de décrémentation

Les opérateurs `++` et `--` existent en deux versions, suivant qu'ils soient préfixes ou suffixes :

1. `a++`, incrémente (de un) la valeur de la variable `a` et est une expression dont la valeur est l'**ancienne valeur** de `a` ;
2. `++a`, incrémente (de un) la valeur de la variable `a` et est une expression dont la valeur est la **nouvelle valeur** de `a`.

– Exemples –

```
int a = 5, b;
b = 3 + a++;
```

`b` vaut `8` et `a` vaut `6`.

```
int a = 5, b;
b = 3 + ++a;
```

`b` vaut `9` et `a` vaut `6`.

Les opérateurs d'incrémentation et de décrémentation

Attention au **pièges d'utilisation** de ces opérateurs.

– Exemples –

Les instructions

```
int a = 5, b;
b = a++ + ++a;
```

```
int a = 5;
a = a++;
```

```
int a = 5;
a = ++a;
```

ne sont pas évaluables (l'effet produit par les lignes 2 dépend du compilateur et de ses options).

Règle : pour éviter ce type de piège, on s'interdit de réaliser plus d'une modification d'une même variable dans une même expression.

Opérateurs relationnels

Op.	Rôle	Ari.	Assoc.	Opérandes
`<`, `>`	comparaison stricte	2	$\longrightarrow$	deux val. numériques
`<=`, `>=`	comparaison large	2	$\longrightarrow$	deux val. numériques
`==`	égalité	2	$\longrightarrow$	deux val. numériques
`!=`	différence	2	$\longrightarrow$	deux val. numériques

Toutes les expressions de la forme

```
v1 CMP v2
```

où `v1` et `v2` sont des valeurs numériques et `CMP` est un opérateur de comparaison produisent une valeur :

- `1` si la comparaison est vraie ;
- `0` sinon.

Opérateurs relationnels

Un pointeur étant une adresse, et donc une valeur numérique, il est possible de comparer deux pointeurs.

– Exemple –

```
char *ptr1, *ptr2;
char c;
c = 'a';
ptr1 = &c;
ptr2 = &c;
if (ptr1 == ptr2)
    printf("ok1\n");
if (&ptr1 == &ptr2)
    printf("ok2\n");
```

Ceci affiche seulement `ok1`.

En effet, les deux pointeurs `ptr1` et `ptr2` pointent vers le même emplacement en mémoire.

Le second test est faux car les adresses des variables `ptr1` et `ptr2` sont différentes.

```
int t1[2], t2[2];
t1[0] = 1;
t1[1] = 2;
t2[0] = 1;
t2[1] = 2;
if (t1 == t2)
    printf("ok\n");
```

Ceci compare les **adresses** de `t1` et `t2` et non pas les valeurs de leurs cases.

Rien n'est donc affiché car les tableaux `t1` et `t2` sont à des adresses différentes.

Opérateurs logiques

Op.	Rôle	Ari.	Assoc.	Opérandes
`&&`	et logique	2	$\longrightarrow$	deux val. numériques
`||`	ou logique	2	$\longrightarrow$	deux val. numériques
`!`	non logique	1	–	une val. numérique

Toutes les expressions formées d'opérateurs logiques produisent une valeur, `0` ou bien `1`.

Cette valeur est

- `1` si l'expression logique est vraie ;
- `0` sinon.

Opérateurs bit à bit

Op.	Rôle	Ari.	Assoc.	Opérandes
`&`	et bit à bit	2	$\longrightarrow$	deux val. entières
`|`	ou bit à bit	2	$\longrightarrow$	deux val. entières
`^`	xor bit à bit	2	$\longrightarrow$	deux val. entières
$\sim$	non bit à bit	1	–	une val. entière
`<<`, `>>`	déc. g./d. bit à bit	2	$\longrightarrow$	deux val. entières

Et / ou / xor /non bit à bit

– Exemples –

`x`	0	1	1	1	0	1	0	1
`y`	1	0	1	0	1	1	0	0
`x & y`	0	0	1	0	0	1	0	0

`x`	0	1	1	1	0	1	0	1
`y`	1	0	1	0	1	1	0	0
`x | y`	1	1	1	1	1	1	0	1

`x`	0	1	1	1	0	1	0	1
`y`	1	0	1	0	1	1	0	0
`x ^ y`	1	1	0	1	1	0	0	1

`x`	0	1	1	1	0	1	0	1
`~x`	1	0	0	0	1	0	1	0

Opérateurs bit à bit et taille des opérandes

Si les deux opérandes n'ont pas la même taille (en nombre de bits), le plus petit est complété à gauche par des

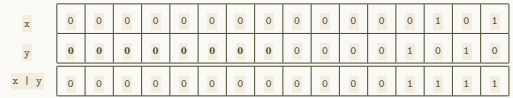
- **zéros** s'il est non signé ou bien positif ;
- **uns** s'il est négatif et signé.

Une conséquence du **codage en complément à deux** est que le signe d'un entier signé est lu sur son bit de poids fort :

- `0` s'il est positif ;
- `1` s'il est négatif.

– Exemples –

```
short x = 5;
char y = 10;
x = x | y;
```




```
short x = 5;
char y = -10;
x = x | y;
```

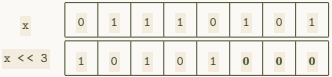
Décalages bit à bit de valeurs non signées

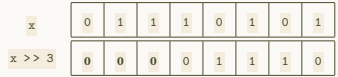
Soit `x` une variable de type entier (**`char`**, **`short`**, **`int`** ou **`long`**).

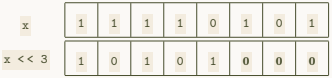
Si `x` est non signé (déclaré avec **`unsigned`**),

- le décalage à gauche insère des bits `0` ;
- le décalage à droite insère des bits `0`.

– Exemples –







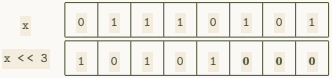
Décalages bit à bit de valeurs signées

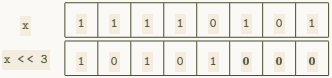
Soit `x` une variable de type entier (**`char`**, **`short`**, **`int`** ou **`long`**).

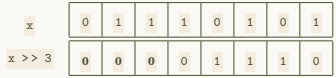
Si `x` est signé (déclaré sans **`unsigned`**),

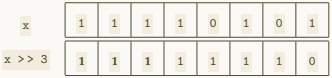
- le décalage à gauche insère des bits `0` ;
- le décalage à droite insère des bits **égaux au bit de poids fort** de `x`.

– Exemples –









Exemple — ensembles finis

Les opérateurs bit à bit sont adaptés pour représenter des **ensembles finis** et réaliser des opérations ensemblistes de manière simple et efficace.

Soit E un ensemble à 32 éléments.

On considère que E est muni d'une relation d'ordre totale de sorte que ses éléments puissent être indexés de 0 à 31. Ainsi,

$$E = \{e_0, e_1, e_2, \ldots, e_{31}\}.$$

Cette indexation permet de représenter tout **sous-ensemble** S de E par un mot de 32 bit dont le i^{e} bit code la présence (`1`) ou l'absence (`0`) de e_i dans S.

– Exemple –

L'entier dont l'écriture binaire est

```
00010000 10000000 00000011 00000101
```

code le sous ensemble $\{e_0, e_2, e_8, e_9, e_{23}, e_{28}\}$ de E.

Exemple — ensembles finis

Ceci mène à la déclaration du type alias

```
typedef unsigned int SousEnsemble;
```

Pour tester si e_i **appartient** à S, il suffit de réaliser un et bit à bit entre l'entier représentant S et le mot binaire constitué d'un unique `1` en i^{e} position.

Cette expression vaut `0` si $e_i \notin S$ et une valeur non nulle sinon.

Ceci est implanté par la fonction

```
int appartient_e_i(SousEnsemble s, int i) {
    assert(0 <= i);
    assert(i <= 31);

    return (1 << i) & s;
}
```

Exemple — ensembles finis

Pour réaliser l'**union** de deux sous-ensembles S_1 et S_2 de E, il suffit de réaliser un ou bit à bit des deux entiers représentant S_1 et S_2. En effet, pour tout i, $e_i \in S_1 \cup S_2$ si $e_i \in S_1$ ou $e_i \in S_2$.

Ceci est implanté par la fonction

```
SousEnsemble union(SousEnsemble s_1, SousEnsemble s_2) {
    return s_1 | s_2;
}
```

Pour réaliser l'**intersection** de deux sous-ensembles S_1 et S_2 de E, il suffit de réaliser un et bit à bit des deux entiers représentant S_1 et S_2. En effet, pour tout i, $e_i \in S_1 \cap S_2$ si $e_i \in S_1$ et $e_i \in S_2$.

Ceci est implanté par la fonction

```
SousEnsemble intersection(SousEnsemble s_1, SousEnsemble s_2) {
    return s_1 & s_2;
}
```

Exemple — compter le nombre de bits `1`

Objectif : écrire une fonction `compter_un` qui

- prend en entrée un nombre `x` de 64 bits ;
- renvoie le **nombre de bits** `1` dans `x`.

Informations et rappels :

- le type `unsigned long long` est adapté pour représenter des nombres de 64 bits ;
- le `unsigned` est utile pour s'assurer que lors de décalages à droite bit à bit, ce sont toujours des `0` qui sont insérés.

Dans la suite, on suppose déclaré le type `Mot64` par

```
typedef unsigned long long Mot64;
```

Méthode 1 : décalages successifs

L'idée consiste à observer par `x & 1` si le bit de poids faible de `x` est à un, à en tenir compte, puis à décaler `x` vers la droite pour poursuivre l'observation sur le bit suivant.

– Exemple –

Pour simplifier, on se place sur 5 bits. On obtient successivement

x	x & 1	x >> 1
0110**1**	**1**	00110
0011**0**	**0**	00011
0001**1**	**1**	00001
0000**1**	**1**	00000
0000**0**	**0**	00000

Voici une implantation possible :

```
int compter_un_v1(Mot64 x) {
    int res;

    res = 0;
    for (int i = 0 ; i < 64 ; ++i) {
        if (x & 1)
            res += 1;
        x = x >> 1;
    }
    return res;
}
```

Méthode 2 : extinctions successives

L'idée consiste à utiliser le fait que, quand `x` $\neq$ `0 ... 0`, l'expression `x & -x` est l'entier qui contient comme unique bit à un le `1` le plus à droite de `x`.

Ainsi, l'instruction

```
x = x ^ (x & -x);
```

transforme le bit `1` le plus à droite de `x` par un `0`.

– Exemple –

Pour simplifier, on se place sur 5 bits. On obtient successivement

x	-x	x & -x	x ^ (x & -x)
01101	10011	00001	01100
01100	10100	00100	01000
01000	11000	01000	00000

Voici une implantation possible :

```
int compter_un_v2(Mot64 x) {
    int res;

    res = 0;
    while (x != 0) {
        x = x ^ (x & -x);
        res += 1;
    }
    return res;
}
```

Méthode 3 : par table

Cette méthode n'utilise **pas de boucle** mas une **table pré-calculée**.

On commence par construire un tableau qui associe à tout entier de huit bits (un `char`) le nombre de bits à un qu'il contient, en utilisant au choix l'une des deux méthodes précédentes.

```
int nombre_un_v3[256]; /* Table en variable globale. */

void initialiser_nombre_un_v3() {
    Mot64 x;
    for (int i = 0 ; i < 256 ; ++i) {
        x = (Mot64) i;
        nombre_un_v3[i] = compter_un_v2(x);
    }
}
```

À ce stade, après avoir effectué ce pré-calcul, nous avons par exemple que

- `nombre_un_v3[0xA1]` a pour valeur `3` car `0xA1` est l'octet `10100001` ;
- `nombre_un_v3[0x78]` a pour valeur `4` car `0x78` est l'octet `01111000`.

Méthode 3 : par table

On isole l'octet d'indice `i` d'une variable `x` de type `Mot64` par l'expression

```
0xFF & (x >> (8 * i))
```

Comme le nombre de bits à un de `x` est la somme du nombre de bits à un de chacun des huit octets qui le constituent, on obtient

```
int compter_un_v3(Mot64 x) {
    int res;
    res = 0;
    res += nombre_un_v3[0xFF & x];
    res += nombre_un_v3[0xFF & (x >> 8)];
    res += nombre_un_v3[0xFF & (x >> 16)];
    res += nombre_un_v3[0xFF & (x >> 24)];
    res += nombre_un_v3[0xFF & (x >> 32)];
    res += nombre_un_v3[0xFF & (x >> 40)];
    res += nombre_un_v3[0xFF & (x >> 48)];
    res += nombre_un_v3[0xFF & (x >> 56)];
    return res;
}
```

Méthode 4 : par table plus grande

Nous pouvons pousser plus loin l'idée précédente en considérant des blocs de deux octets (au lieu d'un seul).

```
int nombre_un_v4[65536];

void initialiser_nombre_un_v4() {
    Mot64 x;
    for (int i = 0 ; i < 65536 ; ++i) {
        x = (Mot64) i;
        nombre_un_v4[i] = compter_un_v2(x);
    }
}
```

Taille mémoire occupée par le tableau `nombre_un` :

$$2^{16} \times \texttt{sizeof(int)}\,\mathrm{o} \;=\; 2^{18}\,\mathrm{o} \;=\; \frac{2^{18}}{2^{10}}\,\mathrm{Kio} \;=\; 256\,\mathrm{Kio}.$$

Méthode 4 : par table plus grande

Ceci fournit la solution suivante.

```
int compter_un_v4(Mot64 x) {
    int res;
    res = 0;
    res += nombre_un_v4[0xFFFF & x];
    res += nombre_un_v4[0xFFFF & (x >> 16)];
    res += nombre_un_v4[0xFFFF & (x >> 32)];
    res += nombre_un_v4[0xFFFF & (x >> 48)];
    return res;
}
```

Elle ne demande que quatre lectures dans le tableau `nombre_un`, au lieu des huit de la méthode précédente.

À l'extrême, il est impossible de maintenir un tableau `nombre_un` qui ne demanderait qu'une seule lecture. En effet, sa taille mémoire serait de

$$2^{64} \times \texttt{sizeof(int)}\,\mathrm{o} \;=\; 2^{66}\,\mathrm{o} \;=\; \frac{2^{66}}{2^{30}}\,\mathrm{Gio} \;=\; 2^{36}\,\mathrm{Gio}.$$

Méthode 5 : par masques bit à bit

Cette méthode est la plus compliquée. Elle se base sur une compréhension très fine du fonctionnement des opérateurs bit à bit.

```
int compter_un_v5(Mot64 x) {
    x = x - ((x >> 1) & 0x5555555555555555LLU);
    x = (x & 0x3333333333333333LLU) + ((x >> 2) & 0x3333333333333333LLU);
    x = (x + (x >> 4)) & 0x0F0F0F0F0F0F0F0FLLU;
    x = (x * 0x0101010101010101LLU) >> 56;
    return (int) x;
}
```

Mesure du temps d'exécution

Mesurons maintenant les efficacités des cinq méthodes.

On utilise pour cela la fonction

```
clock_t clock();
```

de `time.h`.

Schéma général pour **mesurer le temps d'exécution** d'une suite d'instructions :

```
clock_t debut, fin;
double temps;

debut = clock();
/* Instructions a mesurer ici. */
fin = clock();

/* Affichage du temps. */
temps = (double) (fin - debut) / CLOCKS_PER_SEC;
printf("%g s\n", temps);
```

Génération aléatoire d'entrées

On mesure le temps d'exécution des cinq méthodes en leur donnant en entrée des nombres de 64 bits générés de manière aléatoire.

Pour générer un nombre de 32 bits de manière aléatoire, on utilise la fonction `int rand();` de `stdlib.h`.

Le nombre de 64 bits est construit en générant aléatoirement ses quatre octets de droite, puis ses quatre octets de gauche et en les **associant avec un ou bit à bit** :

```
Mot64 mot64_alea() {
    Mot64 gauche, droite;

    gauche = rand();
    droite = rand();
    return (gauche << 32) | droite;
}
```

– Exemple –

En supposant pour simplifier que nous souhaiter générer un nombre de 10 bits à partir de deux nombres de 5 bits :

`gauche`	00000	10011
`gauche << 5`	10011	00000
`droite`	00000	01010
`(gauche << 5) | droite`	10011	01010

Comparaison des performances

Voici les temps réalisés par chacune des cinq méthodes sur 84000000 nombres de 64 bits générés aléatoirement :

Méthode	Caractéristique	Temps (s)
1	Décalages successifs	27.31
2	Extinctions successives	6.36
3	Par petite table	2.53
4	Par grande table	2.37
5	Par masques	2.33

La 2e méthode est plus de **4 fois plus rapide** que la 1re et la 5e est presque **12 fois plus rapide** que la 1re.

Les 3e et 4e méthodes demandent un **pré-calcul** et une occupation mémoire (par la table).

La 5e méthode est la plus rapide et ne demande aucun pré-calcul. Elle est en revanche difficile à comprendre et très difficile à imaginer.

9. Opérateurs

Opérateurs d'affectation

Op.	Rôle	Ari.	Assoc.	Opérandes
`=`	affect.	2	⟵	une var. et une val.
`+=`, `-=`, `*=`, `/=`, `%=`	affect. compo. arith.	2	⟵	une var. num. et une val. num
`&=`, `|=`, `^=`, `<<=`, `>>=`	affect. compo. bit à bit	2	⟵	une var. ent. et une val. ent.

Toute expression de la forme `a X= b` est équivalente à `a = a X b`.

Opérateurs d'affectation

Toutes les expressions d'affectation produisent une valeur qui est la **valeur qui vient d'être affectée**.

– Exemple –

Dans

```
int a, b;
a = 2;
b = 5;
a *= b += 3;
```

à cause de l'associativité des opérateurs d'affectation, la l. 4 s'interprète comme `a *= (b += 3);`.

Ainsi, comme `b += 3` produit la valeur `8`, `a` vaut finalement `16`.

9. Opérateurs

Autres opérateurs

Op.	Rôle	Ari.	Assoc.	Opérandes
`sizeof`	taille	1	–	~~une val.~~, ~~une var.~~ ou un **type**
`(T)`	coercition `T` est un type	1	–	une val.
`? :`	condition	3	–	une val. num. et deux val.
`,`	séquence	2	$\longrightarrow$	deux val.

L'opérateur de séquence

Dans l'expression `v1, v2`, où `v1` et `v2` sont des expressions, on commence par évaluer `v1` puis ensuite `v2`. Cette expression produit la valeur de `v2`.

Attention : l'opérateur de séquence a la plus faible priorité parmi les opérateur en C. Ainsi le code `a = b, c` peut se lire `(a = b), c`.

L'opérateur `,` est le plus souvent utilisé dans les **champs des boucles** `for` et est à proscrire ailleurs.

– Exemple –

```
int i, j, l;
...
for (i = 0, j = l - 1 ; i < j ; ++i, --j) {
    ...
}
```

permet d'obtenir une boucle `for` avec **deux compteurs** : `i` croît et `j` décroît dans l'intervalle allant de `0` à `l - 1`.

10. Pointeurs de fonction

Plan

10. Pointeurs de fonction

Pointeurs de fonction

En C, il est possible de 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`.

– Exemple –

```
#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;
}
```

L'exécution de ce programme affiche `0x40052d` et `0x400541`. Ce sont 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

Pour **déclarer** un pointeur de fonction, nous utilisons

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

où

- `T`, `T1`, …, `TN` sont des types ;
- `FCT` est un identificateur.

Ce pointeur de fonction possède `FCT` comme identificateur et est destiné à contenir l'adresse d'une fonction de type de retour `T` et qui possède `N` paramètres de types respectifs `T1`, …, `TN`.

– Exemple –

```
float moyenne(int a, int b) {
    return (0.0 + a + b) / 2;
}

/* Declaration d'un pointeur 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 mentionnerons les `&` dès que nécessaire,

Champs pointeurs de fonction

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

– Exemple –

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

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

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

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

Ici, `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 main() {
    Suite s;

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

Ceci affiche `2 4 8 16 32 64`.

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 1re méthode.

Tableaux de pointeurs de fonction

– Exemple –

```c
/* Alias pour un type pointeur de fonction */
typedef int (*opb)(int, int);

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

```c
int main() {
    /* Declaration du tableau de pointeur de fonctions */
    opb tab[2];

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

Ceci affiche `30 200`.

Les tableaux de pointeurs de fonction peuvent être **dynamiques**.

– Exemple –

Dans l'exemple précédent, la ligne où le tableau est déclaré peut être remplacée par

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

Plan

10. Pointeurs de fonction

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.

– Exemple –

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

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

– Exemples –

```
int add_1(int n) {
    return n + 1;
}
...
printf("%d\n", appliquer(3, 4, &add_1));
```

Calcule `(((3 + 1) + 1) + 1) + 1` et affiche `7`.

```
int mul_2(int n) {
    return 2 * n;
}
...
printf("%d\n", appliquer(3, 4, &mul_2));
```

Calcule `(((3 * 2) * 2) * 2) * 2` et affiche `48`.

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) {
    /* Corps de la fonction */
}
```

permet de définir directement une fonction `FCT` de type de retour `R` et qui possède `M` paramètres de types respectifs `R1`, …, `RM`.

Cependant, le code devient illisible.

Renvoi d'un pointeur de fonction

– Exemple –

Ici, nous utilisons des pointeurs de fonction pour réaliser des opérations aléatoires.

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

```
/* Type de retour */
typedef int (*opb)(int, int);

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

Ceci affecte, de manière aléatoire et uniforme, la valeur `7`, `-1` ou `3` à `n`.

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

Plan

10. Pointeurs de fonction

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**.

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 le 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`.

Ceci est une expression de type `T*`. Son évaluation ne provoque pas d'effet secondaire.

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

Nous souhaitons comparer deux segments de la mémoire commençant respectivement à des adresses `x` et `y` s'étendant sur `nbo` octets.

```
int ega(int nbo, void *x, void *y) {
    char *xc = x; /* Conversion implicite */
    char *yc = (char *) y; /* Coercition */
    for (int i = 0 ; i < nbo ; ++i)
        if (xc[i] != yc[i]) /* Dereferencement */
            return 0;
    return 1;
}
```

Pour effectuer cette comparaison, nous interprétons les zones de la mémoire à comparer comme des tableaux de `nbo` octets (type `char`).

Cette 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**.

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

Ceci compare deux variables `t1` et `t2`, toutes deux d'un même type `T`.

– Exemple –

```
short tab1[8], tab2[8];
...
ega(sizeof(short) * 8, tab1, tab2);
```

Ceci compare le contenu des deux tableaux statiques `tab1` et `tab2`.

Fonction générique d'affichage de tableau

Nous souhaitons afficher un tableau `tab` de `n` cases de pointeurs génériques.

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

Cette fonction 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**.

Pour afficher un tableau générique, on procède en deux temps :

1. on implante **spécifiquement** une fonction `aff` de même type que `aff_elt` et qui permet d'afficher un élément du tableau ;
2. on appelle la fonction `aff_tab` avec cette fonction.

Fonction générique d'affichage de tableau

– Exemple –

```c
/* Fonction specifique d'affichage d'un element */
void aff_int(void *x) {
    int e = *((int *) x);
    printf("%d", e);
}

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

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

Ceci affiche un tableau `tab` de taille `13` de pointeurs sur des **entiers**.

La coercition `(void **) tab` n'est pas obligatoire dans l'appel à `aff_tab`.

Fonction générique d'affichage de tableau

– Exemple –

```
typedef struct {
    int jour;
    int mois;
    int annee;
} Date;

/* Fonction specifique d'affichage d'un element */
void aff_date(void *d) {
    Date dd = *((Date *) d);
    printf("%d-%d-%d", dd.jour, dd.mois, dd.annee);
}

/* Utilisation */
aff_tab((void **) tab, 23, &aff_date);
```

Ceci affiche un tableau `tab` de taille `23` de pointeurs sur des variables de **type structuré** `Date`.

La coercition `(void **) tab` n'est pas obligatoire dans l'appel à `aff_tab`.

Listes génériques

Nous souhaitons définir une structure de donnée **liste** dont le **type des éléments n'est pas fixé**.

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

typedef Cellule *Liste;
```

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

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

Nous souhaitons afficher une liste générique `lst`.

```
void aff_lst(Liste lst, void (*aff_elt)(void *)) {
    assert(lst != NULL);
    assert(aff_elt != NULL);

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

Cette fonction est générique : elle permet l'**affichage** des éléments d'une liste générique.

Pour afficher une liste générique, on procède en deux temps (comme dans le cas des tableaux) :

1. on implante **spécifiquement** une fonction `aff` de même type que `aff_elt` et qui permet d'afficher un élément de la liste ;
2. on appelle la fonction `aff_lst` avec cette fonction.

Listes génériques

– Exemple –

```c
/* Fonction specifique d'affichage d'un element */
void aff_int(void *e) {
    printf("%d", *((int *) e));
}

/* Creation d'une liste d'entiers */
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;

/* Utilisation */
aff_lst(lst, &aff_int);
```

Ceci créé une liste générique qui s'avère contenir des entiers.

Elle contient les éléments `3`, `14` et `414`.

L'appel à `aff_lst` affiche cette liste.

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 *));`
5. `Liste empiler(Liste lst, void *e);`
6. `Liste miroir(Liste lst);`

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

Plan

10. Pointeurs de fonction

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 $\bullet : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ 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 ;
- $(\mathbb{N}, \max, 0)$ est le monoïde max des entiers naturels ;
- $(\{\mathtt{a}, \mathtt{b}\}^*, \cdot, \epsilon)$ 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 :

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** $n \geqslant 0$ un élément x d'un monoïde.

D'après la définition mathématique d'un monoïde, on aboutit à la définition ci-contre.

```
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 et multiplicatif

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 = (int *) malloc(sizeof(int));
    *res = *((int *) a) + *((int *) b);
    return res;
}
```

Ceci créé le **monoïde additif** :

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

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

Ceci créé le **monoïde multiplicatif** :

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

Définition du monoïde libre

On commence par définir une fonction qui réalise le produit du monoïde libre et dont le type de retour et la signature respectent ceux du champ `produit` :

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

Ceci créé le **monoïde libre** :

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

Fonction générique d'exponentiation

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

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

Application de la fonction d'exponentiation

– Exemples –

```
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 $(\texttt{ab})^4$ dans le monoïde libre, c'est à dire `abababab`.

Fuites mémoire

Les opérations de produit des monoïdes renvoient des valeurs de type `void *`. Ainsi, les fonctions qui implantent ces produits doivent réaliser une allocation de mémoire pour réserver de l'espace mémoire pour ce résultat.

Cependant, dans la fonction `puissance` telle qu'elle est implantée, aucune libération de mémoire n'est entreprise : cette fonction engendre des **fuites mémoires**.

Il est très délicat de disposer des `free` adéquatement dans cette fonction. Il est préférable plutôt de changer le prototype des opérations de produit pour que le résultat soit passé par adresse. Nous travaillons donc plutôt avec le type

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

où le 3e paramètre de `produit` est le résultat du produit, et le retour `int` est un code d'erreur.

Exercice : réécrire avec ce nouveau type les trois implantations des monoïdes précédents ainsi que la fonction générique d'exponentiation, sans provoquer de fuite mémoire.

11. Mémoïsation

Plan

11. Mémoïsation

Notion de variable statique

Une variable statique est une variable dont la **durée de vie** est égale à celle de l'**exécution** du programme. Une telle variable est créée et initialisée à la compilation.

L'instruction

```
static T ID;
```

déclare une variable statique `ID` de type `T`.

Elles sont les contreparties des variables automatiques dont la durée de vie s'étend à l'exécution du bloc dans lesquels elles se trouvent et dont la création se fait à l'exécution le moment voulu.

Les variables automatiques sont les variables habituelles (se déclarent sans le mot clé `static`).

Persistance des variables statiques

Une variable statique est rémanente : elle **conserve sa dernière valeur** jusqu'à une éventuelle modification.

– Exemple –

```
void fct(int a) {
    static int s = 3;
    s = s + a;
    printf("%d\n", s);
}
...
fct(1);
fct(5);
fct(2);
```

Produit l'affichage

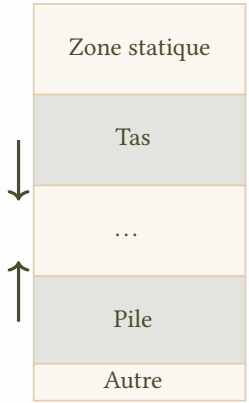
`4`
`9`
`11`.

Ici, `s` est une variable statique. La valeur `3` (initialisation) ne lui est attribuée qu'une fois, lors de la compilation.

De plus, `s` conserve sa valeur (qui peut être modifiée) au fil de l'exécution, d'appel en appel à `fct`.

Géographie de la mémoire lors d'une exécution

Lors de l'exécution d'un programme, la mémoire qui lui est consacrée est divisée de la manière suivante.



Les zones de la mémoire réservées à la pile et au tas ont des tailles variables durant l'exécution.

Le tas grandit vers le bas (vers les adresses hautes) et la pile grandit vers le haut (vers les adresses basses).

La zone statique, qui contient les variables statiques et le code du programme, est de **taille constante durant l'exécution**.

Cette taille est connue et calculée lors de la compilation.

Compter le nombre d'appels à une fonction

Étant donnée une fonction, nous souhaitons **compter** le nombre de fois qu'elle a été **appelée** depuis le lancement du programme.

```c
int fact(int n) {
    if (n <= 1)
        return 1;
    return n * fact(n - 1);
}
```

Fonction originale.

```c
int fact(int n) {

    /* Mecanisme de comptage */
    static int nb_app = 0;
    nb_app += 1;

    if (n <= 1)
        return 1;
    return n * fact(n - 1);
}
```

Fonction modifiée.

À chaque instant, la variable statique `nb_app` a pour valeur le nombre de fois que `fact` a été appelée.

Problème : cette variable a pour portée lexicale le corps de la fonction `fact` et est invisible ailleurs. On ne peut donc pas la lire depuis l'extérieur.

Compter le nombre d'appels à une fonction

Pour avoir accès à la valeur de `nb_app` hors du corps de `fact`, nous la transmettons à l'aide d'un **passage par adresse**.

```
int fact(int n, int *nb) {

    /* Mecanisme de comptage */
    static int nb_app = 0;
    nb_app += 1;
    *nb = nb_app;

    if (n <= 1)
        return 1;
    return n * fact(n - 1, nb);
}
```

Nous l'utilisons de la manière suivante :

```
int nb;
fact(6, &nb);
printf("%d\n", nb);
```

Ceci affiche `6`.

11. Mémoïsation

Principe de la mémoïsation

Soit `f` une fonction **sans effet secondaire** et **déterministe** (sa valeur de retour ne dépend que de ses arguments).

Observation : si l'on appelle `f` deux fois avec des mêmes arguments, elle renverra deux fois la même valeur.

Conséquence : comme il est inutile (et inefficace) de refaire deux fois un même calcul, il suffit de **mémoriser le résultat** renvoyé par `f` lors du 1er appel et de le renvoyer sans calcul lors du 2e.

Marche à suivre : on enregistre toutes les valeurs de retour de `f` au fur et à mesure de ses appels.

Ce procédé s'appelle la mémoïsation de fonction.

Détails sur la mémoïsation

Les résultats renvoyés par `f` sont mémorisés dans une **table de hachage** `mem` associant à chaque jeu d'arguments avec lequel elle est appelée la valeur de retour de `f`.

Il est nécessaire de disposer d'une **fonction de hachage** `h` associant à chaque jeu d'arguments un entier.

Il ne doit y avoir qu'une seule table `mem` pour `f`, dont la durée de vie est égale à l'exécution du programme et rémanente : `mem` est ainsi une **variable statique**, locale à `f`.

Procédé général de mémoïsation d'une fonction

Soit `f` une fonction à `n` paramètres.

Nous notons `f'` la version mémoïsée de `f`, définie de la manière suivante.

Lors de l'appel à `f'(a_1, ..., a_n)` :

(1) si `mem` contient une donnée associée à la clé `(a_1, ..., a_n)`,
   (a) renvoyer cette valeur ;

(2) sinon,
   (a) calculer la valeur de retour de `f(a_1, ..., a_n)` ;
   (b) **mémoriser** cette valeur dans `mem` ;
   (c) renvoyer cette valeur.

L'étape (1) est celle de lecture et l'étape (2) est celle de mémorisation.

Exemple de mémoïsation d'une fonction à un paramètre

– Exemple (début) –

Nous souhaitons mémoïser la fonction

```
int fibo(int n) {
    if (n <= 1)
        return n;
    return fibo(n - 1) + fibo(n - 2);
}
```

Cette fonction est paramétrée par un entier et renvoie un entier. La table `mem` est ainsi un tableau d'entiers. La valeur de hachage d'un argument `n` est `n`.

Nous l'utilisons de la manière suivante :

1. elle est de taille `T_MEM` où `T_MEM` est une constante suffisamment grande ;
2. pour tout $0 \leqslant n < \texttt{T\_MEM}$, la valeur de `mem[n]` est
   - `-1` si `fibo(n)` n'a pas encore été appelée ;
   - la valeur renvoyée par `fibo(n)` si `fibo` a été appelée au moins une fois avec l'argument `n`.

Exemple de mémoïsation d'une fonction à un paramètre

– Exemple (suite et fin) –

La fonction `fibo` peut se mémoïser de la façon suivante.

```
int fibo(int n) {
    /* Table */
    static int mem[T_MEM];

    /* Booleen 1er appel */
    static int init = 1;

    int res;

    /* Initialisation 1er appel */
    if (init) {
        for (int i = 0; i < T_MEM; ++i)
            mem[i] = -1;
        init = 0;
    }
    /* Lecture */
    if (mem[n] != -1)
        return mem[n];

    /* Calcul */
    if (n <= 1)
        res = n;
    else
        res = fibo(n - 1) + fibo(n - 2);

    /* Memorisation */
    mem[n] = res;

    return res;
}
```

Mémoïsation de fonctions à plusieurs paramètres

Soit `f` une fonction à `n` paramètres.

La table `mem` devient un **tableau de variables d'un type structuré** `E_f` composé

1. d'un champ pour chaque **paramètre** de `f` ;
2. d'un champ pour le **résultat** renvoyé par `f` appelé avec les arguments spécifiés par les précédents champs ;
3. d'un champ **booléen** indiquant si le résultat de `f` appelée avec les arguments spécifiés par les précédents champs a bien été calculé.

Il faut de plus disposer d'une **fonction de hachage** `h_f` de même signature que `f` et qui renvoie un entier positif.

Elle permet d'attribuer à chaque jeu d'arguments une position dans la table `mem`.

Exemple de mémoïsation d'une fonction à deux paramètres

– Exemple (début) –

Nous souhaitons mémoïser la fonction

```
int puiss(int n, int p) {
    if (p == 0)
        return 1;
    return n * puiss(n, p - 1);
}
```

Cette fonction est paramétrée par deux entiers. La table `mem` est ainsi un tableau de valeurs de type `E_puiss`.

Le type structuré `E_puiss` est donc défini par

```
typedef struct {
    int n; /* 1er parametre */
    int p; /* 2e parametre */
    int res; /* Resultat */
    int ok; /* Indique si le calcul a deja ete fait */
} E_puiss;
```

On utilise la fonction de hachage suivante :

```
int h_puiss(int n, int p) {
    return n ^ (p << 2);
}
```

Exemple de mémoïsation d'une fonction à deux paramètres

– Exemple (suite et fin) –

La fonction `puiss` peut se mémoïser de la façon suivante.

```
int puiss(int n, int p) {
    /* Table */
    static E_puiss mem[T_MEM];

    /* Booleen 1er appel */
    static int init = 1;

    /* Initialisation 1er appel */
    if (init) {
        for (int i = 0; i < T_MEM; ++i)
            mem[i].ok = 0;
        init = 0;
    }

    /* Lecture */
    int h = h_puiss(n, p) % T_MEM;
    if (mem[h].ok) {
        if (mem[h].n == n && mem[h].p == p)
            return mem[h].res;
    }

    /* Calcul */
    int res;
    if (p == 0) res = 1;
    else res = n * puiss(n, p - 1);

    /* Memorisation */
    mem[h].n = n;
    mem[h].p = p;
    mem[h].res = res;
    mem[h].ok = 1;

    return res;
}
```

12. Génération aléatoire

Plan

12. Génération aléatoire

Générateurs aléatoires

Un générateur aléatoire sur un ensemble E d'objets est une fonction g à valeur dans E **non déterministe**. La valeur renvoyée par g n'est pas nécessairement la même d'un appel à un autre.

On cherche dans la mesure du possible à faire en sorte que, étant donné un ensemble E d'objets, tout objet de E ait la même probabilité d'être renvoyé par le générateur aléatoire g. On parle alors de générateur aléatoire uniforme.

– Exemple –

Si E est l'ensemble des faces d'un dé à six faces, le procédé g qui consiste à lancer le dé et renvoyer la face visible constitue un générateur aléatoire uniforme (et non uniforme si le dé est pipé).

Intérêt des générateurs aléatoires

Les générateurs aléatoires offrent de nombreuses applications. Ils sont utilisés pour :

1. les **méthodes de Monte-Carlo** pour obtenir des solutions approchées à des problèmes ;
2. **tester des algorithmes** en générant des entrées de manière aléatoire (permutations, listes, arbres, graphes, *etc.*) ;
3. **rompre le caractère déterministe** d'un programme en y incluant des éléments imprévisibles (dans les jeux par exemple) ;
4. générer des **mots de passe** ou des **clés de chiffrement**.

Universalité des générateurs aléatoires d'entiers

Un générateur aléatoire d'entiers d'ordre n est un générateur aléatoire sur l'ensemble $\{0, 1, 2, \ldots, n-1\}$.

Pour construire un générateur aléatoire sur un ensemble E d'objets, il suffit en pratique d'avoir un générateur aléatoire d'entiers d'ordre $\#E$.

En effet, il suffit de placer les objets (ou mieux : les adresses des objets) de E dans un tableau

e_0	e_1	$\ldots$	e_{n-1}

,

d'appeler g et de renvoyer l'objet du tableau figurant à l'indice spécifié par l'entier généré par g.

Ainsi, pour faire de la génération aléatoire d'objets, il est suffisant (en 1re approximation) de **savoir construire des générateurs aléatoires d'entiers** d'un ordre suffisamment grand.

Réduire l'ordre d'un générateur aléatoire d'entiers

L'opérateur « **modulo** » offre un moyen très simple pour **réduire l'ordre** d'un générateur aléatoire d'entiers.

En effet, supposons que l'on dispose d'un générateur aléatoire d'entiers g d'ordre n. On souhaite obtenir un générateur aléatoire d'entiers h d'ordre k avec $1 \leqslant k \leqslant n$. On pose pour cela

$$h := g \mod k.$$

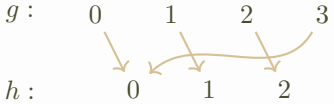
Il est clair que les entiers générés par h sont dans l'ensemble $\{0, \ldots, k-1\}$. On a ainsi réduit l'ordre du générateur g.

On dit que h est la réduction à l'ordre k de g.

Réduction de l'ordre et uniformité

Supposons que g soit un générateur aléatoire uniforme d'ordre $n := 4$ et considérons la réduction h à l'ordre $k := 3$ de g.

On a la situation suivante :



Il y a ainsi deux manières de générer 0 par h (probabilité de $\frac{1}{2}$) alors qu'il n'y en a qu'une seule pour 1 et 2 (probabilités resp. de $\frac{1}{4}$).

Ceci montre que le générateur aléatoire h **perd la propriété d'uniformité**.

Mesure de la non uniformité de la réduction

Soient g un générateur aléatoire uniforme d'ordre n et h la réduction de g à l'ordre k, où $k \leqslant n$.

Notons $\mathrm{gen}_h(i)$ le nombre de manières de générer l'entier $i \leqslant k-1$ par h. Alors, on voit facilement que

$$\mathrm{gen}_h(i) = \begin{cases} \lfloor \frac{n}{k} \rfloor + 1 & \text{si } i \in \{0, \ldots, (n \mod k) - 1\} \\ \lfloor \frac{n}{k} \rfloor & \text{sinon.} \end{cases}$$

La probabilité $\mathbb{P}_h(i)$ de générer l'entier $i \leqslant k-1$ par h vérifie

$$\mathbb{P}_h(i) = \frac{\mathrm{gen}_h(i)}{n}.$$

Les entiers i de l'ensemble $\{0, \ldots, k-1\}$ ont ainsi des probabilités différentes d'être générés. Ils se répartissent en **deux classes** :

1. les petits, de 0 à $(n \mod k) - 1$, avec une probabilité plus grande ;
2. les grands, de $(n \mod k)$ à $k-1$, avec une probabilité plus petite.

Mesure de la non uniformité de la réduction

Comparons numériquement les probabilités de génération des petits et des grands nombres pour différentes valeurs de n et de k :

n	k	**Prob. des petits**	**Prob. des grands**	**Rapport**
4	3	0.5	0.25	2
400	300	0.005	0.0025	2
400	200	0.005	0.005	1
400	101	0.01	0.0075	$\simeq 1.3333$
400	99	0.0125	0.01	1.25
500	99	0.012	0.01	1.2
5000	99	0.0102	0.01	1.02
50000	99	0.01012	0.0101	$\simeq 1.00198$

On observe que plus k est **petit** par rapport à n, plus h se rapproche d'un générateur **uniforme**. Lorsque k est un diviseur de n, l'uniformité est immédiate.

En pratique, la réduction est considérée comme préservant l'uniformité, à condition de partir d'un générateur aléatoire d'entiers uniforme g d'ordre le plus grand possible.

Augmentation de l'ordre d'un générateur

S'il est ainsi préférable de disposer de générateurs aléatoires d'entiers uniformes d'ordres les plus grands possibles, les **générateurs de petits ordres** sont tout aussi intéressants.

Soit en effet g un générateur aléatoire d'entiers uniforme et d'ordre 2. Pour tout entier $\ell \geqslant 1$, on note $g^{(\ell)}$ le générateur aléatoire d'entiers défini par

$$g^{(\ell)} := \sum_{i=0}^{\ell-1} g \times 2^i.$$

Intuitivement, $g^{(\ell)}$ créé une suite de ℓ **bits** u en appelant ℓ fois g. L'entier ainsi construit est celui qui possède u comme écriture en base deux.

La plus petite valeur générée par $g^{(\ell)}$ est 0 et la plus grande est $2^\ell - 1$.

Ainsi $g^{(\ell)}$ est un générateur d'ordre 2^ℓ. Il est de plus uniforme.

Plan

12. Génération aléatoire

Machines et aléatoire

Un ordinateur est par essence une **machine déterministe** : le résultat d'un programme est toujours le même s'il est exécuté deux fois dans les mêmes conditions.

Il est ainsi impossible d'écrire un programme qui implante un générateur aléatoire d'entiers.

La seule chose qu'il est possible de faire est de programmer une fonction qui **semble** le plus possible se comporter comme un générateur aléatoire. On parle alors de générateur pseudo-aléatoire.

Les entiers qu'un générateur pseudo-aléatoire génère lorsqu'on l'appelle plusieurs fois de suite forme une suite. On appelle cette suite une suite d'entiers pseudo-aléatoires.

Principe de fonctionnement

Un générateur pseudo-aléatoire g d'ordre n fonctionne de la manière suivante :

- il dispose d'une graine $g_1, g_2, \ldots, g_\ell$, qui est une suite d'entiers ;
- lorsqu'on l'appelle pour la r^{e} fois, il renvoie un entier $g_{\ell+r}$ calculé à partir des entiers $g_1, \ldots, g_\ell, \ldots, g_{\ell+r-1}$ précédemment calculés, selon une règle spécifiée.

Ainsi, g produit une suite d'entiers

$$g_{\ell+1}, g_{\ell+2}, g_{\ell+3}, \ldots$$

à partir de la graine dans laquelle figurent les **données initiales** et de la règle qui explique comment générer le **prochain entier** de la suite en se basant sur les **entiers précédemment générés**.

Qualités d'un générateur pseudo-aléatoire

Un **bon** générateur pseudo-aléatoire produit une suite d'entiers qui **semble aléatoire**.

Il est possible de mesurer la qualité d'un générateur aléatoire conformément à plusieurs critères :

1. **uniformité** de la génération ;
2. ses **périodes** (longueurs des cycles de génération) ;
3. **sensibilité à la graine** (existence de mauvaises graines) ;
4. **corrélation** entre un entier engendré et les précédents ;
5. **analyse de la distribution** en plusieurs dimensions.

Méthode du carré médian

Le générateur pseudo-aléatoire g utilisant la méthode du carré médian fonctionne de la manière suivante.

La graine de ce générateur est un entier compris entre 0 et 9999. Si g_{i-1} est l'entier précédemment généré (ou bien la graine), le prochain entier généré est

$$g_i := \left(\left\lfloor \frac{g_{i-1}}{10} \right\rfloor \mod 100 \right)^2.$$

– Exemples –

Graine	Suite
1234	529, 2704, 4900, 8100, 100, 100, ...
3733	5329, 1024, 4, 0, 0, ...
9999	9801, 6400, 1600, 3600, 3600, ...

C'est un **mauvais générateur pseudo-aléatoire**. Les périodes sont bien trop courtes et il y a des mauvaises graines (comme 0).

Méthode additive

Le générateur pseudo-aléatoire g utilisant la méthode additive fonctionne de la manière suivante.

On se fixe un ordre $n \geqslant 1$. La graine de ce générateur est un triplet (g_1, g_2, g_3) d'entiers compris entre 0 et $n-1$. La génération de g_i dépend des trois entiers précédemment générés g_{i-1}, g_{i-2} et g_{i-3}. Il est défini par

$$g_i := (g_{i-1} + g_{i-2} + g_{i-3}) \mod n.$$

– Exemples –

n	Graine	Suite
211	$(1, 1, 1)$	3, 5, 9, 17, 31, 57, 105, 193, 144, 20, 146, 99, 54, 88, 30, 172, 79
3099	$(1, 1, 1)$	3, 5, 9, 17, 31, 57, 105, 193, 355, 653, 1201, 2209, 964, 1275, 1349
1347	$(600, 31, 1)$	632, 1263, 1148, 349, 66, 216, 631, 913, 413, 610, 589, 265, 117

Ce générateur pseudo-aléatoire est bien meilleur que le précédent mais **échoue à de nombreux tests statistiques**.

Générateurs congruentiels linéaires

Un générateur pseudo-aléatoire g congruentiel linéaire fonctionne de la manière suivante.

On se fixe un ordre $n \geqslant 1$ ainsi que deux entiers positifs a et b. La graine de ce générateur est un entier compris entre 0 et $n-1$. La génération de g_i dépend de l'entier précédemment généré g_{i-1} et est défini par

$$g_i := (a \times g_{i-1} + b) \mod n.$$

– Exemples –

n	a	b	**Graine**	**Suite**
128	3	1	1	4, 13, 40, 121, 108, 69, 80, 113, 84, 125, 120, 105, 60
128	7	1	1	8, 57, 16, 113, 24, 41, 32, 97, 40, 25, 48, 81, 56, 9, 64
2^{32}	1664525	1013904223	1	1015568748, 1586005467, 2165703038, 3027450565

Sous réserve de **choisir des bons paramètres** n, a et b, ce générateur pseudo-aléatoire est **plutôt bon**. Il est très utilisé en pratique.

Plan

12. Génération aléatoire

Les fonctions `rand` et `srand`

La fonction

```
int rand();
```

du module `stdlib` implante un générateur pseudo-aléatoire d'ordre `RAND_MAX + 1`.

C'est un **générateur congruentiel linéaire**.

À chaque appel, il renvoie le prochain entier de la suite.

La **graine** de ce générateur peut-être **affectée** par la fonction

```
void srand(unsigned int g1);
```

Par défaut, la graine du générateur est `1`.

Modèle d'utilisation de `rand` et de `srand`

Tout projet qui utilise la fonction `rand` doit avoir une fonction principale `main` de la forme ci-contre.

L'appel à `srand` doit être fait le plus tôt possible.

```c
/* Main.c */

#include <stdlib.h>
#include <time.h>
...
int main() {
    ...
    srand(time(NULL));
    ...
}
```

La fonction

```c
time_t time(time_t *timer);
```

du module `time`, lorsque appelée avec l'argument `NULL` renvoie le temps écoulé en secondes depuis le 1er janvier 1970 à minuit, UTC.

Considérer cette valeur est donc un bon moyen d'initialiser la graine du générateur.

Attention (*point très important*) : il est **totalement erroné** d'**initialiser** dans un même projet **plusieurs fois la graine**. Le seul appel à `srand` doit figurer dans `main` et nulle part ailleurs.

Implantation avec variable globale

Une implantation possible des fonctions `rand` et `srand` s'appuie sur une **variable globale** pour mémoriser la dernière valeur générée.

```c
/* RandX.h */

#ifndef RAND_X_H
#define RAND_X_H

    #define N 32768
    #define A 1024
    #define B 1
    #define G_INIT 1

    int randx();
    void srandx(int graine);

#endif
```

```c
/* RandX.c */

#include "RandX.h"

unsigned int g = G_INIT;

int randx() {
    g = (g * A + B) % N;
    return g;
}

void srandx(int graine) {
    g = graine;
}
```

Implantation avec variable statique

Dans cette implantation, on utilise une **variable statique** (voir le chapitre suivant où cette notion sera approfondie) pour mémoriser la dernière valeur générée.

```c
/* RandY.h */

#ifndef RAND_Y_H
#define RAND_Y_H

    #define N 32768
    #define A 1024
    #define B 1

    int randy();

#endif
```

```c
/* RandY.c */

#include "RandY.h"

#include <time.h>

int randy() {
    static int premier = 1;
    static unsigned int g;

    if (premier) {
        g = time(NULL);
        premier = 0;
    }
    g = (g * A + B) % N;
    return g;
}
```

L'initialisation ne se fait qu'au 1er appel et l'utilisateur n'a pas à la faire explicitement. Avec cette méthode, il n'est pas possible de fixer la graine.

Retrouver le déterminisme

Pour reproduire les résultats fournis par un programme utilisant `rand`, il est nécessaire de proposer un **mode de fonctionnement déterministe** dans lequel la graine est fixée et connue.

Pour cela, on met en place dans le module principal d'un projet et dans sa fonction `main` le mécanisme suivant :

```
/* Main.c */

#define DETERMINISTE
...
int main() {
    ...
    #ifdef DETERMINISTE
        srand(0);
    #else
        srand(time(NULL));
    #endif
    ...
}
```

De cette manière, on peut imposer un comportement déterministe (**reproductible**) à l'exécution du projet en gardant la définition de la macro `DETERMINISTE`.

Pour éviter ce comportement, il suffit de commenter cette macro-définition.

Meilleure solution : utiliser l'option `-D` de `gcc`. Il suffit de supprimer la ligne `#define DETERMINISTE` et de compiler avec `-DDETERMINISTE` pour définir la macro en question juste avant la compilation.