# Notes du cours 4M056
# Programmation en `C` et `C++`

Vincent Lemaire

19 janvier 2014

# Organisation du cours 4M056

Ce cours a pour but de vous donner une base solide de programmation en `C` et des connaissances de programmation en `C++` en insitant sur 2 concepts présents dans de nombreux langages modernes : la programmation orienté objet et la programmation générique. On insistera sur la différence entre ces 2 langages en découpant le semestre en 2 parties, la première dédiée au `C` et la seconde au `C++`. Le cours mettra l'accent sur la compréhension des mécanismes de ces langages (mécanisme d'appel des fonctions, pointeurs, structures, objets, héritage,...) et donnera les principales règles de syntaxe de ces langages mais ne pourra pas être exhaustif étant donné le temps imparti. Ainsi certaines fonctions des librairies standards ne seront pas vues en cours ni utilisées en TP.

Le nombre d'étudiants inscrits est limité à 90 étudiants. Il y a 3 groupes de TP et une répartition uniforme des étudiants sur ces 3 groupes est indispensable pour le bon fonctionnement des TP. L'inscription dans l'un des 3 groupes est obligatoire et nécessaire pour valider ce module. Les groupes ont lieu :

— lundi de 13h à 16h : Maison de la Pédagogie B006
— mercredi de 13h45 à 16h45 : Maison de la Pédagogie B006
— jeudi de 13h45 à 16h45 : Maison de la Pédagogie B006

**Les séances de TP sur machine sont obligatoires.** Elles donnent lieu à une note (à travers 2 séances de TP notés) qui compte pour 25% dans la note finale.

### Calendrier prévisionnel pour l'année 2013-2014

— 13/01 au 17/02 : 5/6 cours de `C`
— TP notés en `C` : semaine du 24/02
— 17/02 au 31/03 : 6/7 cours de `C++`
— TP notés en `C++` : semaine du 07/04

### Calcul de la note finale (sur 100)

$$\text{note finale} = \frac{1}{4}\left(\frac{\text{note du TP1} + \text{note du TP2}}{2}\right) + \frac{3}{4}(\text{note examen})$$

*Les notes de cours qui suivent sont en cours de rédaction et ne contiennent pas tous les détails et commentaires vus en cours. Vous trouverez en priorité le plan du cours et les programmes d'exemples. Ce document sera régulièrement mis à jour.*

# Première partie

# Programmation en C

# Chapitre 1

# Introduction

On commence ce cours par 2 citations (d'actualité lors de l'écriture de ce polycopié) à propos de Dennis Ritchie, disparu une semaine après Steve Jobs en octobre 2011.

— *citation de la page Wikipedia de Dennis Ritchie*

> Dennis MacAlistair Ritchie, né le 9 septembre 1941 à Bronxville dans l'État de New York et trouvé mort le 12 octobre 2011 à Berkeley Heights dans le New Jersey, est un des pionniers de l'informatique moderne, inventeur du langage `C` et co-développeur de `UNIX`. Il est parfois désigné par *dmr*, son adresse e-mail aux Laboratoires Bell.
>
> Au début des années 1970, il travaille avec Ken Thompson en tant que programmeur dans les Laboratoires Bell sur le développement d'`UNIX`. Pour `UNIX`, il s'avère nécessaire d'améliorer le langage `B` créé par Ken Thompson et c'est ainsi que Ritchie crée le langage `C`. Par la suite, aidé de Brian Kernighan, il promeut le langage et rédige notamment le livre de référence *The C Programming Language*.
>
> Il reçoit conjointement avec Ken Thompson le prix Turing de l'ACM en 1983 pour leur travail sur le système `UNIX`.

— *citation de l'article de Cade Metz dans Wired*[1]

> "When Steve Jobs died last week, there was a huge outcry, and that was very moving and justified. But Dennis had a bigger effect, and the public doesn't even know who he is," says Rob Pike, the programming legend and current Googler who spent 20 years working across the hall from Ritchie at the famed Bell Labs.
>
> On Wednesday evening, with a post to Google+, Pike announced that Ritchie had died at his home in New Jersey over the weekend after a long illness, and though the response from hardcore techies was immense, the collective eulogy from the web at large doesn't quite do justice to Ritchie's sweeping influence on the modern world. Dennis Ritchie is the father of the `C` programming language, and with fellow Bell Labs researcher Ken Thompson, he used `C` to build `UNIX`, the operating system that so much of the world is built on — including the Apple empire overseen by Steve Jobs.
>
> "Pretty much everything on the web uses those two things : `C` and `UNIX`," Pike tells Wired. "The browsers are written in `C`. The `UNIX` kernel — that pretty much the entire Internet runs on — is written in `C`. Web servers are written in `C`, and if they're not, they're written in `Java` or `C++`, which are `C` derivatives, or `Python` or `Ruby`, which are implemented in `C`. And all of the network hardware running these programs I can almost guarantee were written in `C`.
>
> "It's really hard to overstate how much of the modern information economy is built on the work Dennis did."
>
> Even Windows was once written in `C`, he adds, and `UNIX` underpins both Mac OS X, Apple's desktop operating system, and `iOS`, which runs the iPhone and the iPad. "Jobs was the king of the visible, and Ritchie is the king of what is largely invisible," says

1. http://www.wired.com/wiredenterprise/2011/10/thedennisritchieeffect/

Martin Rinard, professor of electrical engineering and computer science at MIT and a member of the Computer Science and Artificial Intelligence Laboratory.

"Jobs' genius is that he builds these products that people really like to use because he has taste and can build things that people really find compelling. Ritchie built things that technologists were able to use to build core infrastructure that people don't necessarily see much anymore, but they use everyday."

**From B to C**

Dennis Ritchie built C because he and Ken Thompson needed a better way to build UNIX. The original UNIX kernel was written in assembly language, but they soon decided they needed a "higher level" language, something that would give them more control over all the data that spanned the OS. Around 1970, they tried building a second version with Fortran, but this didn't quite cut it, and Ritchie proposed a new language based on a Thompson creation known as B.

Depending on which legend you believe, B was named either for Thompson's wife Bonnie or BCPL, a language developed at Cambridge in the mid-60s. Whatever the case, B begat C.

B was an interpreted language — meaning it was executed by an intermediate piece of software running atop a CPU — but C was a compiled language. It was translated into machine code, and then directly executed on the CPU. But in those days, C was considered a high-level language. It would give Ritchie and Thompson the flexibility they needed, but at the same time, it would be fast.

That first version of the language wasn't all that different from C as we know it today — though it was a tad simpler. It offered full data structures and "types" for defining variables, and this is what Richie and Thompson used to build their new UNIX kernel. "They built C to write a program," says Pike, who would join Bell Labs 10 years later. "And the program they wanted to write was the UNIX kernel."

Ritchie's running joke was that C had "the power of assembly language and the convenience of… assembly language." In other words, he acknowledged that C was a less-than-gorgeous creation that still ran very close to the hardware. Today, it's considered a low-level language, not high. But Ritchie's joke didn't quite do justice to the new language. In offering true data structures, it operated at a level that was just high enough.

"When you're writing a large program — and that's what UNIX was — you have to manage the interactions between all sorts of different components : all the users, the file system, the disks, the program execution, and in order to manage that effectively, you need to have a good representation of the information you're working with. That's what we call data structures," Pike says.

"To write a kernel without a data structure and have it be as consist and graceful as UNIX would have been a much, much harder challenge. They needed a way to group all that data together, and they didn't have that with Fortran."

At the time, it was an unusual way to write an operating system, and this is what allowed Ritchie and Thompson to eventually imagine porting the OS to other platforms, which they did in the late 70s. "That opened the floodgates for UNIX running everywhere," Pike says. "It was all made possible by C."

**Apple, Microsoft, and Beyond**

At the same time, C forged its own way in the world, moving from Bell Labs to the world's universities and to Microsoft, the breakout software company of the 1980s. "The development of the C programming language was a huge step forward and was the right middle ground… C struck exactly the right balance, to let you write at a high level and be much more productive, but when you needed to, you could control exactly what happened," says Bill Dally, chief scientist of NVIDIA and Bell Professor of Engineering at Stanford. "[It] set the tone for the way that programming was done for several decades."

As Pike points out, the data structures that Richie built into C eventually gave rise

to the object-oriented paradigm used by modern languages such as C++ and Java.

The revolution began in 1973, when Ritchie published his research paper on the language, and five years later, he and colleague Brian Kernighan released the definitive C book : *The C Programming Language*. Kernighan had written the early tutorials for the language, and at some point, he "twisted Dennis' arm" into writing a book with him.

Pike read the book while still an undergraduate at the University of Toronto, picking it up one afternoon while heading home for a sick day. "That reference manual is a model of clarity and readability compared to latter manuals. It is justifiably a classic," he says. "I read it while sick in bed, and it made me forget that I was sick."

Like many university students, Pike had already started using the language. It had spread across college campuses because Bell Labs started giving away the UNIX source code. Among so many other things, the operating system gave rise to the modern open source movement. Pike isn't overstating it when says the influence of Ritchie's work can't be overstated, and though Ritchie received the Turing Award in 1983 and the National Medal of Technology in 1998, he still hasn't gotten his due.

As Kernighan and Pike describe him, Ritchie was an unusually private person. "I worked across the hall from him for more than 20 years, and yet I feel like a don't knew him all that well," Pike says. But this doesn't quite explain his low profile. Steve Jobs was a private person, but his insistence on privacy only fueled the cult of personality that surrounded him.

Ritchie lived in a very different time and worked in a very different environment than someone like Jobs. It only makes sense that he wouldn't get his due. But those who matter understand the mark he left. "There's that line from Newton about standing on the shoulders of giants," says Kernighan. "We're all standing on Dennis' shoulders."

## 1.1 Langage de programmation

Un programme désigne un fichier texte (que l'on écrit avec un éditeur de texte et non un traitement de texte) composé d'instructions qui doivent respecter des règles de syntaxe. Chaque langage de programmation a ses propres instructions et règles de syntaxes. Un outil lié au langage de programmation est utilisé pour transformer ce programme texte en une suite de blocs de 8, 16, 32 ou 64 bits. Chaque bloc est appelé un *mot machine* et la suite de ces blocs constitue un programme en langage machine executable par la machine. Bien entendu, ce programme machine est constitué d'une suite de 0 et de 1 et est incompréhensible pour le programmeur.

De façon schématique, il existe deux outils permettant de faire executer un programme par la machine

- un interpréteur : qui à chaque execution parcourt le fichier texte (appelé dans ce cas *script*) et traduit la suite des instructions en langage machine aussitôt executé,
- un compilateur : qui traduit une seule fois le fichier texte (appelé dans ce cas *fichier source*) en un fichier machine (fichier *objet*) executable.

Un programme dans un langage interprété est souvent plus facile à apprendre, à écrire, à corriger et à maintenir mais sera plus long à l'execution (car la phase d'interprétation peut être coûteuse en temps). De nombreux logiciels scientifiques possèdent leur propre langage de programmation interprété, par exemple : scilab, matlab, mathematica, mapple. De même, on peut citer les langages très utilisés actuellement comme : Perl, Python, Ruby, Lua et des langages plus spécialisés comme javascript, sql, php, etc.

La compilation permet la transformation directe d'un fichier source en un fichier objet compréhensible par la machine. Ce fichier objet peut être en lien avec d'autres fichiers objets et le tout forme ce qu'on appelle communément un programme ou une application. Des fichiers objets peuvent aussi être regroupés pour former ce qu'on appelle une librairie (par exemple un fichier .dll sous windows ou .a ou .so sur des systèmes UNIX). Une librairie est un ensemble de codes qui peut être appelé (utilisé) dans un autre programme.

Les langages C et C++ sont des langages compilés et le compilateur utilisé dans ce cours est le

GNU Compiler Collection ou GCC disponible sur toutes les architectures que vous pouvez utiliser. De nombreux langages compilés existent et ont tous leurs propres usages. On peut citer Fortran, Pascal, Java, OCaml, ObjectiveC, etc.

## 1.2 Evolution du langage C

Le langage C a été « modernisé » depuis sa création il y a environ 40 ans. Le C classique, aussi connu sous le nom de C K&R, se réfère au livre fondateur de Kernighan et Ritchie intitulé *The C Programming Language* paru en 1978. Le langage a ensuité été normalisé en 89 et donne lieu au ANSI C qui correspond au standard utilisé par tous les compilateurs sur toutes les architectures (ordinateurs personnels, supercalculateurs, puces spécialisées, etc.). Cette norme a depuis évolué 2 fois : en 99 ce qui a donné lieu au C99 et l'année dernière en 2011. La norme C99 commence a être intégrée par tous les compilateurs (entièrement prise en charge dans gcc) et la norme C11 qui ajoute de nombreuses fonctionnalités n'est pas encore bien implémentée. Dans ce cours, on présente le ANSI C avec de nombreuses parenthèses sur le C99.

— création en 1972-1973 et popularisé en 1978 : C K&R
— normalisation en 1989 : ANSI C
— normalisation en 1999 : C99
— normalisation en 2011 : C11

## 1.3 Phases de compilation

Ce qu'on appelle compilation est en fait composée de 3 phases :

— le préprocesseur : mécanisme rudimentaire qui applique des règles pour préparer le code : inclusion de fichiers d'en-tête (header) contenant la déclaration de fonctions, déclaration de constante et de macros. Ce mécanisme de règles ne fait pas partie du langage C : il n'y a pas de notion d'instruction (donc pas de « ; »), pas de variables, pas de fonctions, etc. Moralement, on peut voir ce mécanisme comme un copié/collé automatisé. Ce mécanisme rudimentaire est néanmoins très efficace et est utilisé par d'autres langages comme le C++.
De nombreux codes écrits dans les années 80 utilisent des macros pour optimiser le code. Dans ce cours, on n'autorisera pas l'utilisation de ces macros qui peuvent produire des erreurs importantes (cf. plus tard...)
Pour information, le résultat de cette phase est accessible en utilisant le compilateur gcc avec les options -E et -P (pour plus de lisibilité).
— la compilation et l'assemblage : c'est la phase principale où le code source est vérifié et si aucune erreur de syntaxe n'est détectée il est traduit en code objet. Il y a donc pour chaque fichier source .c un fichier objet .o créé. Plus précisément le code source est d'abord traduit en langage assembleur (propre à l'architecture du système) puis ce code assembleur est transformé en fichier binaire.
Ces deux premières phases sont accessibles avec l'option -c de gcc.
— l'édition de liens : mécanisme permettant de faire le liens entre les différents fichiers objets produits et les fichiers objets des fonctions système (c'est à ce moment que le code objet d'une fonction est intégrée au programme). S'il n'y a pas d'erreurs (si le compilateur trouve tous les bouts de code objets dont il a besoin) alors il y a création d'un fichier executable. Ce fichier executable est ce qu'on appelle communément programme ou application.
Par défaut sur un système Linux le fichier executable produit s'appelle a.out. Pour obtenir un nom différent on utilise l'option -o de gcc.
Pour l'utilisation de certaines librairies (l'inclusion de certains fichiers d'en-têtes) il peut être nécessaire d'indiquer un chemin au compilateur qui lui permet de trouver le code objet correspondant. Par exemple pour utiliser le fichier système math.h qui contient la déclaration de fonctions mathématiques (comme sqrt, pow, abs, fabs, etc.) on doit ajouter l'option -lm.

## 1.4 Premier programme en C

Voici à quoi ressemble un programme très simple en C. Le fichier source s'appelle test.c.

*Fichier test.c*

```
1 #include <stdio.h>
2 int main(void)
3 {
4    printf("Bonne annee!");
5    return 0;
6 }
```

### Commentaires

**ligne 1 :** il ne s'agit pas d'une instruction en langage C mais d'une ligne destinée au préprocesseur : on indique que l'on a besoin de fonctions déclarées dans le fichier système stdio.h. Ce fichier d'en-tête (header) contient les fonctions standards d'entrées-sorties (**std**andard **i**nput **o**utput), c'est à dire les fonctions qui lisent ou qui écrivent sur la ligne de commande, dans un fichier, ou plus généralement dans ce qu'on appelle un flux.
Toutes les lignes qui débutent par le caractère **#** dans un programme sont destinées au préprocesseur (cf. annexe pour plus de détails).

**ligne 2 :** il s'agit de la déclaration de la fonction qui s'appelle main qui ne prend aucun argument (**void**) et qui renvoie un entier **int**. Le nom de la fonction main n'est pas choisi par hasard, il fait partie de la syntaxe du langage. Cette fonction main joue un rôle très particulier car c'est la première fonction appelée lors de l'execution du programme, et sa présence est nécessaire pour créer un fichier executable lors de l'édition de liens. Un programme qui ne contient pas de fonction main peut néanmoins être compilé en un fichier objet.

**lignes 3 et 6 :** ouverture et fermeture d'un *bloc* qui contient des instructions, ici les lignes 4 et 5. Ce bloc constitue la définition de la fonction main déclarée à la ligne 2. Ainsi on dira que les lignes 2 à 6 constituent la déclaration et la définition de la fonction main.

**ligne 4 :** appel de la fonction printf avec l'argument "Bonne annee!". L'argument est une chaîne de caractères (au format ASCII c'est à dire sans les caractères accentués). Cette ligne forme une instruction en C (appel d'une fonction avec un argument) et se termine par un point virgule. Ce caractère « ; » est utilisé pour séparer les instructions. Le saut de ligne est facultatif pour le compilateur mais nécessaire pour une bonne lisibilité du code.
La fonction printf affiche la chaîne de caractère donnée en argument sur la ligne de commande qui execute le programme.

**ligne 5 :** le mot-clé **return** fait partie du langage C et doit être suivi d'une expression « compatible » avec ce que renvoie la fonction. Ici on se trouve dans la définition de la fonction main qui renvoie un entier **int** donc le mot-clé **return** doit être suivi d'un entier. Le 0 que l'on renvoie dans cet exemple indique au système d'exploitation que le programme s'est bien déroulé. Dans des programmes plus complexes on peut renvoyer un entier codant pour une erreur particulière (tout cela devant être documenté dans la page de manuel accompagnant le programme).

### Compilation

Pour effectuer uniquement la phase du préprocesseur et voir le code C produit, on peut taper dans un terminal la commande suivante [2] :

```
$ gcc -E -P test.c
```

Si l'on veut sauver le résultat dans un fichier texte test_apres_preproc.c on peut rediriger cette sortie

2. le symbole $ en début de ligne indique qu'il s'agit d'une commande dans un terminal et ne doit pas être recopié

en utilisant l'opérateur > de l'interpréteur de la ligne de commande (bash, zsh ou bien d'autres...). On peut donc taper

```
$ gcc -E -P test.c > test_apres_preproc.c
```

puis éditer ce fichier test_apres_preproc.c pour voir son contenu.

Pour vérifier qu'il n'y a pas d'erreur de syntaxe et que l'appel des fonctions existentes est correct, on effectue la phase de compilation (et d'assemblage) qui produit un fichier objet. Ainsi, on tape la commande

```
$ gcc -c test.c
```

qui produit le fichier test.o s'il n'y a pas d'erreur. C'est à cette étape qu'il faut souvent corriger les erreurs de syntaxe.

Enfin pour créer le fichier executable que l'on appelle ici prog1 on utilise la commande suivante

```
$ gcc test.c -o prog1
```

L'option -o indique le nom du fichier executable créé s'il n'y a pas d'erreur lors de l'édition de liens. Sans cette option, le fichier executable s'appelle a.out sur les systèmes UNIX. Les erreurs qui peuvent se produire lors de cette phase viennent principalement du fait que gcc ne trouve pas les fichiers objets correspondants aux fonctions déclarées dans les fichiers d'en-têtes (fichier header)

Pour executer ce fichier executable sur un système UNIX on tape

```
$ ./prog1
```

## 1.5 Un deuxième programme

*Fichier aire.c*

```
#include <stdio.h>
#include <math.h>
#define PI 3.14159

// declaration et definition la fonction aire_rectangle
double aire_rectangle(double l1, double l2) {
   return l1*l2;
};

/* declaration de la fonction aire_cercle
   puis definition ligne 18 */
double aire_cercle(double rayon);

int main(void) {
   double aire = aire_cercle(0.5 * sqrt(5));
   printf("aire du cercle de diametre racine de 5: %g", aire);
   return 0;
};

double aire_cercle(double rayon) {
   return PI*rayon*rayon;
};
```

### Commentaires

**lignes 1-3 :** 3 lignes destinées au préprocesseur : inclusion des 2 fichiers systèmes `stdio.h` et `math.h` et définition d'une constante `PI` suivie de sa valeur. Attention la « définition » de `PI` constitue un racourci d'écriture pour la suite du programme et non la création d'une variable.

**ligne 5 :** tout ce qui suit `//` est mis en commentaire jusqu'au bout de la ligne.

**lignes 6-8 :** déclaration et définition de la fonction `aire_rectangle` (dont on a choisi le nom) qui prend 2 arguments `l1` et `l2` qui sont des variables de type **`double`** (représentant les nombres réels) et qui renvoie une variable de type **`double`**. La fonction `aire_rectangle` est la représentation d'une fonction $\big(\mathbb{R}\times\mathbb{R}\to\mathbb{R}, (l_1,l_2)\mapsto l_1l_2\big)$.
Cette définition de fonction est suivie d'un « ; » à la ligne `8`.

**lignes 10-11 :** tout le texte entre `/*` et `*/` est mis en commentaire, même les sauts de ligne.

**ligne 12 :** déclaration de la fonction `aire_cercle` qui prend un argument qui s'appelle `rayon` de type **`double`** et qui renvoie un **`double`**.

**lignes 14-18 :** déclaration et définition de la fonction principale

- l.15 : déclaration de la variable `aire` de type **`double`** et initialisation de cette variable avec la valeur `aire_cercle(0.5 * sqrt(5))`. Cette valeur est le résultat de l'appel de la fonction `aire_cercle` avec l'argument `0.5 * sqrt(5))`. De même, `sqrt(5)` est le résultat de l'appel de la fonction `sqrt` (**sq**uare **r**oo**t**) déclarée dans le fichier `math.h`. les appels de fonctions peuvent être imbriqués sans restriction.
  La variable `aire` est dite *locale* car elle est déclarée dans une fonction. Dans ce cours, on utilisera exclusivement des variables locales
- l.16 : appel de la fonction d'affichage `printf` avec 2 arguments. attention il est rare qu'une fonction en `C` puisse s'appeler avec un nombre variable d'arguments (ici 2 et dans le premier programme 1). Ces fonctions à nombre variables d'arguments sont plus complexes à écrire et utilisent le préprocesseur. Dans un premier temps il faut donc voir la fonction `printf` comme un cas particulier de fonction en `C` (cf. annexe pour plus de détails).

**lignes 20 :** définition de la fonction `aire_cercle` précédemment déclarée.

### Compilation

Pour compiler et executer ce programme on utilise alors les commandes suivantes :

```
$ gcc aire.c -o prog2 -lm
$ ./prog2
```

## 1.6 Organisation d'un programme

Comme dans les deux exemples vus précédemment, un programme doit être composé (pas forcément ordonné) de la façon suivante :

- des lignes destinées au préprocesseur précédées d'un **`#`**
- des lignes de commentaires précédées de `//` ou entre `/*` et `*/`
- des déclarations de fonctions qui doivent précédées (dans le code source) leur utilisation.
  Notez que la définition peut se trouver après dans le code. Par exemple la fonction `aire_cercle` est déclarée ligne `12` puis utilisée ligne `15` et enfin définie ligne `20`.
- des définitions de fonctions qui contiennent des instructions séparées par « ; »
- des définitions de types (que nous verons plus tard)

Enfin, un code source peut contenir en dehors de toute fonction des variables déclarées. Ces variables sont appelées variables globales et sont utilisables dans les lignes qui suivent leur déclaration. Dans ce cours, on n'utilisera pas ces variables globales.

De plus, les fonctions ne peuvent pas être imbriquées les unes dans les autres : on en peut pas déclarer et définir une fonction dans une autre fonction. Par contre, les appels peuvent être imbriqués sans restriction.

## 1.7 Outils pour le développeur (non utilisés dans ce cours)

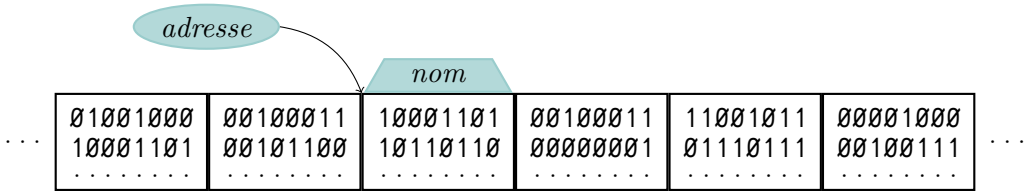
Le compilateur est l'outil essentiel pour le programmeur. Cependant il existe tout un « écosystème » d'outils puissants pour aider le développeur. On peut citer par exemple

- un système de construction qui permet d'automatiser les appels au compilateure et d'éviter les lignes de commande trop longues ou complexes. L'outil historique et encore très utilisé est l'utilitaire `make` qui nécessite l'écriture d'un `Makefile` (cf. annexe pour des exemples). Il existe aussi `scons`.
- un débogueur (*debugger*) qui permet de suivre l'execution d'un programme pour trouver les sources d'erreur (`bugs`). On peut vérifier l'état de la mémoire, de chaque variable du programme, visualiser le code assembleur, etc. Le débogueur du projet `GNU` s'appelle `gdb`.
- un gestionnaire de versions, très utile pour développer un projet à plusieurs : il gère les conflits entre les différents codes, garde des versions antérieures, etc. On peut citer `CVS`, `Subversion` ou `git`.
- un environnement de développement intégré (IDE) qui regroupe dans une interface graphique cohérente un éditeur de texte avec coloration syntaxique (mot-clés du langage en couleur), un système de construction, un débogueur, un gestionnaire de versions, etc. Un exemple d'un tel IDE est `eclipse`.
- un profileur de code qui permet d'aider à optimiser le code en analysant les performances du programme. Par exemple on peut utiliser `gprof` ou `valgrind`.

# Chapitre 2

# Syntaxe du C

On représente la mémoire qui contient les données binaires (en base 2) du programme par le graphe suivant. Chaque case représente une zone mémoire qui contient un ou plusieurs octets (8 bits). Certaines cases possèdent un *nom* et/ou une *adresse* qui permet d'accéder au contenu mémoire (la *valeur*). De plus, à chaque case correspond un *type* qui permet de « décoder » le contenu, c'est à dire connaître la signification des bits.




Les cases qui ne possèdent ni nom, ni adresse (mais bien un type et une valeur) seront appelées *constantes.*

## 2.1 Vocabulaire

Le langage C est un langage fortement typé c'est à dire qu'il faut déclarer le type de chaque variable que l'on veut utiliser dans la suite du programme. De même pour les fonctions qu'il faut déclarer avant de les appeler.

**Définition 1** *Une **variable** est un espace réservé en mémoire qui possède les caractéristiques suivantes :*

- son nom *(unique) qui permet de l'identifier,*
- son type*, c'est la convention d'interprétation de la séquence de bits qui constitue la variable,*
- sa valeur*, c'est la séquence de bits elle-même,*
- son adresse*, c'est l'endroit dans la mémoire où elle est stockée.*

La *déclaration* d'une variable se fait en choisissant un nom précédé d'un ***type*** (le ***type*** est un mot-clé du langage ou défini précédemment dans le programme par le mot-clé **typedef**). Le nom doit être lisible, avoir du sens pour aider la compréhension du programme, composé d'une combinaison de caractères alpha-numériques sans accent (a...z A...Z 0...9) et du caractère « _ ». Le nom d'une variable ne peut pas commencer par un chiffre. La *lecture* ou l'accès à la valeur de la variable se fait simplement en utilisant son nom.

Une variable a une durée de vie qui correspond à l'endroit du code dans laquelle elle existe. Par défaut une variable existe uniquement dans le bloc défini entre accolades { } et elle est accessible et modifiable par les instructions de ce bloc. Ce comportement peut être modifié en utilisant des mot-clés appelés *qualifieurs*. Les plus importants sont **const**, **static** et **extern**. Dans ce cours, on insistera essentiellement sur **const** qui est fondamental. En général, on place le qualifieur devant le type de la variable que l'on déclare. Il ne peut pas changer par la suite.

Pour agir sur ces variables on utilise deux mécanismes : des opérateurs et des appels de fonction[1]. On parle d'*expression* simple pour désigner une constante(`0`, `3.14`, `'A'`, etc.) ou une variable existante (*i.e.* déjà déclarée). Attention, une déclaration n'est pas une expression !

**Définition 2** *Un **opérateur** est un symbole qui agit sur une ou deux expression(s) simple(s) (variable ou constante) pour créer une expression complexe. S'il agit sur une seule expression on parle d'opérateur **unaire** et sinon d'opérateur **binaire**.*

On peut regrouper ces opérateurs de la façon suivante :
- opérateurs arithmétique (binaires) : `+ - * / %`
- opérateurs arithmétique (unaires) : `- ++ --`
- opérateurs sur les bits[2] : `<< >> & | ^ ~`
- opérateurs d'affectation : `= += -= *= /= %=`
- opérateurs de comparaison : `== != < <= > >=`
- opérateurs logiques : `! || &&`
- opérateurs « système » (unaires[3]) : `() [] * & . -> , ?: sizeof() (type simple)`

En `C` il n'est pas possible de définir de nouveaux opérateurs ou de changer le comportement de ces opérateurs.

Dans toute la suite de ce cours on désigne par expression toute combinaison valide (syntaxiquement correcte) d'expressions simples (variables ou constantes), d'opérateurs et d'appels de fonctions.

**Définition 3** *Une **instruction** se termine toujours par un point-virgule et peut-être définie par*
- *une expression suivie par un point-virgule,*

*ou*
- *une combinaison de mot-clés[4] agissant sur des expressions et des instructions.*

*Un **bloc** est un ensemble de déclarations et d'instructions rassemblées entre accolades `{ }`.*

Notations : Dans les codes qui suivent, on notera ***expr*** l'endroit où le programmeur doit mettre une expression isolée, ***instr*** là où l'on attend une instruction ou un bloc d'instructions, et ***type*** pour le type d'une variable.

L'*affectation* permet d'attribuer une nouvelle valeur à une variable. Pour cela on utilise l'opérateur `=` avec à sa gauche le nom de la variable et à sa droite la nouvelle valeur. Lorsqu'on donne une première valeur à la variable on parle d'*initialisation* (on utilise alors le symbole `=` qui n'est pas équivalent à l'opérateur d'affectation `=`).

*Syntaxe de deux blocs qui ont le même effet*

```
{
    type var = expr;      // déclaration et initialisation
}
{
    type var;             // déclaration
    var = expr;           // affection
}
```

On peut enchaîner les déclarations et initialisations éventuelles de variables de même type en utilisant l'opérateur « , ». Exemple :

*Syntaxe de déclarations multiples*

```
type var1 = expr1, var2, var3 = expr3; // déclarations
```

Tout ce vocabulaire doit être maîtrisé pour suivre le cours.

1. en fait il s'agit aussi de l'action d'un opérateur sur une zone mémoire
2. non utilisés dans ce cours
3. avec arguments possibles
4. mots définis dans le langage : **`if`**, **`else`**, **`do`**, **`while`**, **`for`**, **`switch`**, **`break`**, **`continue`**, **`return`**

## 2.2 Types

Le type d'une variable permet de lire corectement la valeur (l'ensemble des bits) de la variable. En particulier, c'est le type qui donne la taille de cette zone mémoire. Pour accéder à cette information on peut utiliser l'opérateur **`sizeof()`** qui agit directement sur le type (cf. l'exemple plus bas).

Il y a peu de types prédéfinis en `C` mais on a la possibilité d'en définir de nouveaux. On fait souvent la distinction entre types simples (entiers, réels, adresse) et types composés : tableaux, structures, etc.

### 2.2.1 Types simples

#### Types entiers

Plusieurs mots-clés sont définis pour représenter les entiers (relatifs et naturels). La seule différence entre ces types est la taille de la zone mémoire utilisée pour le stockage de la valeur de l'entier. Une remarque importante est que cette taille peut changer d'une architecture à une autre. Cependant sur des systèmes classiques (ordinateurs personnels) on peut donner les informations suivantes :

— **`short`** codé sur 2 octets
— **`int`** codé sur 4 octets
— **`long`** codé sur 4 ou 8 octets
— **`long long`**[5] codé sur 8 octets

Par défaut ces types représentent des entiers relatifs c'est à dire signés. Pour utiliser des entiers naturels on fait précéder le type du mot-clé **`unsigned`**.

Les constantes entières peuvent s'écrire en base 10 (par défaut), en base 8 (1 octet, nombre octal) ou en base 16 (2 octets, nombre hexadécimal). Dans ce cours on utilisera uniquement des nombres en base 10 mais il existe un piège à éviter :

— pour écrire un nombre en base 8 on fait précéder ce nombre du caractère `0`
— pour écrire un nombre en base 16 on fait précéder ce nombre des caractères `0x`

Par exemple le résultat de l'extrait de code suivant est l'affichage de 0 (car 27 en base 10 s'écrit 33 en base 8) :

*Attention, le `0` devant une constante entière a une signification !*

```
int m = 033, n = 27;  // déclaration et init. de n et m
printf("Resultat de m-n: %d\n", m-n);
```

Depuis le `C99` il existe des types de la forme `int_Nt` (pour les entiers signés) et `uint_Nt` (pour les entiers non signés) où `N` indique la taille en bits parmi `8`, `16`, `32`, `64`. Par exemple, on peut déclarer un entier `x` non signé sur 32 bits par l'instruction : `uint_32t x;`

Les opérateurs utilisables sur les variables entières sont les opérateurs arithmétiques (binaires et unaires), d'affectation, de comparaison et logiques. Notez que `/` désigne la division entière et `%` désigne l'opération *modulo*. On détaillera l'utilisation des opérateurs logiques plus loin dans ce cours.

A propos des opérateurs d'affectation il faut préciser la signification des opérateurs de la forme `o=` où `o` désigne l'un des caractères `+ - * / %` : l'expression *`expr1 o= expr2`* a le même effet que l'expression *`expr1 = expr1 o expr2`* (sauf qu'il n'y a qu'une évalutation de *`expr1`*). Exemple :

*Opérateurs d'affectation arithmétique*

```
unsigned long x = 45;
x /= 10 - 5;            // même effet que x = x / (10 - 5);
```

Reste enfin les opérateurs d'incrément `++` et de décrément `--`. Il existe 2 versions de ces opérateurs : une version préfixée et une version postfixée.

— version préfixée : l'opérateur est placé devant l'expression et incrémente ou décrémente cette expression **avant** un opérateur d'affecation éventuel

5. depuis le `C99`

— version postfixée : l'opérateur est placé derrière l'expression et incrémente ou décrémente cette expression **après** un opérateur d'affecation éventuel

Exemple :

*Opérateurs d'affectation arithmétique*

```
int x = 45;
int y = ++x;      //  préfixé, même effet que x=x+1; y=x;
int z = x++;      // postfixé, même effet que y=x;   x=x+1;
```

### Type caractère

Les caractères sont manipulés par le système comme des nombres entiers codés sur 1 octet. A chaque caractère correspond une valeur numérique entre 0 et 255. La table donnant la correspondance entre 1 caractère et sa valeur numérique s'appelle la table `ASCII`. Toutes les opérations effectuées sur les entiers sont valables sur les caractères. Le mot-clé utilisé pour le type caractère est **`char`**.

Une constante caractère est délimitée par des apostrophes simples ' et '. Voici un exemple complet à tester :

*Exemple d'utilisation du type **`char`***

```
#include <stdio.h>
int main(void) {
   char c1 = 'b';     // initialitation de c1 avec la valeur 'b'
   char c2 = c1 / 2;
   printf("Le caractere 'b' a pour code: %d\n", c1);
   printf("La nombre %d code le caractere: %c\n", c2, c2);
   return 0;
}
```

### Types réels

La représentation d'un nombre réel sur un nombre donné de bits est complexe. Quelque soit le système choisi, on utilise une approximation de ce nombre réel et l'erreur commise s'appelle erreur d'arrondie. Plusieurs mots-clés sont définis pour coder ces représentations de réels :

— **`float`** codage simple précision sur 4 octets (32 bits)
— **`double`** codage double précision sur 8 octets (64 bits)
— **`long double`** encore peu utilisé

Dans ce cours on privilégiera les variables de type **`double`** qui permettent une bonne précision et moins d'erreurs d'arrondies que le type **`float`** (cf. l'exemple suivant). Les constantes réelles peuvent s'écrire en notation scientifique en utilisant la lettre `e` (ou `E`).

*Manque de précision du type **`float`***

```
float x = 1e-4, s = 0;
int i;
for (i = 0; i < 1e6; ++i)   // i prend les valeurs de 0 à 1e6
   s += x;
```

La valeur de `s` à la fin de cette boucle est `99.3273` et non `100` comme attendu. Que se passe-t-il si on remplace l'instruction `s += x;` par l'instruction `s = (j+1)*x;` ? Ecrire le programme avec le type **`double`** et vérifier le résultat.

Les mêmes opérateurs que pour les entiers sont disponibles pour les réels (à l'exception du *modulo* **`%`**).

### Type adresse

Il existe un mécanisme très puissant que l'on détaillera dans la suite du cours qui permet de manipuler des zones mémoires (des variables ou des fonctions) par leur adresse. Il faut donc définir un type adresse qui dépend du type de la zone mémoire. Par exemple à l'adresse d'un **char** se trouve un **char** (1 octet) et à l'adresse d'un **double** se trouve un **double** (8 octets) : ces deux adresses ont des types différents.

La syntaxe pour déclarer une variable de type adresse sur un ***type*** est

*Déclaration d'une variable de type adresse*

```
type * variable;
```

Le ***type*** est suivi du caractère `*`. L'espace entre ***type*** et `*`, ou celui entre `*` et `variable`, est facultatif. La notation dépend du programmeur.

**Définition 4** *Une variable de type adresse sur un type est appelée* ***pointeur****. Par extension, le type adresse sera appelé type pointeur.*

Les opérateurs utilisables sont : `*` `&` `->` `++` `--` `-` (binaire) `+` (avec un entier). Les pointeurs seront détaillés dans le chapitre suivant.

### Conversions

Tous les types simples peuvent être convertis à l'aide de l'opérateur **(*type simple*)**. Ce mécanisme est à utilisé avec précaution mais permet par exemple la convertion d'un entier en un réel. Par la convertion d'un réel en un entier on utilisera plutôt les fonctions arrondi ou partie entière de la libairie mathématique : `round`, `floor`.

Dans l'exemple qui suit `n` est converti en **double** avant la division. Le compilateur utilise donc la division entre 2 réels et non la division entière entre 2 entiers.

*Exemple d'utilisation d'un opérateur de conversion*

```
int n = 4;
double x = 1 / (double) n;   // different de double x = 1 / 4;
```

## 2.2.2 Tableaux statiques

On parle de tableau statique car c'est un tableau dont la taille est connue à la compilation. Tous les éléments d'un tableau doivent être de même type. Pour déclarer un tableau de `N` éléments de même type ***type*** on utilise la syntaxe suivante :

*Déclaration d'un tableau de taille `N`*

```
type variable[N];
```

Un tableau de taille `N` est indexé de `0` à `N-1`. L'opérateur pour accéder à un élément du tableau est l'opérateur d'accès `[]` et prend pour unique argument un entier. Dans l'exemple suivant on déclare un tableau de `10` **double** que l'on initialise à `0` :

*Initialisation d'un tableau*

```
double tab[10];
int i;
for (i = 0; i < 10; ++i) {
   tab[i] = 0;   // i prend les valeurs successives de 0 à 9
};
```

Attention l'expression `tab[k]` avec `k` non dans `{0,...,9}` est autorisée par le compilateur (il n'y a pas d'erreur de syntaxe) mais peut être dangereux à l'execution et créer une erreur système.

Une syntaxe particulière est autorisée à l'initialisation : indiquer toutes les valeurs prises par les éléments du tableau dans une liste entre accolades :

*Syntaxe réservée à l'initialisation*

```
int tab[] = { 2, -3, 4e2, 11 };
```

### Tableaux de tableaux

Il est tout à fait possible de créer des tableaux d'éléments de types quelconques, en particulier de type tableau. Pour déclarer une variable `matrice` qui serait un tableau de `N` tableaux de `M` éléments de type **`double`** on écrit ainsi

*Tableau de `N` tableaux de `M` réels*

```
double matrice[N][M];
```

Il n'y a pas d'opérateur spécifique pour l'utilisation d'une telle structure de données. Il faut donc utiliser plusieurs fois l'opérateurs d'accès `[]` pour accéder à un élément :

*Accès à un élément*

```
matrice[i];         // accès au (i+1)-ème tableau (de taille M)
matrice[i-1][j-1]; // accès au j-ème double du i-ème tableau
```

### Chaînes de caractères

Une chaîne de caractères est représentée en interne comme un tableau de **`char`**. Plus précisément, une chaîne de `N` caractères est codée par un tableau de `N+1` **`char`** dont le dernier est le caractère nul `'\0'` (de code `ASCII 0`). Il existe une facilité d'écriture des constantes chaînes de caractères à l'aide des guillements " ".

*Initialisation d'une chaîne de caractères*

```
char st1[] = "Bonjour";
char st2[] = { 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };
```

Les deux variables `st1` et `st2` représentent les mêmes chaînes de caractères.

## 2.2.3 Structures

Une structure est un type composé qui permet de regrouper plusieurs variables de types différents. Les variables regroupées au sein d'une structure s'appellent des *champs*. On associe souvent un nom à la structure (donné après le mot-clé **`struct`**) ce qui permet de la réutiliser dans la suite du programme. La déclaration d'une structure se fait dans un fichier d'en-tête `.h` si elle doit être réutilisée ou bien dans un fichier source : dans une fonction (déclaration locale) ou en-dehors de toute fonction (déclaration globale). La syntaxe pour définir une structure qui porte le nom `nomStruc` et qui contient `N` champs est

*Syntaxe pour la définition d'une structure de N champs*

```
struct nomStruct {
   type1 var1;
   type2 var2;
   ...
   typeN varN;
};
```

Pour utiliser cette structure, il faut déclarer des variables de type **struct** nomStruct. L'accès aux champs se fait ensuite en utilisant l'opérateur « . ».

Voici un exemple où l'on utilise la structure fiche qui contient les 2 champs : numero de type **int** et nom de type **char[20]** (une chaîne d'au plus 20 caractères).

*Exemple d'utilisation d'une structure*

```
struct fiche {       // définition de la structure fiche
   int      numero;
   char[20] nom;
};
struct fiche f;      // décl. de la variable f de type struct fiche
f.numero = 1;        // accès au champ numero
f.nom = "toto";      // accès au champ nom
```

Comme pour l'initialisation des tableaux, il est possible d'initialiser les champs au moment de l'initialisation d'une variable de type structure. L'ordre des éléments de la liste d'initialisation doit respecter l'ordre de déclaration des champs. Par exemple

*Syntaxe réservée à l'initialisation*

```
struct fiche f = { 1, "toto" };
```

Depuis le C99, il est possible d'initialiser les champs dans un ordre différent en indiquant le nom du champ.

*Syntaxe possible en C99*

```
struct fiche f = { .nom = "toto", .numero = 1 };
```

### 2.2.4 Unions

Une union se déclare exactemet comme une structure, en remplaçant le mot-clé **struct** par le mot-clé **union**. La différence réside uniquement dans le fait qu'une seule zone mémoire est utilisée pour tous les champs de l'union. Donc modifier un champ revient à modifier tous les autres champs.

### 2.2.5 Enumérations

Une énumération regroupe un ensemble de membres appelés éléments. Par exemple, on peut considérer une énumération appelée *couleurs* composée de 4 éléments : *pique, cœur, trèfle, carreau*. A la compilation il y a une correspondance entre un élément et une valeur numérique entière, et tout opérateur s'appliquant à un entier pourra s'appliquer à un élément. Une énumération permet essentiellement de rendre plus lisible un code en spécifiant des groupes d'éléments.

La syntaxe est la suivante : mot-clé **enum** suivi du nom de l'énumération puis d'une liste contenant les éléments.

*Exemple d'utilisation d'une énumération*

```
enum couleur { pique, coeur, trefle, carreau };
struct carte {
   short valeur;
   enum couleur col;            // champ col de type enum couleur
};
struct carte X = { 4, pique };   // init. de la variable X
```

En interne, `pique` vaut 0, `coeur` vaut 1, etc.

On peut aussi préciser des valeurs numériques spécifiques pour les éléments. Voici un exemple

*Définition d'une énumération avec valeurs spécifiées*

```
enum ville { paris = 75, marseille = 13, lille = 59 };
```

### 2.2.6 Autres considérations sur les types

#### Taille réservée en mémoire

Il existe un opérateur permettant de connaître la taille réservée par un type. Il s'agit de l'opérateur **`sizeof`**, le seul opérateur en `C` qui peut s'appliquer directement sur un type. Par extension, on peut aussi l'appliquer à une variable. La syntaxe est simplement **`sizeof(`*`type`*`)`** ou **`sizeof(variable)`**. L'opérateur renvoie une taille entière[6] qui correspond au nombre d'octets réservés.

Pour connaître la taille du type **`int`** ou **`long`** ainsi que la taille d'un tableau de double on peut écrire les instructions suivantes :

*Définition d'une énumération avec valeurs spécifiées*

```
printf("taille d'un int:%u\n", sizeof(int));
printf("taille d'un long:%u\n", sizeof(long));
printf("taille d'un tableau de 4 double:%u\n", sizeof(double[4]));
double tab[4];
printf("taille d'un tableau de 4 double:%u\n", sizeof(tab));
```

#### Redéfinition de type

On peut redéfinir un type à l'aide du mot-clé **`typedef`** (ce n'est ni un opérateur, ni une instruction, juste un mot-clé comme **`break`**, **`return`**, etc.). La syntaxe est la suivante

*Syntaxe pour la redéfinition de type*

```
typedef ancientypelongetcompliqueetillisible type_simple;
```

Il est très facile de redéfinir des types et c'est une bonne habitude de programmation à prendre. Si l'***`ancientypelongetcompliqueetillisible`*** est un tableau il faut donner les dimensions du tableau derrière le ***`type_simple`***. Par exemple on doit écrire

*Exemple de redéfinitions de type*

```
typedef struct carte jeu[52];
jeu J;          // J est un tableau de 52 cartes
```

6. il s'agit d'une valeur de type `size_t` équivalent à un entier non signé

**Nouveaux types en C99**

A faire !

## 2.3 Tests

Les instructions de tests permettent d'executer des instructions en fonction du résultat d'une expression. Moralement, si l'expression est vraie on execute un bloc d'instructions et si l'expression est fausse on passe ce bloc sans l'évaluer et/ou on execute un autre bloc. Une expression qui peut être évaluée comme vrai ou fausse est dite expression booléenne.

### 2.3.1 Expression booléenne

En C une expression booléenne est une expression prenant des valeurs entières. A la valeur 0, l'expression est dite fausse et pour toute autre valeur l'expression est dite vraie. Pour former ces expressions on utilise principalement les opérateurs de comparaisons et les connecteurs logiques.

Il existe 6 opérateurs de comparaisons : == (égalité), != (différent), < , <= (inférieur ou égal), > , >=. Ces opérateurs s'utilisent uniquement entre des expressions de type simple (entier, réel, adresse) et de même type (sauf si on sait ce que l'on fait...). Le résultat d'un opérateur de comparaison est un entier qui vaut 0 ou 1, c'est donc une expression booléenne.

Les opérateurs de comparaison ont la priorité sur les opérateurs d'affectation, mais il est recommandé d'utiliser les parenthèses pour plus de lisibilité :

*Utilisation d'un opérateur de comparaison*

```
int test1 =  3.0 >= sqrt(8.0);     // OK priorité de >= sur =
int test2 = (3.0 >= sqrt(8.0));    // écriture recommandée
```

En plus de ces opérateurs il existe 2 opérateurs logiques && (et), || (ou), et l'opérateur de négation !. L'opérateur de négation a la priorité sur les opérateurs de comparaisons et les opérateurs de comparaison ont la priorité sur les 2 opérateurs logiques. Ainsi on peut écrire le code suivant

*Utilisation d'un opérateur logique*

```
int test1 = !(10 > x);                 // parenthèses obligatoires
int test2 = ((x >= 0) && (x < 10));    // écriture recommandée
int test3 = x >= 0 && x < 10;          // écriture possible
```

Ordre d'évaluation des connecteurs logiques :

— *expr1* && *expr2* : évaluation de *expr1* :
- si faux (0) : *expr2* n'est pas évaluée et résultat faux
- sinon : *expr2* est évaluée et donne le résultat

— *expr1* || *expr2* : évaluation de *expr1* :
- si faux (0) : *expr2* est évaluée et donne le résultat
- sinon : *expr2* n'est pas évaluée et résultat vrai

Dans la suite on notera *expr_bool* une expression booléenne.

### 2.3.2 Test if else

Il existe deux syntaxes pour cette instruction de branchement bien connue. La première consiste en l'évaluation de *instr* uniquement si *expr_bool* est vraie :

*Syntaxe du if*

```
if (expr_bool)
   instr
```

La seconde syntaxe est utilisée pour l'évalution de *instr1* si *expr_bool* est vraie et de *instr2* si *expr_bool* est fausse.

*Syntaxe du if else*

```
if (expr_bool)
   instr1
else
   instr2
```

Attention les parenthèses autour de l'expression conditionnelle *expr_bool* sont obligatoires et font partie de la syntaxe.

L'instruction **if else** est une instruction. On peut donc imbriquer des **if else** comme dans l'exemple suivant

*Exemple de if else imbriquées*

```
if (n % 2 == 0)
   if (n % 4 == 0)
      printf("multiple de 4\n");
   else
      printf("multiple de 2\n");
else
   if (n % 3 == 0)
      printf("multiple de 3\n");
printf("pas multiple de 2, 3 et 4");
```

### 2.3.3 Opérateur conditionnel

Il existe auss l'opérateur conditionnel ? : qui permet une écriture concise et l'utilisation conjointe d'un opérateur d'affectation. La syntaxe est

*Syntaxe de l'opérateur conditionnel*

```
expr_bool ? instr1 : instr2
```

L'expression booléenne *expr_bool* est évaluée : si vraie on execute *instr1* sinon *instr2*. L'effet est donc le même qu'un **if else** mais il s'agit d'un opérateur et non d'une instruction. Cet opérateur est donc très utile pour former des expressions dont le comportement change en fonction d'une expression booléenne.

Par exemple

*Exemple*

```
double x_plus = x > 0 ? x : 0;
double abs_x = x > 0 ? x : (-x);
```

### 2.3.4 Test switch

L'instruction **switch** permet d'executer des instructions en fonction de la comparaison d'une expression donnée avec des valeurs constantes de même type. Dans la syntaxe suivante on note *expr_cst1*, ..., *expr_cstN* des constantes que l'on peut comparer avec l'expression *expr* :

*Syntaxe du* `switch`

```
switch (expr) {
   case expr_cst1:
      instr1
   case expr_cst2:
      instr2
   //  ...  (raccourci ne fait pas partie de la syntaxe)
   case expr_cstN:
      instrN
   default:    // facultatif
      instr_def
}
```

Le cas **`default`** est facultatif.

Attention, le fonctionnement (non intuitif au départ) est le suivant :

— s'il existe une constante *`expr_cstK`* telle que *`expr == expr_cstK`*, toutes les instructions suivantes sont executées : `instrK`, ..., *`instrN`*, et *`instr_def`*.

— si aucune constante ne correspond, l'instruction *`instr_def`* est executée si elle existe.

On utilise souvent l'instruction **`break`** à la fin des instructions `instrK` pour sortir du immédiatement du **`switch`** et éviter l'execution des instructions suivantes. Voici un exemple

*Exemple d'un* `switch` *avec* `break`

```
enum couleur c = coeur;  // voir la définition de enum couleur
switch (c) {
   case pique:
      printf("c est du Pique\n");
      break;
   case coeur:
      printf("c est du Coeur\n");
      break;
   case trefle:
      printf("c est du Trefle\n");
      break;
   case carreau:
      printf("c est du Carreau\n");
      break;
   default:
      printf("c n'est pas une couleur\n");
}
```

## 2.4 Boucles

On vient de voir que le langage `C` était composé d'expressions (mélangeant variables, constantes, opérateurs et résultats d'appels de fonction), d'instructions (séquentielles) et de blocs, et d'instructions de branchements (**`if`**, **`switch`**). Pour en faire un langage impératif complet il faut lui rajouter des instructions de boucles qui permettent de répéter des instructions.

### 2.4.1 Boucle `for`

L'instruction **`for`** est extrêmement souple en `C`. La syntaxe est la suivante

*Syntaxe de l'instruction for*

```
for (expr1 ; expr2 ; expr3)
   instr
```

Les expressions *expr1*, *expr2* (booléenne), et *expr3* sont facultatives (du point de vue syntaxique) mais les points virgules sont obligatoires ! Le comportement est le suivant : l'expression *expr1* est évaluée (initialisation), puis tant que *expr2* est vraie (continuation) on execute *instr* suivie de *expr3* (incrémentation).

### 2.4.2 Boucle `while`

L'instruction **while** permet l'effet suivant : tant que l'expression booléenne *expr_bool* est évaluée,l'instruction *instr* est évaluée.

*Syntaxe de l'instruction while*

```
while (instr_bool)
   intr
```

### 2.4.3 Boucle `do while`

L'instruction **do while** est similaire à l'instruction précédente saut qu'on s'assure qu'au moins une instruction *instr* est évaluée. Le test de sortie (évaluation de *expr_bool*) a lieu après *instr*.

*Syntaxe de l'instruction do while*

```
do {
   instr
} while (instr_bool);
```

Voici un exemple d'utilisation. Que fait ce programme ?

*Exemple d'un do while*

```
#include <stdio.h>
int main(void) {
   char c;
   do {
      printf("Veuillez tapez la lettre 'x'\n");
      scanf("%c", &c);
   } while (c != 'x');
   return 0;
```

## 2.5 Fonctions

En C, les fonctions sont identifiées par leur nom. Ainsi deux fonctions différentes ne peuvent pas porter le même nom. La définition d'une fonction est un bloc de déclarations et d'instructions qui s'appliquent à des arguments et qui produisent un résultat : renvoie d'une valeur (d'un type donné) ou de rien. Ce bloc se situe comme les données (variables, constantes) en mémoire, et après compilation se compose d'une suite de 0 et 1.

**Définition 5** *Une **fonction** est un espace réservé en mémoire qui possède les caractéristiques suivantes :*

— son nom *(unique) qui permet de l'identifier,*
— son prototype*, qui indique le type des arguments et le type de retour,*
— son contenu*, c'est la séquence de bits qui code la fonction,*
— son adresse*, c'est l'endroit dans la mémoire où elle est stockée.*

Dans sa syntaxe de base, une fonction prend un nombre fixe d'arguments. Ce nombre peut être quelconque mais doit être déduit du prototype de la fonction. Le nom des arguments est optionnel dans le prototype d'une fonction.

*Prototypes d'une fonction*

```
type_retour nom0(void)                  // fonction sans argument
type_retour nom1(type)                  // fonction à 1 argument
type_retour nom2(type1, type2)          // fonction à 2 arguments
type_retour nom3(type1, type2, type3)   // fonction à 3 arguments,...
```

Il est important de distinguer
— la *déclaration* d'une fonction qui se fait en déclarant uniquement le prototype, souvent dans un fichier d'en-tête pour une réutilisation du code,
— la *définition* d'une fonction qui fait suivre le prototype (avec des noms aux arguments) d'un bloc contenant les déclarations et instructions de la fonction.

De plus, il existe 2 comportements possibles :
— *la fonction ne renvoie rien* : dans ce cas le **type_retour** doit être le type **void**, le bloc d'instructions est executé jusqu'à l'accolade fermante ou jusqu'au mot-clé **return** s'il est présent (et il n'est suivi d'aucune expression : **return** ;).
— *la fonction renvoie une valeur* : alors le **type_retour** doit correspondre à la valeur renvoyée et le mot-clé **return** doit être présent dans le bloc d'instructions suivi d'une expression de type **type_retour**.

Par exemple, si on veut une fonction `puissance` qui effectue l'opération $x^n$ pour $x \in \mathbb{R}$ et $n \in \mathbb{N}$ on peut écrire

*Exemple de déclaration / définition d'une fonction*

```
double puissance(double x, unsigned int n); // déclaration uniquement
double puissance(double x, unsigned int n) {// bloc de définition
   double y = 1;
   while (n-- > 0)      // d'abord le test n > 0 puis décrément
      y *= x;
   return y;            // y est bien de type double
};
```

Attention il ne peut pas y avoir de fonctions imbriquées, c'est à dire qu'il est impossible de déclarer ou définir une fonction à l'intérieur (dans le bloc) d'une autre fonction.

Dès que le prototype est déclaré, il est possible d'utiliser la fonction. C'est à dire d'effectuer un appel de la fonction avec des arguments donnés. Lors de l'appel, les arguments doivent être des expressions qui ont le même type que ceux déclarés dans le prototype (sinon il y a une erreur de compilation facile à corriger). L'appel d'une fonction se fait en donnant le nom de la fonction suivi de l'opérateur `()`, exemple :

*Exemples d'appel d'une fonction*

```
double pi = 3.14159;
int n = 3;
double z = puissance(sqrt(2*pi), n);
z /= puissance(1+puissance(2*pi, n), 2); //appels imbriqués autorisés
```

### 2.5.1 Arguments

Les arguments d'une fonction doivent être vus comme des variables locales du bloc de définition de la fonction. Ainsi les arguments ne vivent (sont en mémoire, visibles et accessibles) que dans le bloc de définition de la fonction. *La valeur de chaque argument est donné lors de l'appel de la fonction.* Après l'appel de la fonction, il est impossible de récupérer la valeur des variables locales de la fonction.

#### Passage par valeur

Pour toute fonction, le passage des arguments se fait par valeur. Pour bien comprendre, supposons qu'on souhaite écrire une fonction qui échange le contenu de 2 variables de type **int**. On souhaite donc avoir le comportement suivant :

*Appel d'une fonction* `echange`

```
int n = 2, m = -1e5;
echange(n, m);
if (n == -1e5 && m == 2)
   printf("echange reussi !");
else
   printf("echec total !");
```

Quelque soit le code de la fonction `echange`, le résultat de ce programme sera l'affichage `"echec total !"`. Il est impossible de coder une fonction de prototype `void echange(int a, int b)` qui échange le contenu de 2 variables entières. En effet, lors de l'appel `echange(n, m)` le contenu de `n` (*i.e.* `2`) est copié dans l'argument `a` et `-1e5` est copié dans l'argument `b`. Puis la fonction agit uniquement sur `a` et `b`, sans avoir accès à `n` et `m`.

Pour information, voici un code naïf possible pour la fonction `echange`

*Code possible pour la fonction* `echange`

```
void echange(int a, int b) {
   int tmp = a;      // on utilise une variable locale
   a = b;
   b = tmp;
   // pas de return car le type de retour est void
};
```

#### Ordre d'évaluation des arguments

On dit qu'une fonction (ou un opérateur) est à *effet de bord* ou à *effet secondaire* si elle modifie des variables autre que ses variables locales et sa valeur de retour ou bien si son comportement diffère en fonction du contexte. Par exemple, une fonction qui modifie ou qui dépend d'une variable globale est à effet de bord. Ce sont des fonctions qui peuvent rendre un programme complexe à comprendre et à valider. Dans ce cours, nous évitons l'utilisation de variables globales et nous écrirons des *fonctions pures* qui donnent toujours le même résultat pour les mêmes arguments.

Un autre point important est que l'ordre d'évaluation des arguments n'est pas défini dans la norme du langage `C` et dépend donc du compilateur utilisé. Cela peut avoir des conséquences si les arguments sont initialisés par des expressions ayant des effets de bord. Par exemple :

*Attention à l'ordre d'évaluation des arguments*

```
int f(int a, int b, int c) { return a*b*c; }
int main(void) {
   int n = 2;
   int a = f(5, ++n, n % 2);
}
```

Si `n % 2` est évalué avant `++n` le résultat est `a = 0` puis `n` sera à 3, et si `++n` est évalué avant `n % 2` le résultat est `a = 15` (et `n` sera à 3). La fonction `f` est une fonction pure mais elle est appelée avec une expression ayant un effet de bord : l'opérateur `++`. La bonne façon d'appeler la fonction `f` est la suivante :

*Attention à l'ordre d'évaluation des arguments : version correcte*

```
int f(int a, int b, int c) { return a*b*c; }
int main(void) {
   int n = 2;
   int m = ++n;
   int a = f(5, m, m % 2);
}
```

### 2.5.2 Fonction `main`

On a déjà vu la fonction `main` qui est nécessaire pour créer un code executable (mais pas nécessaire pour un fichier objet). Il existe (exceptionnelement) 2 prototypes possibles pour la fonction `main`

*Prototypes de la fonction `main`*

```
int main(void)
int main(int argc, char *argv[])
```

Le type de retour est toujours un entier : s'il vaut `0` on indique au système d'exploitation que le programme s'est bien déroulé, sinon on renvoit un entier qui indique un code d'erreur qui doit être documenté.

Le deuxième prototype prend 2 arguments qui permettent de récupérer des informations sur la ligne de commande qui a lancé le programme. Par exemple, quand on execute le programme `gcc` pour compiler le fichier `test.c` on écrit `gcc -c test.c`. Il y a donc 3 mots sur la ligne de commande : `gcc` puis `-c` puis `test.c`. Dans ce cas, l'argument `argc` qui compte le nombre de mots sur la ligne de commande sera à `3`, et l'argument `argv` qui peut être vu comme un tableau de chaînes de caractères sera défini comme ceci : premier mot dans `argv[0]`, deuxième mot dans `argv[1]`, etc.

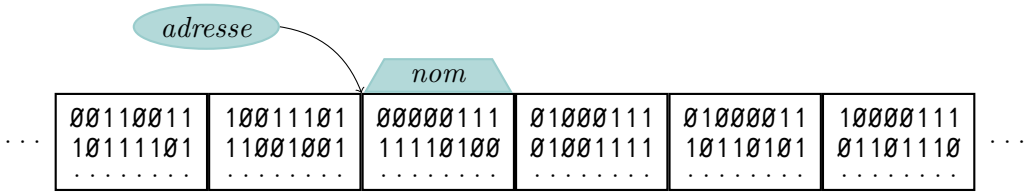
### 2.5.3 Fonctions à nombre variable d'arguments

A faire.

# Chapitre 3

# Pointeurs

Reprenons la représentation de la mémoire :




On a vu que des parties du code (variables et fonctions) sont stockées en mémoire et possèdent un `nom` et une `adresse`. Cette adresse liée au type de ce qui s'y trouve peut être directement utilisée par le programmeur. Pour cela, il utilise une variable de type adresse ou encore pointeur.

## 3.1 Déclaration

La déclaration d'un pointeur sur un ***type*** utilise le symbole `*` entre le ***type*** et le nom du pointeur :

*Syntaxe de déclaration d'un pointeur*

```
type * pointeur1;
type ** pointeur2;
type * pointeur3[5];
```

On dit alors que la variable `pointeur1` est de type : pointeur sur ***type***. De même, il est possible de déclarer une variable qui permet d'agir sur l'adresse de `pointeur1`. Il s'agit ici de `pointeur2` qui est de type : pointeur sur pointeur sur ***type***. Enfin, la variable `pointeur3` ci-dessus est un un pointeur sur un tableau de 5 ***type***.

Il existe 2 pointeurs particuliers qui permettent notamment la programmation générique et les structures récursives.

### Pointeur générique

Le pointeur générique est le pointeur de type **`void *`**. C'est un pointeur compatible avec tout autre type de pointeur donc n'importe quel pointeur peut être affecté à une variable de type **`void *`**. Réciproquement, en utilisant l'opérateur de conversion **`(type *)`** ou une conversion implicite on peut récupérer un pointeur typé d'un pointeur générique.

### Pointeur `NULL`

Il s'agit d'une constante préprocesseur définie dans `stddef.h` inclus par `stdlib.h`. Cette constante est le pointeur générique d'adresse `0` :

*Définition de `NULL` dans `stddef.h`*

```
#define NULL ((void *) 0)
```

C'est le pointeur qui ne pointe vers rien ! Il sera utilisé comme valeur particulière d'adresse pour représenter une erreur ou une valeur de fin dans une structure récursive.

Nous verrons la syntaxe spécifique aux pointeurs sur fonctions plus loin. Dans toute la suite, on considère des pointeurs sur variables.

## 3.2 Initialisation

**Définition 6** *L'opérateur `&` appliqué à une variable permet de récupérer son adresse, c'est l'*opérateur de référencement.

Il n'existe que 2 façons d'initialiser un pointeur :

— en récupérant l'adresse d'une variable existente en mémoire : pour cela on utilise l'opérateur de référencement,

— en démandant la création d'une zone mémoire de taille et de type donnés (on peut parler de variable anonyme car seul le nom manque) ; pour cela on utilise une fonction d'allocation comme `malloc` *(memory allocation)* déclarée dans `stdlib.h`.

La fonction `malloc` a pour prototype : `void * malloc(size_t size)`. C'est donc une fonction à un argument : `size` de type `size_t` (que nous considérons ici comme un `unsigned`) et qui renvoie le pointeur générique. Si une erreur s'est produite lors de l'allocation, c'est le pointeur `NULL` qui est renvoyé.

*Exemple des deux initialisations possibles*

```
// initialisation en récupérant l'adresse de x
double x = 10;
double * pt1 = &x;

// initialisation en demandant la création d'une zone mémoire
double * pt2 = malloc(sizeof(double));
if (pt2 == NULL) printf("Erreur d'allocation !");
```

L'allocation de mémoire permet une grande liberté au programmeur mais demande une grande rigueur dans l'écriture du programme. Notamment, une allocation doit toujours être suivie d'une libération de mémoire dans le même programme. Pour cela, on utilise la fonction `void free(void *)` de la `stdlib.h`.

*Exemple complet : allocation / libération*

```
#include <stdlib.h>

int main(void) {
   double * pt = malloc(sizeof(double));  // conversion implicite
   if (pt == NULL) return 1;
   // on travaille avec la variable pt
   free(pt);
   return 0;
}
```

## 3.3 Utilisation

**Définition 7** *L'opérateur* `*` *appliqué à une adresse permet de récupérer la valeur de ce qui s'y trouve, c'est l'*opérateur de déréférencement.

On utilise donc l'opérateur `*` sur un pointeur pour accéder à la valeur pointée. Attention, si le pointeur est non intialisé, ou initialisé à `NULL`, on obtient une erreur d'execution : un arrêt du programme.

*Utilisation de l'opérateur de déréférencement*

```
double x = 10;
double * pt1 = &x;  // symbole * indiquant pt1 de type double *
*pt1 = 15;          // operateur * appliqué à pt1
```

Que vaut `x` à la fin du programme ?
Cet exemple très important doit être bien compris : le contenu pointé par `pt1` est le contenu de la variable `x`. Tout accès `*pt1` correspond à un accès sur la valeur de la variable `x`.

### Passage par adresse

Une première utilisation très utile des pointeurs est de forcer le passage par adresse des arguments d'une fonction. On a déjà vu qu'il est impossible d'écrire une fonction `void echange(int a, int b)` qui échange les valeurs de 2 variables entières.

Si on remplace les entiers `a` et `b` par leurs adresses, alors les adresses seront recopiées en des variables locales (passage par valeur) mais le contenu de ces pointeurs locaux sera les adresses de `a` et `b`. Cette fonction pourra donc bien modifier le contenu des entiers dont les adresses sont données en argument. Voici le code d'une telle fonction :

*Pointeurs pour forcer le passage par adresse*

```
void echange(int * a, int * b) {
   int tmp = *a;
   *a = *b;
   *b = tmp;
}

int main(void) {
   int x = 10, y = 15;
   echange(&x, &y);
   ...
}
```

## 3.4 Arithmétiques des pointeurs

On peut effectuer des opérations arithmétiques simples sur les adresses. Plus précisément

— addition/soustraction d'un pointeur avec un entier `n` : permet d'avancer ou de reculer de `n` cases mémoires ; la taille d'une case mémoire est donnée par le *type du pointeur*. Le programmeur n'a pas à se soucier de la taille d'une case.
Les opérateurs utilisés sont : `+ - ++ --` (et les opérateurs d'affectations associés).

— différence entre 2 pointeurs du même type : permet de donner la distance (en nombre de cases mémoires) entre 2 adresses. L'opérateur est `-`.

### Tableaux dynamiques

Ces opérations simples sur les pointeurs permettent des tableaux dont la taille est connue à l'exécution : on parle de tableau dynamique. Il suffit de demander l'allocation de `n` cases mémoires et de se déplacer avec un pointeur. Voici un exemple avec un tableau d'**`int`** que l'on initialise à 1 :

*Exemple d'un tableau dynamique*

```
unsigned n = 10;
int * tab = malloc(n * sizeof(int));  // tab: adresse du 1er int
int * pt;                             // pt: se deplace d'int en int
for (pt = tab; pt != tab+10; ++pt)
   *pt = 1;
// autres instructions
free(tab);
```

L'opérateur d'indexation `[]` est possible avec un pointeur et à la signification suivante :

`tab[i]` est équivalent à `*(tab+i)`

Par exemple, c'est contre-intuitif et donc déconseillé, vous pouvez écrire `i[tab]` qui est équivalent à `*(i+tab)` ou `tab[i]`. Ainsi l'exemple précédent peut s'écrire :

*Exemple d'un tableau dynamique*

```
unsigned n = 10;
int * tab = malloc(n * sizeof(int));  // tab: adresse du 1er int
int i;
for (i = 0; i < 10; ++i)
   tab[i] = 1;
// autres instructions
free(tab);
```

Attention, le code n'est pas équivalent car dans la première version vous incrémentez le pointeur `pt` alors que dans la seconde version vous incrémentez l'indice `i` puis vous calculez la nouvelle adresse `tab+i`.

De même pour créer des tableaux multidimensionnels, on peut utiliser un pointeur sur pointeur. Par exemple, pour allouer un tableau de 4 lignes et 10 colonnes et initialiser tous ces éléments à 1 on peut écrire le code suivant :

*Tableau bidimensionnel dynamique*

```
double ** matrice;
unsigned i, j, nl = 4, nc = 10;
matrice = malloc(nl * sizeof(double *));
for (i = 0; i < nl; ++i) {
   matrice[i] = malloc(nc * sizeof(double));
   for (j = 0; j < nc; ++j)
      matrice[i][j] = 1;
}
// autres instructions
for (i = 0; i < nl; ++i)
   free(matrice[i]);
free(matrice);
```

Dans ces exemples, on ne vérifie pas que l'allocation de mémoire s'est bien passée, c'est à dire que `malloc` renvoie une adresse valide et non l'adresse `NULL`.

Comme exercice, vous pouvez écrire ce programme en ajoutant les instructions de tests vérifiants l'allocation de mémoire : si une erreur se produit lors de l'allocation de `matrice[i]` on doit libérer (avec `free`) tout ce qui a déjà été alloué puis quitter avec un code d'erreur.

### Priorité ++/− sur *

Les opérateurs arithmétiques unaires `++` et `--` ont priorité sur l'opérateur de déréférencement `*`. Si il y a une affectation dans l'expression à évaluée, elle a lieu avant ou après en fonction de la place de l'opérateur unaire (préfixé ou postfixé).

*Explication des différents cas*

```
int i, * p; // i entier et p pointeur sur entier
i = *++p;   // incrément de p puis affectation de la valeur pointée
i = *p++;   // affectation de la valeur pointée puis incrément de p
i = ++*p;   // incrément de la valeur pointée puis affectation
i = (*p)++; // affecte la valeur pointée puis incrémentation
```

Par exemple, la fonction `strcpy` (dont le prototype est déclarée dans `string.h`) qui effectue la copie entre 2 chaînes de caractères (des tableaux de `char`) peut s'écrire de la façon suivante :

*Exemple de la fonction `strcpy`*

```
char * strcpy(char * dest, char * src) {
   char * d = dest, * s = src;
   while ((*d++ = *s++) != '\0')
      ;        // instruction vide, tout est déjà fait !
   return dest;
}
```

## 3.5 Pointeurs et `const`

Le qualifieur **`const`** peut s'utiliser lors de la déclaration d'une variable pour changer son comportement. Une variable déclarée **`const`** ne pourra pas changer de valeur : elle gardera donc toujours sa valeur initiale (la valeur donnée explicitement lors de la déclaration). La place du qualifieur **`const`** peut prêtée à confusion lors de son utilisation avec les pointeurs. Le qualifieur fait partie du ***`type`***.

Si ***`type`*** n'est pas une adresse vous pouvez de façon équivalent écrire

*Ecritures équivalentes si **`type`** n'est pas une adresse*

```
const type x1 = val;     // écriture historique très utilisée
type const x2 = val;     // écriture équivalente
```

Dans le cas où ***`type`*** est une adresse, par exemple **`double *`** il faut faire attention car les 2 écritures précédentes ne sont pas équivalentes. La première correspond à un *pointeur sur valeur constante* et la seconde à un *pointeur constant*.

### Pointeur constant

Un pointeur constant doit être obligatoirement initialisé.

*Syntaxe d'un pointeur constant*

```
type * const p = val;
```

Un pointeur constant correspond exactement à un tableau statique (ou l'inverse !). Le nom du pointeur représente le nom du tableau et vous ne pouvez pas changer sa valeur. Par contre il est possible de changer la valeur sur laquelle vous pointez :

*Exemple d'un pointeur constant*

```
double * const p = malloc(10 * sizeof(double));
p[0] = 1;     // autorisé
p = p+1;      // interdit
```

### Pointeur sur valeur constante

Un pointeur sur valeur constante est utilisé principalement pour forcer le passage par adresse des arguments d'une fonction, tout en préservant le contenu de la variable pointée. Il y a deux syntaxes équivalentes qu'il faut connaître, même si dans ce cours on utilisera plus souvent la seconde.

*Syntaxe d'un pointeur sur valeur constante*

```
const type * p;       // écriture historique très utilisée
type const * p;       // écriture conseillée
```

Exemple d'utilisation :

*Exemple d'un pointeur sur valeur constante*

```
void affiche_tableau(double const * tab, unsigned size) {
   *tab = 2;      // interdit
   tab[0] = 2;    // interdit, c'est équivalent à *tab
   double * pt = tab;              // interdit
   int i;
   for (i = 0; i < size; ++i)
      printf("%g\n", tab[i]);      // autorisé
}
```

Les accès en écriture sont donc interdits et les accès en lecture sont autorisés. Noter l'importance de la ligne 4 : il n'y a pas de conversion implicite entre un `double const *` et un `double *`. Ce sont deux types différents.

### Pointeur constant sur valeur constante

On peut aussi déclarer un pointeur constant sur valeur constante en combinant les 2 syntaxes précédentes.

*Syntaxe d'un pointeur constant sur valeur constante*

```
const type * const p;       // écriture historique
type const * const p;       // écriture conseillée
```

## 3.6 Pointeurs de fonction

Quelques règles sont spécifiques aux pointeurs sur fonctions. Tout d'abord, il n'y a qu'une seule façon d'initialiser un pointeur sur fonction : récupérer l'adresse d'une fonction existante. On ne peut pas allouer un espace et mettre des instructions dedans. La syntaxe pour déclarer un pointeur est proche de celle de la déclaration d'un prototype avec les symboles `(*)` en plus :

*Prototypes d'une fonction*

```
type_retour (*pt0)(void); // pointeur sur: une fonction sans argument
                          // qui renvoie une valeur type_retour
type_retour (*pt1)(type);
type_retour (*pt2)(type1, type2);
type_retour (*pt3)(type1, type2, type3);
```

Noter que les parenthèses autour du nom précédé de `*` sont obligatoires pour ne pas confondre avec une fonction qui renvoie un pointeur sur ***type_retour***.

Supposons qu'il existe une fonction de prototype `double puissance(unsigned n, double x)`. Alors on peut effectuer les instructions suivantes :

*Exemple d'un pointeur sur fonction*

```
double (*fct)(unsigned, double);  // déclaration du pointeur
fct = &puissance; // on initialise avec l'adresse de puissance
(*fct)(2, 3.14);  // on déréférence, correspond à puissance(2, 3.14)
```

Mais il est équivalent, et cela uniquement pour les pointeurs sur fonction, d'écrire le code suivant

*Exemple d'un pointeur sur fonction*

```
double (*fct)(unsigned, double);  // déclaration du pointeur
fct = puissance;
fct(2, 3.14);
```

Les opérateurs `&` et `*` sont ici implicites.

Les pointeurs sur fonction permettent de passer une fonction en argument d'une autre fonction. Dans l'exemple qui suit, on écrit une fonction `integre` qui calcule une approximation de l'intégrale d'une fonction réelle sur un intervale $[a, b]$. L'approximation se fait par la méthode des rectangles en considérant $n$ points équidistants dans $[a, b]$.

*Exemple d'un pointeur sur fonction*

```
typedef double (*fctRR)(double);     // on définit le type fctRR

double integre(double a, double b, unsigned n, fctRR f) {
   double result = 0, h = (b-a) / (double) n;
   int k;
   for (k = 0; k < n; ++k)
      result += f(a + k*h);
   return result;
}
```

## 3.7 Pointeurs et structures

Les pointeurs sur structures sont très utilisés et permettent de créer des structures récursives comme les listes chaînées, les arbres, etc. L'utilisation est exactement la même que pour une variable d'un type simple. Reprenons l'exemple de la `struct carte` qui contient 2 champs `valeur` et `col`.

*Exemples d'utilisation d'un pointeur sur structure*

```
struct carte X = { 4, pique };      // init. de la variable X
struct carte * pt = &X;             // pt pointe sur X
(*pt).valeur = 1;                   // on change la valeur de X
pt = malloc(sizeof(struct carte)); // pt pointe sur une nouvelle case
(*pt).valeur = 2;
(*pt).col = coeur;
```

Comme on le voit sur l'exemple précédent, lorsqu'on utilise un pointeur sur une structure on le déréférence `*` puis on accède à un champ donné (avec `.`). L'opérateur `->` permet d'effectuer ces 2 opérations directement et d'avoir un code plus clair et lisible :

`pt->champ` est équivalent à `(*pt).champ`

## 3.8 Exemple complet

*Exemple complet*

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

enum couleur { pique, coeur, trefle, carreau };
typedef struct carte {
   short valeur;
   enum couleur col;
} carte;   // définition du type carte

int cmp(void const *a, void const *b) {
   carte const * pa = a;  // conversion implicite, const obligatoire
   carte const * pb = b;  // pareil pour b qui était générique
   if (pa->col > pb->col) return 1;
   if (pa->col < pb->col) return -1;
   if (pa->valeur > pb->valeur) return 1;
   if (pa->valeur < pb->valeur) return -1;
   return 0;
}

int main(void) {
   int i, n = 5;
   srandom(time(NULL));                // initialisation pour random
   carte * const jeu = malloc(n * sizeof(carte));
   if (jeu == NULL) return 1;
   carte * m = jeu;
   while (m != jeu+n) {
      m->valeur = 1 + random() % 13;
      m->col = random() % 4;
      ++m;
   }
   qsort(jeu, n, sizeof(carte), cmp); // appel de qsort
   printf("main: ");
   for (i = 0; i < n; ++i) {
      printf("(%u, %u)  ", jeu[i].valeur, jeu[i].col);
   }
   printf("\n");
   free(jeu);   // obligatoire
   return 0;
}
```

# Chapitre 4

# Structures récursives et POO

L'utilisation conjointe des pointeurs et des structures permet une grande souplesse et une grande efficacité dans la programmation. On voit dans ce chapitre 2 illustrations : la création de structures récursives comme les listes chaînées ou les arbres récursifs et la création d'un code *orienté objet*.

## 4.1 Listes chaînées

La première façon de regrouper et d'organiser des données de même type est d'utiliser un tableau : une zone mémoire où l'on peut accéder aux données directement en utilisant l'opérateur de déréférencement et l'arithmétique des pointeurs. Le coût d'accès à une donnée ne dépend pas de la taille du tableau mais l'insertion ou la suppression d'un élément a un coût linéaire en la taille du tableau.

Une liste chaînée est une façon différente d'organiser des données de même type. Une liste est une succession d'éléments appelés *maillons* qui sont reliés entre eux. Une liste *simplement chaînée* peut se représenter par le graphe suivant : et une liste *doublement chaînée* peut se représenter de la même façon :

*Par convention*, le pointeur `NULL` (pointeur générique d'adresse 0) sera utilisé pour représenter une liste vide ainsi que le maillon suivant le dernier maillon d'une liste non vide.

### 4.1.1 Définition

Dans toute la suite on code une liste d'entiers pour simplifier l'écriture du code. On considère d'abord une structure appelée `maillon` qui contient 2 champs

— un champ `data` de type `int`
— un champ `next` de type `struct maillon *`

Le champ `data` code la valeur de la donnée du maillon (ici une valeur entière mais ce pourrait être tout type de donnée : une autre structure, etc.) et le champ `next` code l'adresse du prochain maillon (celui qui suit dans la liste). La structure `struct maillon` est dite récursive car le champ `next` est l'adresse d'une variable de type `struct maillon` (que l'on est en train de définir).

*Définition de la structure récursive `maillon`*

```
struct maillon {
   int data;
   struct maillon * next;
};
typedef struct maillon maillon;
```

On peut redéfinir `struct maillon` en le type `maillon` pour faciliter l'écriture[1].

1. cela est fait automatiquement en `C99` par les compilateurs modernes.

Une liste simplement chaînée peut se définir comme l'adresse du premier maillon [2]. Ainsi on définit le type `list_int` comme une redéfinition de `maillon *`. Voici un code complet de déclaration d'une liste vide (!) d'entiers :

*Déclaration d'une liste vide d'entiers*

```
#include <stdlib.h>

struct maillon {
   int data;
   struct maillon * next;
};
typedef struct maillon maillon;
typedef maillon * list_int;

int main(void) {
   list_int ma_liste = NULL;
   return 0;
}
```

### 4.1.2 Manipulation de listes chaînées

Pour remplir cette liste vide il faut ajouter des éléments : une liste chaînée se construit toujours de façon itérative. Les deux ajouts conventionnels d'éléments sont les ajouts en début (en-tête) de liste et les ajouts en fin (en queue) de liste.

L'ajout en début de liste est simple : on crée un `maillon` que l'on initialise (l.2-4 de la fonction `push_front` dans le code qui suit) avec la valeur donnée puis que l'on raccorde avec la liste existante (l.5.). On renvoie l'adresse du maillon créé qui devient la valeur de la liste.

*Ajout en début de liste*

```
list_int push_front(list_int ma_liste, int valeur) {
   maillon * nouveau = malloc(sizeof(maillon));
   // test si l'allocation est OK
   nouveau->data = valeur;
   nouveau->next = ma_liste;
   return nouveau;
}
```

Vérifier que le code fonctionne même si la liste donnée en argument est vide.

Pour l'ajout en fin de liste, c'est un peu plus compliqué car il faut parcourir toute la liste. Voici un code possible pour la fonction `push_back`

2. on peut faire un peu plus compliqué en encapsulant cette adresse dans une structure.

*Ajout en fin de liste*

```
list_int push_back(list_int ma_liste, int valeur) {
   maillon * nouveau = malloc(sizeof(maillon));
   // test si l'allocation est OK
   nouveau->data = valeur;
   nouveau->next = NULL;
   if (ma_liste == NULL) return nouveau;
   maillon * pt = ma_liste;
   while (pt->next != NULL)
      pt = pt->next;
   pt->next = nouveau;
   return ma_liste;
}
```

On crée un maillon, puis on parcourt la liste et on affecte le maillon crée comme voisin du dernier maillon existant.

Remarquer que les 4 premières lignes des fonctions `push_back` et `push_front` sont très similaires. Il serait plus élégant de regrouper ces instructions dans une fonction `cree_maillon` qui serait appelée à chaque fois que l'on veut créer et initialiser un maillon.

*Crée un `maillon`*

```
maillon * cree_maillon(int valeur, maillon const * suivant) {
   maillon * a_creer = malloc(sizeof(maillon));
   if (a_creer != NULL) {
      a_creer->data = valeur;
      a_creer->next = suivant;
   }
   return a_creer;
}
```

Après ces fonctions de création il est indispensable de créer une fonction de destruction de liste. Le plus simple est d'appeler une fonction qui retire le premier maillon tant que la liste n'est pas vide.

*Retire le premier élément de la liste et vide la liste*

```
list_int pop_front(list_int ma_liste) {
   if (ma_liste == NULL) return 0;
   maillon * deuxieme = ma_liste->next;
   free(ma_liste);
   return deuxieme;
}

void vide(list_int ma_liste) {
   while (ma_liste != NULL)
      ma_liste = pop_front(ma_liste);
}
```

On peut encore écrire de nombreuses fonctions pour manipuler les listes chaînées. Par exemple les fonctions `insert` et `remove` qui permettent d'insérer ou de retire un maillon de la liste. Ou encore une fonction `echange` qui echange deux maillons de la chaîne. Voici un code possible pour la fonction `insert`.

*Insertion d'un élément après le maillon passé en argument*

```
maillon * insert(maillon * current, int valeur) {
   maillon * nouveau = cree_maillon(valeur, current->next);
   return current->next = nouveau;
}
```

Le code est extrêmement simple et reflète bien la souplesse des structures récursives pour l'insertion et la suppression d'éléments. Insérer un élément dans un tableau est plus complexe et bien plus coûteux : il faut créer une zone mémoire (ou agrandir celle existente) de $n+1$ cases et recopier le tableau avec l'élément supplémentaire.

De même pour supprimer un maillon d'une liste simplement chaînée on peut écrire

*Suppression du maillon suivant celui passé en argument*

```
void remove(maillon * current) {
   maillon *  a_supprimer = current->next;
   current->next = current->next->next;
   free(a_supprimer);
}
```

### 4.1.3 Utilisation

Voici un exemple d'utilisation de notre liste chaînée `list_int`. On veut écrire une fonction qui supprime le plus grand élément d'une liste donnée en argument (liste d'entiers positifs).

*Retire le maximum de la liste*

```
int retire_max(list_int ma_liste) {
   int max = 0;
   if (ma_liste == NULL) return 0;
   maillon * pt_nm1 = ma_liste;
   maillon * pt_n = ma_liste->next;
   maillon * pt_sauv = NULL;
   // on recherche le max
   while (pt_n != NULL) {
      if (pt_n->data > max) {
         max = pt_n->data;
         pt_sauv = pt_nm1;
      }
      pt_n = pt_n->next;
      pt_nm1 = pt_nm1->next;
   }
   remove(pt_sauv);
   return max;
}
```

Exercice : écrire une version de cette fonction avec seulement 2 pointeurs sur `maillon` (sans le pointeur `pt_nm1`).

## 4.2 Arbres récursifs

*Cf. feuille de TD 4.*

## 4.3 Programmation orientée objet en C

On illustre les concepts de la programmation orientée objet en C. Le but est d'acquérir le vocabulaire (constructeur, destructeur, méthodes) et de comprendre les mécanismes qui seront utilisés en C++ (pointeur **this**, etc.).

Comme exemple, on considère la décomposition $p$-adique d'un entier naturel $n$ (où $p$ est un nombre premier)

$$n = \sum_{k=0}^{r-1} a_k p^k, \quad 0 \leq a_k < p \quad \text{et} \quad a_{r-1} \neq 0, \quad \text{et} \quad a_r = 0. \tag{4.1}$$

Pour représenter cette décomposition $p$-adique d'un entier, on propose le type `tp_adic` qui contient les 4 champs suivants : `value` qui code $n$, `p` qui code la base, `r` le nombre de coefficients $(a_k)_{k=0,\ldots,r-1}$, et `coeff` le tableau dynamique des coefficients.

*Définition de la structure*

```
struct tp_adic {
   unsigned value;
   unsigned p, r;
   unsigned * coeff;
};
typedef struct tp_adic tp_adic;
```

Depuis le C99 la redéfinition de type de la ligne 5 n'est plus nécessaire car le **typedef** est automatique (de même en C++).

### 4.3.1 Constructeurs

Pour faciliter l'initialisation d'une variable de type `tp_adic` (qu'on appelera objet `tp_adic`) on va définir une fonction d'initialisation qui renvoie un `tp_adic`. Il y a deux façons de définir un `tp_adic` : à partir des coefficients $(a_k)_{k=0,\ldots,r-1}$ et de la base $p$ ou bien à partir de $n$ et de la base $p$.

Commençons par la première fonction d'initialisation `p_adic`

*Fonction d'initialisation à partir des coefficients*

```
tp_adic p_adic(unsigned p, unsigned r, unsigned a[]) {
   tp_adic result;
   int k;
   result.p = p;
   result.r = r;
   result.value = 0;
   result.coeff = malloc(r * sizeof(unsigned));
   for (k = r-1; k >= 0; k--) {
      result.coeff[k] = a[k];
      result.value += a[k];
      if (k > 0) result.value *= p;
   return result;
};
```

La deuxième fonction d'initialisation peut s'écrire comme suit :

*Fonction d'initialisation à partir de la valeur*

```
tp_adic p_adic_alt(unsigned p, unsigned m) {
   tp_adic result;
   int k, m_tmp = m;
   result.p = p;
   result.value = m;
   result.r = (m == 0) ? 1 : 0;
   while (m_tmp > 0) {
      m_tmp = (m_tmp - m_tmp % p) / p;
      ++(result.r);
   };
   result.coeff = malloc(result.r * sizeof(unsigned));
   for (k = 0; k < result.r; k++) {
      result.coeff[k] = m % p;
      m = (m - result.coeff[k]) / p;
   }
   return result;
};
```

Les fonctions `p_adic` et `p_adic_alt` sont appelées constructeurs : elles sont en charge de l'allocation de la zone mémoire `coeff` et de l'initialisation des champs. Les appels possibles pour la création d'un `tp_adic` sont donc les suivants :

*Appels des constructeurs*

```
unsigned a[] = {1, 2, 14, 7};
tp_adic k = p_adic(19, 4, a);
tp_adic l = p_adic_alt(19, 45);
```

### 4.3.2 Destructeur

Que se passe-t-il si on considère la fonction suivante ?

*Fonction qui crée localement un `tp_adic`*

```
unsigned get_last_coeff(unsigned p, unsigned m) {
   tp_adic n = p_adic_alt(p, m);
   return n.coeff[n.r-1];
};
```

Le code est tout à fait correct, mais à la sortie de cette fonction la variable locale `n` est détruite. La destruction de cette variable correspond à la destruction de l'espace mémoire de la variable `n`. Cependant, le champ `coeff` de la variable `n` contient l'adresse à laquelle sont alloués `r` entiers. Cette adresse est perdue et il n'est plus possible de libérer l'espace mémoire. Il y a une fuite de mémoire ! C'est une erreur importante (non indiquée par le compilateur).

Il faut donc une fonction qui désalloue la mémoire, ce qu'on appelle un destructeur. Appelons cette fonction (unique) `detruit_p_adic` dont le code peut-être :

*Destructeur*

```
void detruit_p_adic(tp_adic * n) {
   free(n->coeff);
};
```

Muni de ce destructeur, on doit écrire la fonction `get_last_coeff` comme ceci :

*Fonction qui crée puis détruit un `tp_adic`*

```
unsigned get_last_coeff(unsigned p, unsigned m) {
   tp_adic n = p_adic_alt(p, m);
   unsigned result = n.coeff[n.r-1];
   detruit_p_adic(&n);
   return result;
};
```

### 4.3.3 Méthodes

On souhaite maintenant associer des fonctions à la structure `tp_adic`. Par exemple la fonction `affiche` qui affiche un objet `tp_adic`, la fonction `incremente` qui ajoute 1 à un objet `tp_adic` et la fonction `change_base` qui modifie $p$. L'idéal serait que ces fonctions soient des membres de l'objet pour permettre les appels suivants :

*Code souhaité*

```
tp_adic n = p_adic_alt(2, 8);
n.affiche(&n);
n.incremente(&n);
n.change_base(&n, p);
```

Commençons par déclarer et définir des fonctions globales qui font le travail :

*Définition des fonctions globales*

```
void affiche_p_adic(tp_adic const * const pt) {
   printf("valeur: %u\n", pt->value);
};
void incremente_p_adic(tp_adic * const pt) {
   pt->value++;
   // modification des coefficients
};
void change_base_p_adic(tp_adic * const pt, unsigned new_p) {
   // code à écrire !!
};
```

Pour « inclure » ces fonctions à la classe `tp_adic`, on utilise des pointeurs sur fonctions. Il faut alors modifier la définition de la structure puis la définition des constructeurs vus précédemment.

*Redéfinition de la structure et des constructeurs*

```
struct tp_adic {
   unsigned value, p, r;
   unsigned * coeff;
   void (*affiche)(struct tp_adic const * const);
   void (*increment)(struct tp_adic * const);
   void (*change_base)(struct tp_adic * const, unsigned);
};
typedef struct tp_adic tp_adic;

tp_adic p_adic(unsigned p, unsigned r, unsigned a[]) {
   tp_adic result;
   result.affiche = affiche_p_adic;
   result.incremente = increment_p_adic;
   result.change_base = change_base_p_adic;
   // reste du code identique
}

// de même pour p_adic_alt
```

### 4.3.4 Pointeur `this` en `C++`

Il est clair que dans l'expression `n.affiche(&n)` il y a une redondance entre l'objet `n` qui appelle la méthode `affiche` et l'argument `&n` qui contient l'adresse de l'objet qui appelle la méthode (l'objet courant pour la méthode).

En `C++`, cette redondance est masquée : l'argument `tp_adic const * const pt` est caché par le système. Dans toute méthode en `C++` il existe un pointeur appelé **`this`** qui correspond à ce pointeur `pt`, c'est à dire qui pointe sur l'objet qui appelle la méthode. Dans tous les cas, le pointeur **`this`** est un pointeur constant. C'est un pointeur constant sur valeur constante si la méthode est qualifiée de **`const`** et un pointeur constant sinon.

# Deuxième partie

# Quelques aspects du C++

# Chapitre 5

# Syntaxe du C++

Un changement majeur du C++ vis à vis du C est la notion de structure. La structure en C++ est enrichie et permet une programmation orientée objet plus aisée et plus élaborée que celle qu'on a vu en C. En plus des champs usuels, on peut déclarer des fonctions membres (appelées *méthodes*) pour une structure. Le mot-clé **struct** coexiste avec le mot-clé **class** pour déclarer ces structures appelées aussi classes. Une variable de type **struct** ou **class** sera appelée *objet*. On verra tout cela en détail dans le chapitre suivant, mais on commence par les changements syntaxiques entre le C et le C++.

## 5.1 Premiers exemples

On considère le code suivant écrit dans un fichier test1.cpp.

*Fichier test1.cpp*

```
#include <iostream>
int main() {
    std::cout << "Bonjour" << std::endl;
    return 0;
}
```

### Commentaires

**ligne 1 :** on indique via le préprocesseur (qui est le même qu'en C) qu'on a besoin du fichier de fonctions déclarées dans le fichier système iostream.h (on n'indique pas le .h ou .hpp en C++). Ce fichier d'en-tête (header) contient les fonctions standards d'entrées-sorties (**i**nput **o**utput **stream**), c'est à dire les fonctions qui lisent ou qui écrivent sur la ligne de commande, un flux.

**ligne 2 :** à la différence du C une fonction qui ne prend pas d'argument, comme ici la fonction main, peut se déclarer sans le mot-clé **void** mais directement **int** main().

**ligne 3 :** utilisation de l'opérateur << entre std::cout et la chaîne de caractères "Bonjour" puis entre le résultat de cette opération et std::endl. Cette instruction écrit sur la console le mot Bonjour. En fait on envoie << la chaîne Bonjour vers l'objet std::cout (variable de la classe ostream définie dans iostream). Puis on envoie le caractère std::endl défini dans iostream qui correspond à un saut de ligne.

### Compilation

Comme en C, la compilation se passe en 3 phases : préprocesseur, compilation en code objet, édition de liens et création d'un executable. Le programme effectuant la compilation en C++ est g++. Il s'utilise exactement de la même façon que gcc.

Pour vérifier qu'il n'y a pas d'erreur de syntaxe et que l'appel des fonctions et des opérateurs est correct, on effectue la phase de compilation (et d'assemblage) qui produit un fichier objet. Ainsi, on

tape la commande

```
$ g++ -c test1.cpp
```

Pour la création de l'executable `prog1` on utilisera la commande

```
$ g++ test1.cpp -o prog1
```

Voici une légère variante de l'exemple précédent.

*Fichier test2.cpp*

```
#include <iostream>
#include <string>
int main() {
    std::string c = "Bon";
    std::cout << c + "jour" << std::endl;
    return 0;
}
```

### Commentaires

**ligne 3 :** on déclare `c` de type `std::string`. Le type `std::string` est une classe définie dans le header `string.h` et permet une manipulation aisée des chaînes de caractères. La variable `c` est un objet initialisé avec `"Bon"` (appel du constructeur de la classe `std::string` avec l'argument `"Bon"`).

**ligne 4 :** la première opération de cette instruction est l'opérateur `+` (prioritaire sur `<<`) entre l'objet `c` et la constante `"jour"`. En fait, la chaîne `"jour"` est convertie implicitement en objet `std::string` (par un appel du constructeur) puis l'opérateur `+` agit entre 2 objets de la classe `std::string` en effectuant la concaténation des 2 chaînes de caractères.

## 5.2 Différences syntaxiques avec le C

### 5.2.1 Espace de nom

Une première nouveauté est la notion d'espace de nom ou *namespace*. Un espace de nom regroupe une partie de code contenant aussi bien des déclarations/définitions de variables, de types ou de fonctions (à la différence d'un bloc). Un espace de nom possède un identifiant et toutes les fonctions, les types et les variables/objets du C++ standard sont dans l'espace de nom `std`.

Il y a deux façons d'accèder à un élément d'un espace de nom :

— **localement** en utilisant l'opérateur de résolution de portée `::` entre l'identifiant de l'espace de nom et l'élément. Par exemple, `std::cout` signifie qu'on accède à l'élément `cout` (ici un objet) de l'espace de nom `std`.

— **globalement** en utilisant l'instruction **`using namespace`** suivit de l'identifiant de l'espace de nom. Tout ce qui suit cette instruction est recherché dans l'espace de nom donné.

Par exemple le premier code d'exemple peut se réécrire de la façon suivante :

*Fichier test1.cpp équivalent au premier*

```
#include <iostream>
using namespace std;
int main() {
    cout << "Bonjour" << endl;
    return 0;
}
```

Un espace de nom `e1` peut en contenir un second `e2`. Pour accéder à un élément `a` de `e2` on utilise 2 fois l'opérateur de résolution de portée `::`. L'expression est donc `e1::e2::a`.

Il est aussi possible de définir ses propres espaces de nom.

### 5.2.2 Déclaration de variables

Le placement des déclarations de varaibles est plus souple en `C++` qu'en `C`. Il est possible de déclarer une variable dès qu'on en a besoin, partout où on attend une instruction.

Par exemple on peut écrire en `C++`

*Déclaration à la volée dans une boucle* ***for***

```
for (int i = 0; i < 10; ++i) {
    // instructions
}
```

La variable `i` est considérée comme locale dans le bloc de la boucle **`for`**. Après l'accolade fermante, la variable `i` n'existe plus.

### 5.2.3 Fonctions : surcharge et valeurs par défaut des arguments

La notion de fonction est légèrement différente en `C++`. Une première différence est que l'identification repose sur sa signature.

**Définition 8** *La* ***signature*** *d'une fonction est composée de son nom, du type de ses arguments et d'un éventuel qualifieur (***`const`***, ***`static`***, etc.) si c'est une méthode (fonction d'une classe).*

La signature est donc différente du prototype.

**Définition 9** *Une* ***fonction*** *est un espace réservé en mémoire qui possède les caractéristiques suivantes :*

- sa signature *(unique) qui permet de l'identifier,*
- son prototype*, qui indique le type des arguments et le type de retour,*
- son contenu*, c'est la séquence de bits qui code la fonction,*
- son adresse*, c'est l'endroit dans la mémoire où elle est stockée.*

#### Surcharge

Si plusieurs fonctions (donc de signatures différentes) ont le même nom, on dit qu'il y a surcharge du nom de la fonction.

Ainsi les 3 fonction suivantes peuvent être définies dans un même programme :

*Surcharge de la fonction* `puissance`

```
int puissance(int a, int n);
double puissance(double x, int n);
double puissance(double x, double y);
```

Les appels suivants sont tous corrects. C'est le compilateur qui choisit la bonne fonction en fonction des types des arguments. S'il y a litige, il y a erreur à la compilation.

*Différents appels de la fonction* `puissance` *(des fonctions de nom* `puissance`*)*

```
int x = puissance(2, 4);
double y = puissance(3.14, 4);
double z = puissance(3.14, 1.5);
```

#### Valeurs par défaut des arguments

A la déclaration, on peut attribuer des valeurs par défaut aux derniers arguments d'une fonction (les derniers étants ceux le plus à droite dans l'écriture du prototype). Une valeur par défaut est une valeur utilisée pour initialiser un argument si celui-ci est omis lors de l'appel de la fonction. Une fonction peut avoir tous ses arguments prenant une valeur par défaut.

Voici un exemple :

*Fonction `f` avec 2 arguments prenant des valeurs par défaut*

```
double f(double a, double y = 3.14, int * pt = NULL); // déclaration
// ...
f(2, 3, &n); // appel classique possible (n est un int déclaré avant)
f(2, 2.78);  // appel de f(2, 2.78, NULL)
f(2);        // appel de f(2, 3.14, NULL)
```

Attention, l'ordre des arguments ne peut pas être modifié. Par exemple l'appel `f(2, &n)` est invalide car le deuxième argument doit être un **`double`**.

### 5.2.4 Inlining

Une fonction peut être déclarée **`inline`**. Cela modifie le mécanisme utilisé au moment de l'appel de la fonction. Le comportement classique est l'envoi des arguments vers le code de la fonction qui s'execute sur une copie des arguments et qui renvoie le résultat. Dans le cas d'une fonction déclarée **`inline`**, le code (les instructions) de la fonction est recopié [1] à la place de l'appel : cela permet d'avoir un mécanisme d'appel plus rapide. Le désavantage de ce mécanisme est que le code produit est plus gros et peut-être moins efficace.

Il faut réserver la déclaration **`inline`** pour de petites fonctions (1 ou 2 instructions) appelées très souvent (ce sera le cas en POO). Les autres fonctions ne doivent pas être **`inline`**.

*Syntaxe de déclaration d'une fonction inline*

```
inline prototype;
```

Si on définit la fonction après sa déclaration, on ne remet pas le mot-clé **`inline`** devant la définition.

## 5.3 Nouveaux opérateurs

### 5.3.1 Résolution de portée

L'opérateur de résolution de portée `::` apparaît dans différents contextes. Il est utilisé lorsqu'on veut accéder à un élément (type, fonction, objet, variable,...) d'un espace de nom ou d'une structure (ou une classe, cf. chapitre suivant).

*Syntaxe de l'opérateur de résolution de portée*

```
nom::element
```

### 5.3.2 Injection, extraction

Les entrées-sorties en `C++` reposent sur des objets (des classes `ostream`, `istream`, `ofstream`, `ifstream`, etc.), appelés *flux*, et sur les opérateurs d'injection `>>` et d'extraction `<<`. On injecte une expression dans un flux de sortie (écriture) et on extrait une valeur d'un flux d'entrée (lecture).

1. ici on simplifie, le recopiage est plus intelligent que celui fait par le préprocesseur si on avait écrit une `MACRO`.

*Syntaxe des opérateurs de flux*

```
flux << expression  // opérateur d'injection (écriture)
flux >> variable  // opérateur d'extraction (lecture)
```

### 5.3.3 Allocation dynamique de mémoire

L'allocation de mémoire doit se faire en C++ en utilisant l'opérateur **new** et non la fonction malloc. De même, pour libérer la mémoire après utilisation, on utilise l'opérateur **delete** et non la fonction free.

Il y a 2 syntaxes, l'une pour la création d'une seule case mémoire d'un type donné et l'autre pour la création d'un tableau de $n$ cases mémoires d'un type donné.

*Syntaxe pour allouer 1 case mémoire*

```
type * pt = new type;
// utilisation de la zone mémoire pt
delete pt;
```

*Syntaxe pour allouer n cases mémoires*

```
type * pt = new type[n];
// utilisation de la zone mémoire pt
delete [] pt;
```

Attention il faut bien utiliser l'opérateur **delete** suivit des crochets droits [] si l'on doit libérer plusieurs cases mémoires.

## 5.4 Nouveaux types

### 5.4.1 Type bool

Il s'agit simplement du type utilisé pour coder une expression booléenne *ie* une expression qui est évaluée comme vraie ou fausse. Une variable de type **bool** prend donc 2 valeurs possibles : **true** pour signifié vrai et **false** pour signifié faux. En particulier les opérateurs de comparaisons en C++ retournent une expression de type **bool** et les opérateurs de comparaisons agissent entre des expressions de type **bool**.

### 5.4.2 Type référence

Les références sont un nouveau moyen, avec les pointeurs qui existent toujours, de manipuler des variables à travers leur adresse. Les références sont moins souples mais plus sûres que les pointeurs et doivent être utilisées partout où cela est possible.

On rappelle qu'un pointeur est associé uniquement à un ***type*** : il peut être initialisé à partir de l'adresse d'une variable (ou fonction) ou bien en demandant une allocation mémoire (création d'une variable anonyme).

Une référence est associé à un ***type*** et à une variable. A la déclaration, une référence doit être initialisée à partir d'une variable existente : elle correspond alors à son adresse. Cependant, à l'utilisation la référence se comporte exactement comme la variable référencée : pas besoin des opérateurs * et &.

*Syntaxe de déclaration/initialisation d'une référence*

```
type & nom_reference = var;
```

vlc 'Une référence peut être vue comme un synonyme de la variable référencée. Après sa déclaration, agir sur la référence revient à agir sur la variable référencée. Voici un exemple :

*Exemple d'une référence sur un* ***int***

```
int x = 10;
int & r = x; // initialisation de la reference r sur x
r = 2;       // utilisation de r comme x
```

A la fin de ce code `x` vaut 2 et `r` est toujours une référence sur `x`.

### Passage par adresse

Une première possibilité est de forcer le passage par adresse des arguments d'une fonction en utilisant les références.

*Références pour forcer le passage par adresse*

```
void echange(int & a, int & b) {
   int tmp = a;
   a = b;
   b = tmp;
}

int main(void) {
   int x = 10, y = 15;
   echange(x, y);
   //...
}
```

### Fontion renvoyant une référence

Une fonction qui renvoie une référence permet de se trouver à gauche d'une affectation. D'ordinaire, les éléments se trouvant à gauche d'une affectation sont les variables dont le contenu doit être modifié. A priori une fonction renvoie une expression, et cette expression n'est pas une variable donc ne peut pas être affecté d'une nouvelle valeur.

Cependant, une fonction qui renvoie une référence initialise la variable de sortie (de type référence) avec l'expression (qu'on suppose être une variable `result`) renvoyée par la fonction. Après cet appel la référence peut-être affectée d'une nouvelle valeur, ce qui a pour effet d'affecter une nouvelle valeur à `result` (il faut que cette variable existe encore).

Prenons l'exemple suivant : on suppose qu'il existe 2 variables globales `etudiants` et `notes` qui sont des tableaux respectivement de `string` et de `int`. On veut écrire une fonction `note` qui à un nom d'étudiant dans le tableau des `notes` modifie la note correspondante dans le tableau des `notes`. On veut donc le comportement suivant :

*Références pour forcer le passage par adresse*

```
string etudiants[N]; // tableau initialisé
int notes[N];
note("marcel") = 12;
```

Il suffit pour cela d'écrire la fonction `note` comme renvoyant une référence sur la bonne case mémoire du tableau `notes`.

*Références pour forcer le passage par adresse*

```
int & note(string nom) {
   for (i = 0; i < N; ++i)
      if (etudiants[i] == nom)
         return notes[i];
}
```

Il est recommandé de bien comprendre cet exemple.

#### Référence constante

Une référence est toujours constante car reliée définitivement à une variable existente. Par abus on parle de référence constante pour une référence sur variable constante. C'est à dire que la référence ne pourra pas modifier la variable référencée. Une référence constante permet par exemple de forcer le passage par adresse mais de s'assurer que la fonction ne modifiera pas la variable référencée. Cela est très utilisé pour passer des objets à des fonctions : il est plus rapide de recopier une adresse que de recopier un objet en entier (qui peut être volumineux en termes d'octets).

Les 2 syntaxes équivalentes sont les suivantes :

*Syntaxe de déclaration/initialisation d'une référence constante*

```
type const & nom_reference = var;
const type & nom_reference = var;
```

Par exemple, pour une fonction `f` qui doit juste accéder en lecture à un objet `string`, on utilisera le mécanisme de passage par adresse tout en s'assurant que la fonction `f` ne modifie pas cet objet.

*Exemple d'utilisation d'une référence constante*

```
char f(string const & s) {
   return s[0];
}
```

### 5.4.3 Classes de la librairie standard

(vu en TP, pas vu en cours)

#### Classe `string`

#### Classe `complex`

#### Classe `valarray`

# Chapitre 6

# Définition de classes et POO

## 6.1 Champs et méthodes

Le type **struct** a considérablement évolué en C++. C'est toujours un type nommé mais il y a une redéfinition automatique de type donc après la déclaration d'un type **struct** NomStructure on peut déclarer une variable de type NomStructure. Une variable de type structure sera appelé un *objet*.

En plus des champs de données qui constituaient une structure en C on peut y ajouter des fonctions qui ont pour but d'agir sur ces données. De telles fonctions sont appelées méthodes. Pour ajouter une méthode à une structure il suffit déclarer son prototype dans la définition de la structure :

*Définition d'une structure avec 2 méthodes (dans un fichier header)*

```
struct Point {
   int x, y;
   void deplace(int, int);
   void affiche();
};
```

La définition des méthodes deplace et affiche peut être faite au moment de la déclaration (dans la structure) mais dans ce cas, les appels seront **inline**. Cela est recommandé que pour les petites fonctions appelées souvent. Dans le cas contraire, *ie* le plus souvent, il faut définir les méthodes en dehors de la structure. Pour cela on doit utiliser l'opérateur de résolution de portée. En effet le « vrai » nom de la méthode deplace de la classe Point est Point::deplace, et le vrai nom de la méthode affiche est Point::affiche.

Les méthodes peuvent accéder aux champs de la structure directement par leurs noms. Ces champs sont comme des variables globales pour les méthodes.

*Définition des 2 méthodes de la structure Point (dans un fichier compilable)*

```
void Point::deplace(int a, int b) {
   x = a; y = b;
}
void Point::affiche() {
    std::cout << "(" << x << "," << y << ")" << std::endl;
}
```

L'appel d'une méthode se fait exactement comme l'accès à un champ : on utilise l'opérateur . à partir de l'objet ou avec l'opérateur -> à partir de l'adresse d'un objet.

*Utilisation d'un objet* `Point`

```
Point P;
P.deplace(2, 5);
P.affiche();
```

### 6.1.1 Public et privé

Il est souvent important de préserver la cohérence entre les champs d'un objet. Reprenons par exemple la structure `p_adic` vue en 4.3 adaptée en `C++`.

*Structure* `p_adic` *vue en* `C` *adaptée en* `C++`- première version -

```
struct p_adic {
   unsigned value, p, r;
   unsigned * coeff;
   void affiche();
   void increment();
   void change_base(unsigned);
};
```

Les pointeurs sur fonctions ont été remplacés par les méthodes `affiche`, `increment` et `change_base`. Il faudrait aussi écrire la fonction d'initialisation qui sera remplacée en `C++` par le constructeur (cf. plus loin). On rappelle qu'il y a une relation étroite entre les champs `value` (qui code $n$), `p`, `r` et `coeff` (qui code les $(a_k)_{k=0,\dots,r-1}$) qui doivent vérifier

$$n = \sum_{k=0}^{r-1} a_k p^k.$$

Afin de préserver la cohérence entre les champs d'un objet, on va interdir l'accès à ces champs au reste du programme. La seule façon d'interagir avec l'objet se fera à l'aide de méthodes accessibles. Il faudra donc définir des méthodes pour chaque besoin d'interaction avec un objet. Cette règle de séparation entre champs inaccessibles (cachés) et méthodes accessibles (visibles) est appelée *encapsulation* en POO.

**Définition 10** *Un champ ou une méthode déclaré* ***`public`*** *est accessible dans tout le programme (autres fonctions, autres méthodes d'autres classes, etc.).*
*Un champ ou une méthode déclaré* ***`private`*** *n'est accessible que pour les méthodes de la classe, ou pour des méthodes ou fonction amies de la classe (cf. plus loin la notion d'amitié* ***`friend`****).*

**Définition 11** *Dans une structure déclarée avec le mot-clé* ***`struct`****, tout est* ***`public`*** *par défaut.*
*Dans une structure déclaré avec le mot-clé* ***`class`****, tout est* ***`private`*** *par défaut. On appelera une structure déclarée ave le mot-clé* ***`class`*** *une* classe.

En dehors de cette différence, il n'y a pas d'autres différences entre une structure et une classe.

Voici deux façons équivalentes de définir la classe `p_adic` en protégeant les champs de la classe.

*Structure* `p_adic` *vue en* `C` *adaptée en* `C++`- seconde version -

```
class p_adic {
   unsigned value, p, r;
   unsigned * coeff;
   public:
      void affiche();
      void increment();
      void change_base(unsigned);
};
```

*Structure `p_adic` vue en `C` adaptée en `C++`-* seconde version (bis) -

```
class p_adic {
   public:
      void affiche();
      void increment();
      void change_base(unsigned);
   private:
      unsigned value, p, r;
      unsigned * coeff;
};
```

Ainsi dans le reste du programme (en dehors des méthodes de la classe `p_adic`) si `n` est un objet de la classe `p_adic`, les expressions `n.value`, `n.p`, `n.r` et `n.coeff` sont interdites. Par contre les expressions `n.affiche()`, `n.increment()`, et `n.change_base(p2)` sont autorisées (*a priori*).

### 6.1.2 Pointeur `this`

Dans chaque méthode existe un pointeur appelé **`this`** qui pointe sur l'objet courant, c'est à dire l'objet à partir duquel la méthode est appelée. Pour comprendre d'où vient ce pointeur, il faut relire la partie POO en `C`. La définition de la méthode `deplace` peut s'écrire comme ceci :

*Définition de la méthode `deplace` à l'aide du pointeur **`this`***

```
void Point::deplace(int a, int b) {
   this->x = a;
   this->y = b;
}
```

*Dans toutes les méthodes, le pointeur **`this`** est un pointeur constant.*

### 6.1.3 Méthode constante

Dans une méthode qualifiée de **`const`**, le pointeur **`this`** est un pointeur constant vers valeur constante. Il s'agit donc d'une méthode qui ne peut pas modifier les champs de l'objet courant. Par exemple dans la classe `p_adic` la méthode `affiche` doit être qualifiée de constante. Pour cela on fait suivre le prototype de mot-clé **`const`** (dans la déclaration et dans la définition). De plus, le mot-clé **`const`** fait partie de la signature de la fonction.

*Structure `p_adic` vue en `C` adaptée en `C++`-* troisième version -

```
class p_adic {
   public:
      void affiche() const;
      void increment();
      void change_base(unsigned);
   private:
      unsigned value, p, r;
      unsigned * coeff;
};
// dans un autre fichier, la définition de p_adic::affiche() const:
void p_adic::affiche() const {
   // contenu de la méthode
};
```

Une méthode `const` et `public` peut être appelée dans tout le reste du programme. Une méthode non qualifiée de `const` n'est pas utilisable sur un objet déclaré comme `const`. Par exemple, dans la fonction de signature `f(p_adic const & n)` on peut utiliser la méthode `affiche` (l'expression `n.affiche()` est autorisée) mais pas les méthodes `increment` et `changer_base` qui modifient l'objet référencé par `n`.

## 6.2 Méthodes particulières

### 6.2.1 Constructeurs

**Définition 12** *Un* constructeur *est une méthode qui a le même nom que la classe, qui ne renvoie rien et qui n'a pas de type de retour spécifié dans son prototype.*

Un constructeur a pour but de créer un objet (allouer la zone mémoire pour les différents champs) et d'initialiser les champs. Il peut coexister plusieurs constructeurs de signatures différentes. L'initialisation des champs doit se faire en respectant l'ordre de déclaration des champs.

#### Constructeur avec argument(s)

Un constructeur pour la classe `Point` doit initialiser les champs `x` et `y`. Comme toute fonction, un constructeur peut prendre des valeurs par défaut pour ses argumenents. On définit ci-dessous le constructeur qui prend 2 arguments, mais qui peut être appelé avec 1 ou 0 argument.

*Constructeur pour la classe `Point`*

```
class Point {
   public:
      Point(int a = 0, int b = 0) { x = a; y = b; };
   private:
      int x, y;
};
```

Voici des exemples d'appel de ce constructeur

*Constructeur pour la classe `Point`*

```
Point A(2, 3);  // appel avec 2 arguments
Point B(1);     // appel avec 1 argument; équivalent à Point B(1, 0)
Point C = 1;    // appel avec 1 argument; équivalent à Point C(1, 0)
Point D;    // appel sans argument
Point * Q = new Point(2, 3);
Point * R = new Point;
Point T[10];
```

Noter la syntaxe particulière de la ligne `3`.

#### Constructeur par défaut

**Définition 13** *Le* constructeur par défaut *est le constructeur qui peut être appelé sans argument : soit il ne prend pas d'argument soit ils ont tous des valeurs par défaut.*

Si aucun constructeur n'est défini, alors un constructeur par défaut est généré par le compilateur initialisant tous les champs à `0`, `NULL` ou avec leur constructeur par défaut. Dès qu'un constructeur est défini dans une classe, le constructeur par défaut automatique n'est plus généré par le compilateur : il est préférable d'en écrire un !

Le constructeur écrit précédemment pour la classe `Point` est un constructeur par défaut. Les appels de l'exemple précédent pour initialiser `D`, `R` et `T` utilisent le constructeur par défaut.

Considérons maintenant la classe `p_adic` suivante :

*Classe `p_adic` sans constructeur par défaut*

```
class p_adic {
   public:
      p_adic(unsigned p, unsigned valeur = 0);
      p_adic(unsigned p, unsigned r = 0, coeff = NULL);
   private:
      unsigned valeur, p, r;
      unsigned * coeff;
};
```

Dans cette classe, il n'y a pas de constructeur par défaut. En effet, 2 constructeurs sont définis mais ils prennent au moins un argument : l'entier `p`. Les appels suivants sont donc impossible

*Appels impossibles sans constructeur par défaut*

```
p_adic n;                  // incorrect
p_adic * pt = new p_adic; // incorrect
p_adic pt[10];             // incorrect
```

### Constructeur de copie / de clonage

**Définition 14** *Le* constructeur de copie*, ou de clonage, est le constructeur qui prend une référence constante sur un objet de la classe et qui crée l'objet courant comme une copie de l'objet référencé.*

Si aucun constructeur de copie n'est défini, alors un constructeur de copie est généré par le compilateur. Cette copie correspond à une copie « bit à bit » ce qui peut provoquer des erreurs mémoires dans la suite du programme si des champs pointent vers des zones mémoires extérieures à l'objet (revoir la partie POO en `C`).

Dans la classe `Point` le constructeur de copie généré par le compilateur est correct. Dans la classe `p_adic` il faut réécrire une version.

*Classe `p_adic` avec constructeur de copie valide*

```
class p_adic {
   public:
      p_adic(unsigned p, unsigned valeur = 0);
      p_adic(unsigned p, unsigned r = 0, coeff = NULL);
      p_adic(p_adic const & n);    // décl. constructeur de copie
   private:
      unsigned valeur, p, r;
      unsigned * coeff;
};
// définition dans un fichier compilable:
p_adic::p_adic(p_adic const & n) {
   valeur = n.valeur;
   p = n.p;
   r = n.r;
   coeff = new unsigned[r]; // et non pas coeff = n.coeff
   for (int i = 0; i < r; ++i)
      coeff[i] = n.coeff[i];
};
```

Il est possible d'appeler le constructeur de copie des deux façons suivantes :

*Appels du constructeur de copie*

```
p_adic m = n; // m créé à partir de n
p_adic q(n);  // q créé à partir de n
```

### Liste d'initialisation

Une syntaxe *réservée aux constructeurs* permet d'initialiser les champs d'un objet. Cette syntaxe est nécessaire pour initialiser des champs objets d'une classe sans constructeur par défaut, ou des champs de type référence ou encore des champs constants. Sans être nécessaire pour les autres champs, cette syntaxe est utile car concise et élégante.

Pour comprendre, considérons une classe `paire` dans laquelle on déclare 2 champs `n` et `m` qui sont des objets de la classe `p_adic` définie précédemment sans constructeur par défaut. Voici cette classe :

*Classe `paire`*

```
class paire {
   public:
      paire(p_adic const & a, p_adic const & b);
   private:
      p_adic n, m;
};
```

A priori il n'est pas possible de définir le constructeur de cette classe. Une définition telle que

*Définition naïve du constructeur : version incorrecte*

```
paire::paire(p_adic const & a, p_adic const & b) { n = a; m = b; }
```

serait érroné car l'expression `n = a` présuppose que le champ `n` a été créé : on affecte la valeur `a` au champ `n`. Si un constructeur par défaut pour la classe `p_adic` existe, alors il y a eu un appel avant même la première instruction du constructeur `paire` et le champ `n` existe. Sinon le champ n'a pas été construit.

Pour agir avant la première instruction du constructeur `paire`, au moment de la création des champs en mémoire, on utilise une liste d'initialisation qui appelle les constructeurs des différents champs avec des arguments si nécessaire. Pour initialiser un champ on appelle le constructeur après le prototype du constructeur suivi du caractère `:` et avant le bloc de définition `{` qui peut être vide s'il ne reste plus rien à faire ou avec des instructions supplémentaires. L'initialisation de plusieurs champs se fait de la même façon, les appels des différents constructeurs sont séparés par une virgule.

Par exemple le constructeur de la classe `paire` peut s'écrire :

*Définition du constructeur `paire` : version correcte*

```
paire::paire(p_adic const & a, p_adic const & b) : n(a), m(b) {}
```

De même, on peut écrire le code suivant pour définir la classe `paire`.

*Classe `paire` avec 2 constructeurs corrects*

```
class paire {
   public:
      paire(p_adic const & a, p_adic const & b) : n(a), m(b) {}
      paire(unsigned p, unsigned val1, unsigned val2)
         : n(p, val1), m(p, val2) {}
   private:
      p_adic n, m;
};
```

Par abus, on peut aussi initialiser un champ d'un type primitif en utilisant cette même notation d'appel de constructeur. Par exemple

*Exemple d'un constructeur avec liste d'initialisation*

```
class A {
   public:
      A(int a, int b) : x(a), r(a), u(a, b) {}
   private:
      double x;
      int & r;
      const std::complex<double> u;
};
```

Sans la liste d'initialisation, est-il possible d'initialiser le champ `r` ? et le champ `u` ?

### 6.2.2 Destructeur

**Définition 15** *Le destructeur est la méthode du même nom que la classe précédée d'un tilde* `~`*, qui ne renvoie rien, ne possède pas de type de retour et ne prend aucun argument.*

Le destructeur est en charge de la destruction en mémoire de l'objet. S'il n'est pas défini dans une classe, une version est générée par le compilateur. Cette version sera correcte s'il n'y pas de champ qui pointe vers une zone mémoire allouée pour un objet de la classe.

Pour résumé, dès qu'on redéfinit le constructeur de copie (avec au moins un appel de **`new`**), on doit redéfinir le destructeur (avec les appels de **`delete`** nécessaires). Par exemple pour la classe `p_adic` on écrit

*Classe `p_adic` avec destructeur valide*

```
class p_adic {
   public:
      p_adic(unsigned p, unsigned valeur = 0);
      p_adic(unsigned p, unsigned r = 0, coeff = NULL);
      p_adic(p_adic const & n);
      ~p_adic() { delete [] coeff; }    // destructeur
   private:
      unsigned valeur, p, r;
      unsigned * coeff;
};
```

### 6.2.3 Accesseurs, mutateurs

On appelle accesseur une méthode qui permet d'accéder à un champ en lecture seulement. Cette méthode doit donc être qualifiée de **`const`**. On appelle mutateur une méthode qui permet d'accéder à un champ en écriture : elle sera non **`const`** et renverra une référence sur le champ en question.

Par exemple, on ajoute la méthode `X() const` pour accéder au champ `x` de la classe `Point` en lecture et on ajoute la méthode `X()` pour y accéder en écriture. Les deux méthodes ont même nom mais des signatures différentes.

*Accesseur et mutateur du champ `x`*

```
class Point {
   public:
      Point(int x, int y) : x(x), y(y) {}
      int X() const { return x; } // accesseur
      int & X() { return x; }     // mutateur
   private:
      int x, y;
};
```

Les différents appels possibles sont les suivants :

*Appels de l'accesseur et du mutateur du champ `x`*

```
Point P(2, 3);
P.X() = 5;       // utilisation du mutateur
cout << P.X();   // utilisation de l'accesseur
```

Quelle la différence fondamentale avec le fait de déclarer le champ `x` comme **`public`** ?

## 6.3 Fonctions amies

**Définition 16** *Une* fonction amie *d'une classe peut accéder aux champs et méthodes* ***`private`*** *d'un objet de cette classe.*

Il faut ajouter le prototype de la fonction amie précédé du mot-clé **`friend`** dans la définition de la classe. Attention, cela n'en fait pas une méthode, il n'y pas de pointeur **`this`**.

Par exemple pour faire d'une fonction **`double`** `f(Point` **`const`** `&` `P,` **`double`** `x)` une amie de la classe `Point` on l'indique dans la classe `Point`

*Déclaration de la fonction `f` amie de la classe `Point`*

```
class Point {
   friend double f(Point const & P, double z); // décl. d'amitié !
   public:
      Point(int x, int y) : x(x), y(y) {}
   private:
      int x, y;
};
// définition dans un fichier compilable
double f(Point const & P, double z) {
   return P.x + z;     // P.x est autorisé car f est amie
}
```

## 6.4 Surcharge des opérateurs

Il est possible de redéfinir la majorité des opérateurs usuels pour qu'ils soient utilisables avec des objets d'une classe que l'on définit. On dit alors qu'on surcharge un opérateur pour la classe. Par exemple, dans le cas de la classe `Point` on aimerait faire l'addition entre 2 objets `Point P` et `Q` en utilisant l'expression suivante : `P + Q`. De même, on aimerait recopier `Q` dans `P` en utilisant l'expression `P = Q`, *i.e.* l'opérateur d'affectation `=` entre `P` et `Q`.

Les seuls opérateurs[1] qu'on ne peut pas surcharger sont : `::` `.` `.*` `?:` **`sizeof`** **`typeid`**.

1. vus dans ce cours, il faudrait ajouter les opérateurs de conversion **`static_cast`**, **`dynamic_cast`**, **`const_cast`**, **`reinterpret_cast`**

### 6.4.1 Par une méthode

On peut surcharger un opérateur par une méthode dont le nom est **operator** suivi du (ou des) symbole(s) de l'opérateur, par exemple **operator=**, **operator+**, **operator[]**, etc. Il faut connaître la règle d'appel suivante :

— si l'opérateur est unaire : l'appel de l'opérateur appliqué à un objet P correspond à l'appel de la méthode correspondante de l'objet P.
— si l'opérateur est binaire : l'appel de l'opérateur appliqué entre 2 objets P et Q (P à gauche de l'opérateur et Q à droite) correspond à l'appel de la méthode de l'objet P avec l'objet Q en argument.

#### Opérateur d'affectation

L'opérateur d'affectation = doit être surchargé dès qu'on doit redéfinir l'opérateur de copie et le destructeur. En effet, pour les mêmes raisons il peut y avoir des erreurs mémoires si on ne le redéfinit pas. L'opérateur d'affectation existe même s'il n'est pas écrit par le programmeur. Une version est générée par le compilateur par défaut.

Dans le cas de la classe Point, il n'est pas nécessaire de redéfinir l'opérateur d'affectation. Dans le cas de la classe p_adic, il faut le redéfinir. Le code est très semblable à l'opérateur de copie mais il faut gérer le cas particulier de l'auto-affectation, c'est à dire l'appel n = n si n est un objet p_adic. Pour cela on compare les adresses des 2 objets.

*Classe p_adic avec opérateur d'affectation valide*

```
class p_adic {
   public:
      p_adic(unsigned p, unsigned valeur = 0);
      p_adic(unsigned p, unsigned r = 0, coeff = NULL);
      p_adic(p_adic const & n);
      ~p_adic() { delete [] coeff; }
      p_adic & operator=(p_adic const & n);
   private:
      unsigned valeur, p, r;
      unsigned * coeff;
};
// définition dans un fichier compilable:
p_adic & p_adic::operator=(p_adic const & n) {
   if (&n == this) return *this;
   valeur = n.valeur;
   p = n.p;
   r = n.r;
   delete [] coeff;
   coeff = new unsigned[r]; // et non pas coeff = n.coeff
   for (int i = 0; i < r; ++i)
      coeff[i] = n.coeff[i];
   return *this;
};
```

L'opérateur d'affectation doit renvoyé une référence sur un p_adic initialisée avec l'objet courant ***this**. Cela permet d'enchainer plusieurs opérateurs d'affectation et d'écrire le code suivant

*Exemple d'appel de l'opérateur d'affectation*

```
p_adic n(2, 15), p(2), q(2);
p = q = n;
p_adic r = n; // attention ici ce n'est pas l'opérateur d'affectation
              // c'est le constructeur de copie !!
```

### Opérateur d'indexation

L'opérateur d'indexation `[]` doit respecter la règle suivante : il prend un unique argument entier. En fait comme pour les accesseurs et mutateurs, on définit généralement 2 opérateurs d'indexation : l'un par une méthode **`const`** pour la lecture seule et l'autre pour l'écriture (qui renvoie une référence).

Par exemple, on peut ajouter l'opérateur `[]` à la classe `p_adic` pour récupérer la valeur d'un coefficient. Seule la lecture est ici autorisée.

*Définition d'un opérateur d'indexation*

```
class p_adic {
   public:
      ...
      unsigned operator[](int k) const {
         return k < r ? coeff[k] : 0; }   // lecture uniquement
   private:
      unsigned valeur, p, r;
      unsigned * coeff;
};
```

### Opérateur fonctionnel

L'opérateur fonctionnel `()` est très souple et la méthode **`operator()`** utilisée pour le définir peut être de n'importe quel prototype. Il sera très utilisé dans la suite du cours pour définir des objets fonctionnels (fonctors) qui ont pour but de remplacer des fonctions.

### Opérateurs d'incrément/décrément

On rappelle qu'il existe 2 opérateurs d'incrément : l'opérateur préfixé qui agit avant l'opérateur d'affectation (s'il est présent) et l'opérateur postfixé qui agit après l'opérateur d'affectation. Il faut donc redéfinir 2 méthodes de nom **`operator++`**. *A priori* ces deux méthodes doivent modifier l'objet courant, donc ne sont pas **`const`** et ne prennent pas d'argument : elles ont donc même signature. Comment faire alors pour les différencier ? Il faut connaître la règle suivante :

— la version préfixée a pour signature **`operator++()`** et elle renvoie une référence,
— la version postfixée a pour signature **`operator++(int)`** (où l'argument entier n'est pas utilisé par la méthode) et elle renvoie un objet.

*Opérateurs d'incrément pour la classe* `Point`

```
class Point {
   public:
      Point(int a, int b) : x(a), y(b) {};
      Point & operator++() {      // incrément préfixé
         ++x; ++y;
         return *this;
      }
      Point operator++(int) {     // incrément postfixé
         Point copie(*this);
         ++x; ++y;
         return copie;
      }
   private:
      int x, y;
};
```

Vérifier que le comportement de ces méthodes est bien celui qui est cohérent avec l'opérateur `++` pour des types primitifs. Que donnent les appels suivants ?

*Appels des opérateurs d'incrément pour la classe* `Point`

```
Point P(1, 2), Q;
Q = ++P;
Q = P++;
```

### Opérateurs de conversion

Il est possible de définir des opérateurs pour convertir un objet en une variable d'un type de base : `int`, `double`, `bool`,... Pour cela on doit définir la méthode de prototype `operator type()` qui ne possède pas de type de retour ! Par exemple, pour convertir un `p_adic` vers un `int` on écrira

*Opérateur de conversion vers un* `int`

```
class p_adic {
   public:
      // constructeurs
      // destructeur
      // opérateur d'affectation
      operator int() { return valeur; }
   private:
      unsigned valeur, p, r;
      unsigned * coeff;
};
```

### Autres opérateurs

Il est aussi possible de définir d'autres opérateurs par des méthodes, par exemple les opérateurs référence/déréférencement `&` et `*`, les opérateurs arithmétiques et les opérateurs de comparaison. Pour les opérateurs binaires on utilisera en général une surcharge par une fonction globale et non une méthode. Voici un exemple illustrant la raison :

*Surcharge de l'opérateur + par une méthode*

```
class Point {
   public:
      Point(int a = 0, int b = 0) : x(a), y(b) {};
      Point operator+(Point const & P) const {
         return Point(x + P.x, y + P.y)
      };
   private:
      int x, y;
};
```

Les appels possibles sont les suivants

*Exemple d'appel de l'opérateur +*

```
Point P(2, 3);
Point Q = 2;
Point R1 = P + Q;  // P.operator+(Q)
Point R2 = P + 2;  // P.operator+(Point(2,0)) conversion implicite
Point R3 = 2 + P;  // ??? erreur à la compilation
```

### 6.4.2 Par une fonction globale

On peut aussi surcharger un opérateur par une fonction globale dont le nom est **operator** suivi du (ou des) symbole(s) de l'opérateur, par exemple **operator==**, **operator+**, **operator<<**, etc. Cette fonction prend 1 seul argument si l'opérateur est unaire et 2 arguments si l'opérateur est binaire. La règle d'appel est alors la suivante :

— si l'opérateur est unaire : l'appel de l'opérateur appliqué à un objet `P` correspond à l'appel de la fonction avec pour argument l'objet `P`.

— si l'opérateur est binaire : l'appel de l'opérateur appliqué entre 2 objets `P` et `Q` correspond à l'appel de la fonction avec pour arguments les objets `P` et `Q`.

La plupart du temps, on déclarera le fonction globale comme amie de la classe (et ce à l'intérieur de la classe, cf 6.3).

#### Opérateurs arithmétiques

Les opérateurs arithmétiques doivent renvoyer un nouvel objet résultat de l'opération. Voici l'exemple de la surcharge de l'opérateur `+` pour la classe `Point`

*Surcharge de l'opérateur + par une fonction globale*

```
class Point {
   public:
      Point(int a = 0, int b = 0) : x(a), y(b) {};
      friend Point operator+(Point const & P, Point const & Q);
   private:
      int x, y;
};
// définition de la fonction globale en dehors
Point operator+(Point const & P, Point const & Q) {
   return Point(P.x + Q.x, P.y + Q.y);
};
```

Les appels possibles sont les suivants

*Exemple d'appel de l'opérateur +*

```
Point P(2, 3);
Point Q = 2;
Point R1 = P + Q;  // operator+(P, Q)
Point R2 = P + 2;  // operator+(P, Point(2,0)) conversion implicite
Point R3 = 2 + P;  // operator+(Point(2,0), P) OK
```

### Opérateurs de comparaison

Un opérateur de comparaison `<,<=,==,>,>=` doit renvoyer un **`bool`**. Il faut donc que la fonction globale utilisée pour coder un opérateur de comparaison renvoie un **`bool`**. Par exemple dans le cas de la class `p_adic` on peut écrire

*Surcharge des opérateurs < et == par des fonctions globales*

```
class p_adic {
   public:
      // constructeurs, destructeur, opérateur d'affectation
      friend bool operator<(p_adic const & n, p_adic const & m);
      friend bool operator==(p_adic const & n, p_adic const & m);
   private:
      unsigned valeur, p, r;
      unsigned * coeff;
};
bool operator<(p_adic const & n, p_adic const & m) {
   return n.valeur < m.valeur;
};
bool operator==(p_adic const & n, p_adic const & m) {
   return n.valeur == m.valeur;
};
```

Noter qu'il est nécessaire de définir au moins les opérateurs `<` et `==` pour effectuer toutes les comparaisons possibles entre 2 objets d'une classe. Bien souvent on définit les opérateurs restants `<=`, `>`, `>=`, `!=` de façon générique à partir des opérateurs `<` et `==`. On pourra consulter la section 6.5.3.

### Opérateurs de flux

Pour surcharger les opérateurs de flux `<<` (injection) et `>>` (extraction) on doit utiliser des fonctions globales. En effet, l'objet à gauche de ces opérateurs est toujours un objet d'une classe de flux : `ostream` (ou une classe dérivée...) pour `<<` ou `istream` (ou une classe dérivée) pour `>>`. Les prototypes pour une classe `nom_classe` sont les suivants

*Prototypes des fonctions surchargeant les opérateurs de flux*

```
std::ostream & operator<<(std::ostream & o, nom_classe const & X);
std::istream & operator>>(std::istream & i, nom_classe & X);
```

## 6.5 Fonctions génériques, une introduction

Il s'agit juste ici d'illustrer l'intérêt de surcharger les opérateurs classiques pour vos propres classes. On reviendra en détail sur les aspects de la programmation générique dans le chapitre suivant.

Comme on vient de surcharger l'opérateur `<` pour des objets des classes `p_adic` et `Point`, il suffit d'utiliser cet opérateur pour comparer 2 objets de même classe. Ainsi on pourrait définir plusieurs fonctions `min` qui auraient toutes le même code mais des prototypes différents

*Multiples fonctions `min` dont le code est identique*

```
double min(double x, double y) {
   return x < y ? x : y;
}
complex<double> min(complex<double> const & x,
                    complex<double> const & y) {
   return x < y ? x : y;
}
Point min(Point const & x, Point const & y) {
   return x < y ? x : y;
}
p_adic min(p_adic const & x, p_adic const & y) {
   return x < y ? x : y;
}
```

La fonction `min` aura le même code pour 2 objets d'une classe dont l'opérateur `<` existe (et dont le constructeur de clonage est correct).

### 6.5.1 Déclaration et définition d'une fonction générique

Un mécanisme en `C++` permet d'écrire un modèle de fonction (ou encore une fonction générique) qui remplace la définition de multiples fonctions au code identique (à des types près). La syntaxe est la suivante : on fait précéder la déclaration **et** la définition de la fonction générique par le mot-clé **`template`** suivit d'une liste entre chevrons `< >` de déclarations composées de types génériques. Par exemple

*Fonction `min` générique*

```
template <typename T>  // T est un type générique
T min(T x, T y) {
   // T peut être utilisé dans la fonction générique \src{min<T>}
   return x < y ? x : y;
}
```

La fonction `min` définie ici est dite générique et sera notée `min<T>` dans la suite de ce cours. Le nom du type générique `T` est bien sûr arbitraire. Il peut aussi y avoir plusieurs types génériques :

*Fonction `f` générique*

```
template <typename T, typename S>
double f(T x, S y) {
   // dans cette fonction générique on peut déclarer des
   // variables locales de type T et de type S
}
```

Dans le code précédent, on a défini la fonction générique `f<T, S>`.

### 6.5.2 Appel d'une fonction générique

A partir d'une fonction générique par exemple `min<T>` il va exister plusieurs *instances* de cette fonction, c'est à dire des fonctions générées à partir de cette fonction générique (c'est pour cela qu'on parle aussi de modèle de fonctions). Par exemple, on peut instancier la fonction `min<T>` avec le type **`double`** et obtenir la fonction `min<double>`, de même avec le type `Point` pour obtenir la fonction `min<Point>`, etc. Le compilateur détermine automatiquement à la compilation les instances à créer en

fonctions des appels : c'est un mécanisme statique qui a lieu à la compilation donc le code produit en écrivant 1 seule fonction générique `min<T>` est aussi rapide que le code écrit avec 4 fonctions `min` distinctes.

Voici un exemple d'appel de la fonction générique instanciée avec différents types :

*Appels explicites d'instances de la fonction `min<T>`*

```
min<double>(2.3, 4);
min< complex<double> >(x, y);
min<Point>(1, Point(1,1));
min<p_adic>(n, m);
```

S'il n'y a pas de confusion possible lors d'un appel, le nom de la fonction `instanciée` peut être implicite. Cela permet une écriture aisée du code. De plus ce mécanisme est très utile lorsqu'on ne connaît pas explicitement le type des arguments (on verra des exemples plus tard).

*Appels implicites d'instances de la fonction `min<T>`*

```
min(2.3, 4);
min(x, y);
min(1, Point(1,1));
min(n, m);
```

### 6.5.3 Opérateurs génériques

Il est possible d'appliquer ces fonctions génériques à des fonctions utilisées pour surcharger des opérateurs. Par exemple si les opérateurs de comparaison `<` et `==` sont définis pour une classe, il peut être utile de définir les opérateurs `<=`, `>`, `>=` et `!=` par des fonctions génériques.

*Opérateurs de comparaison surchargés par des fonctions génériques*

```
template <typename T>
bool operator>(T const & a, T const & b) { return b < a; }
template <typename T>
bool operator<=(T const & a, T const & b) { return !(b < a); }
template <typename T>
bool operator>=(T const & a, T const & b) { return !(a < b); }
template <typename T>
bool operator!=(T const & a, T const & b) { return !(a == b); }
```

On peut aussi déclarer ces fonctions globales **`inline`** pour une meilleure optimisation du code.

## 6.6 Liens entre classes

### 6.6.1 Inclusion

Une classe `A` peut avoir pour champ un objet d'une autre classe `B`. La classe `B` a sa propre définition et est donc autonome par rapport à `A` mais il est nécessaire de la déclarer avant la définition de la classe `A`. On pourra dire qu'il y a une relation d'inclusion entre `B` et `A`.

L'initialisation du champ de type `B` doit se faire en appelant le constructeur de la classe `B` dans la liste d'initialisation du constructeur de la classe `A`. Prenons l'exemple suivant : on veut coder une classe `Pixel` contenant un champ `Point` et un champ `couleur` (de type **`int`**). On dira que `Point` est incluse dans `Pixel`. Voici le code de cette classe `Pixel`

*Objet de classe `Point` membre de la classe `Pixel`*

```
class Pixel {
   public:
      Pixel(int a = 0, int b = 0, int c = 0) : P(a,b), couleur(c) {}
   private:
      Point P;
      int couleur;
};
```

Un `Pixel` contient donc un `Point` mais ces deux classes sont étrangères l'une pour l'autre : un objet de la classe `Pixel` ne peut pas accéder aux champs privés de la classe `Point` et réciproquement. De plus une méthode de la classe `Point` ne peut pas s'appeler à partir d'un objet de la classe `Pixel`. Même si la notion de point et de pixel est proche, un objet de la classe `Pixel` est très différent d'un objet de la classe `Point`.

### 6.6.2 Amitié

Dans la section 6.3 on a vu qu'il est possible de déclarer une fonction globale comme amie d'une classe `A` ce qui lui permet d'accéder à ses champs privés. Cette déclaration se fait dans la classe `A`.

De la même façon, il est possible de déclarer une classe `B` comme amie de la classe `A` et cela donne le droit à toutes les méthodes de la classe `B` d'accéder aux champs privés de la classe `A`. Cette déclaration doit se faire dans la classe `A` en déclarant : **`friend class B`**.

Par exemple, dans l'exemple précédent on peut déclarer `Pixel` comme amie de `Point` (et `Point` incluse dans `Pixel`).

*Déclaration de la classe `Pixel` comme amie de `Point`*

```
class Point {
   friend class Pixel;  // déclaration d'amitié
   public:
      Point(int x, int y) : x(x), y(y) {}
      double norme() const { return sqrt(x*x + y*y); }
   private:
      int x, y;
};

class Pixel {
   public:
      Pixel(int a = 0, int b = 0, int c = 0) : P(a,b), couleur(c) {}
      double norme() const { return sqrt(P.x*P.x + P.y*P.y); }
   private:
      Point P;
      int couleur;
};
```

Attention, la relation d'amitié n'est ni symétrique, ni transitive !

### 6.6.3 Héritage

L'héritage est un mécanisme bien plus complexe et plus puissant que ceux qui précèdent. Dans tout ce qui suit, on abordera uniquement l'*héritage simple et public*. La relation d'héritage est transitive, non symétrique, non réflexive et non cyclique. L'héritage constitue un lien très fort entre 2 classes. De façon équivalente, on dira que

— `A` est la classe mère de `B`,

— `B` hérite de `A`,
— `B` est une classe dérivée de `A`, ou classe fille de `A`.
— `B` est un `A`
— `B` est une spécialisation de `A`, `A` est une généralisation de `B`

Un objet de la classe fille `B` hérite des champs (membres et méthodes) de la classe mère `A`. Intuitivement, un objet d'une classe fille est similaire à un objet de la classe mère avec des modifications telles que : des champs en plus, des nouvelles méthodes, des méthodes redéfinies (avec un code différent) ou surchargées (avec une signature différente), etc.

Le constructeur de la classe fille appelle en premier le constructeur de la classe mère : si l'appel n'est pas explicite dans le code, alors c'est le constructeur par défaut de la classe qui est appelé s'il existe (si il n'existe pas il y a une erreur à la compilation). De même, le destructeur de la classe fille termine par l'appel (toujours implicite) du destructeur de la classe mère.

La syntaxe pour l'héritage public sera la suivante :

*Syntaxe pour l'héritage public*

```
class A {
   // définition de la classe A
};
class B : public A {  // B hérite de A publiquement
   public :
      B(type1 var1, ..., typeN varN) : A(type1 var1, ...) {
         // définition du constructeur de B
      }
};
```

Pour reprendre l'exemple précédent, il est naturel de définir `Pixel` comme une classe dérivée de `Point`. On écrira donc le code suivant

*La classe `Pixel` dérive de `Point`*

```
class Point {
   public:
      Point(int a=0, int b=0) : x(a), y(b) {}
      double dist0() const { return sqrt(x*x + y*y); }
      void affiche() const { std::cout << x << " , " << y; }
   private:
      int x, y;
};

class Pixel : public Point {
   public:
      Pixel(int a=0, int b=0, int c=0) : Point(a,b), couleur(c) {}
   private:
      int couleur;
};
```

Noter d'abord que le constructeur `Pixel` appelle explicitement le constructeur de la classe mère avec les arguments `a` et `b` : `Point(a,b)` puis initialise le champ supplémentaire `couleur`.

La classe `Point` contient ici 2 champs privés `x` et `y` et 2 méthodes publiques (en plus du constructeur) `dist0()` et `affiche()`. Ces champs et méthodes seront présents dans tout objet `Point` ainsi que dans tout objet d'une classe fille comme `Pixel`. C'est le premier avantage de l'héritage : la réutilisation du code déjà écrit.

Dans l'exemple précédent on peut donc faire les appels suivants :

*Appels des méthodes de la classe mère*

```
Point P(2, 3);
Pixel Q(2, 3, 245);
Q.affiche();
if (P.dist0() == Q.dist0())
   cout << "P et Q sont à même distance de 0" << endl;
```

### Accessibilité

Les méthodes `dist0()` et `affiche()` sont déclarées **`public`** dans la classe mère et seront donc **`public`** dans la classe fille. Concernant les champs `x` et `y` c'est un peu plus complexe. En effet, ils sont déclarés **`private`** ce qui les rend uniquement accessible par des méthodes de la classe `Point` ou à des amies (fonctions ou méthodes d'une classe amie). *Les champs `x` et `y` sont donc présents mais inaccessibles dans la classe `Pixel`.*

Pour assouplir cette notion de **`private`** lors de l'héritage, il existe la notion de **`protected`**. Un membre (champ ou méthode) déclaré **`protected`** sera inaccessible dans tout le reste du programme et accessible dans les méthodes et amies (comme **`private`**) mais restera **`protected`** dans une classe fille.

L'héritage public respecte donc les règles suivantes :

| Classe mère | | Classe fille |
|---|---|---|
| **`public`** | reste | **`public`** |
| **`protected`** | reste | **`protected`** |
| **`private`** | devient | *inaccessible* |

En résumé si l'on veut utiliser explicitement des membres (champs ou méthodes) de la classe mère dans des méthodes d'une classe fille il faut les déclarer **`protected`**.

### Visibilité

Tous les membres **`public`** et **`protected`** d'une classe mère sont *a priori* visibles dans une classe fille. Cependant des membres de la classe fille peuvent avoir le même nom que certains de la classe mère et ainsi masquer l'accès aux membres de la classe mère. Par exemple on peut définir une méthode **`void`** `affiche()` **`const`** dans la classe fille `Pixel` qui masquera la méthode **`void`** `affiche()` **`const`** de la classe mère. La méthode de la classe mère sera donc masquée mais elle restera accessible par son nom complet qui est `Point::affiche()`. De même le nom complet du champ `x` est `Point::x`.

Il y a essentiellement 2 façons de redéfinir la méthode `affiche()` pour la classe `Pixel`. Soit on utilise les champs `x` et `y` et le même code d'affichage que la méthode `affiche()` de la classe `Point` auquel on ajoute l'affichage de la `couleur`. Soit on appelle explicitement `Point::affiche()` (cette solution est plus élégante et plus souple).

*Deux façons presque équivalentes de redéfinir la méthode `affiche()`*

```
void Pixel::affiche() const {
   // il faut que x et y soient protected
   std::cout << x << " , " << y << " - " << couleur;
};
// ou
void Pixel::affiche() const {
   Point::affiche(); // on appelle la méthode de la classe mère
   std::cout << " - " << couleur;
};
```

Dans la classe `Pixel` il existe donc maintenant 2 fonctions `affiche()`. L'une accessible uniquement par son nom complet `Point::affiche()` est celle de la classe mère et l'autre accessible par son nom `affiche()` ou par son nom complet `Pixel::affiche()` est celle de la classe fille.

*Appel des 2 méthodes `affiche()` coéxistentes de la classe `Pixel`*

```
Pixel Q(2, 3, 254);
Q.affiche();         // affichage de 2 , 3 - 254
Q.Point::affiche(); // affichage de 2 , 3
```

### Liens entre objets et adresses d'objets de même famille

Il existe un lien très fort un objet d'une classe fille et un objet d'une classe mère. On a les 2 propriétés suivantes :

- un objet de la classe fille est accepté partout où un objet de la classe mère est attendu : **l'objet est alors tronqué**,
- l'adresse d'un objet de la classe fille est compatible avec l'adressse d'un objet de la classe mère : **l'objet à cette adresse n'est pas modifiée**.

La deuxième propriété est fondamentale et donne toute sa puissance à l'héritage. Il faut bien comprendre ces 2 propriétés.

Voici un exemple qui illustre ces propriétés

*Illustration des liens entre objets et adresses d'objets de classes mère/fille*

```
Pixel Q(2, 3, 254);
Point P = Q;        // OK 1ère propriété
Point * pt = &Q;    // OK 2ème propriété
Point & r = Q;      // OK 2ème propriété
```

Quels sont les types des variables `Q`, `P`, `pt` et `r` ? On a vu que le type était donné par le code source à la déclaration des variables, donc le type de `Q` est `Pixel`, le type de `P` est `Point`, le type de `pt` est `Point *` (pointeur sur `Point`) et le type de `r` est `Point &` (référence sur `Point`). D'autre part, l'adresse de `Q` (un `Pixel`) est compatible avec celle d'un `Point` et cette adresse initialise le pointeur `pt` et la référence `r`. Donc `pt` pointe vers un `Pixel` et `r` est une référence sur un `Pixel` : l'adresse au moment de l'execution du programme est celle d'un `Pixel` et non d'un `Point`.

**Définition 17** *Le type statique d'une variable est donné par le code source à la déclaration de la variable : il est déterminé à la compilation.*

*Le type dynamique d'une variable adresse est donné par le type effectif du contenu présent à cette adresse à l'execution.*

Le type dynamique de `pt` est donc un pointeur sur un `Pixel` et le type dynamique de `r` est une référence sur un `Pixel` (en l'occurence `Q`).

**Attention :** le mécanisme d'appel d'une méthode se base sur le type statique et non sur le type dynamique. Ainsi les appels `pt->affiche()` et `r.affiche()` correspondent aux appels `pt->Point::affiche()` et `r.Point::affiche()` car `pt` et `r` ont pour type statique une adresse sur un `Point`.

### Polymorphisme dynamique : méthodes virtuelles

Pour passer outre cette restriction d'appel d'une méthode se basant sur le type statique, on doit déclarer la méthode **`virtual`** (mot-clé devant la déclaration du prototype). Le mécanisme d'appel d'une méthode virtuelle se base sur le type dynamique.

Par exemple pour définir la méthode `affiche()` comme virtuelle dans la classe `Point` et toutes ses classes filles, il suffit d'ajouter le mot-clé **`virtual`** devant le prototype.

*La classe `Pixel` dérive de `Point` avec redéfinition de la méthode virtuelle `affiche()`*

```
class Point {
   public:
      Point(int a=0, int b=0) : x(a), y(b) {}
      double dist0() const { return sqrt(x*x + y*y); }
      virtual void affiche() const { std::cout << x << " , " << y; }
   protected:
      int x, y;
};
class Pixel : public Point {
   public:
      Pixel(int a=0, int b=0, int c=0) : Point(a,b), couleur(c) {}
      void affiche() const; // code inchangé !
   private:
      int couleur;
};
```

Avec cette méthode virtuelle, les appels `pt->affiche()` et `r.affiche()` correnspondent aux apples `pt->Pixel::affiche()` et `r.Pixel::affiche()` car `pt` et `r` ont pour type dynamique une adresse sur un `Pixel`. Ce mécanisme est appelé *polymorphisme dynamique* et permet d'écrire une fonction qui s'applique à un objet d'une classe mère comme à un objet d'une classe fille. Prenons l'exemple suivant où une fonction `f` peut s'appliquer aussi bien à un `Point` qu'à un `Pixel` :

*Illustration du polymorphisme dynamique*

```
void f(Point const & P) {
   // ...
   P.affiche(); // appel d'affiche en fonction du type dynamique de P
};
// appels possibles plus loin dans le code:
Pixel Q(2, 3, 254);
Point P(4, 5);
f(Q); // OK il y aura affichage de 2 3 254
f(P); // OK il y aura affichage de 4 5
```

Noter qu'une redéfinition d'une méthode virtuelle n'est pas obligatoire, qu'un constructeur ne peut pas être virtuel et qu'un destructeur peut être virtuel.

### Classe abstraite

Pour terminer sur l'héritage simple et public on introduit la notion de classe abstraite utilisée principalement pour l'organisation du code et l'utilisation du polymorphisme dynamique : écrire une fonction qui s'applique à tout objet de toute classe fille de cette classe abstraite.

**Définition 18** *Une méthode est dite virtuelle pure si elle est virtuelle (mot-clé **`virtual`** devant le prototype) et non définie c'est à dire si le prototype est suivi de `= 0;`.*

*Une classe abstraite est une classe ayant au moins une méthode virtuelle pure.*

Une propriété importante est qu'il ne peut exister d'objets d'une classe abstraite (d'où son nom...).

Par exemple on peut définir une classe `var_alea` qui contient une méthode **`operator()`** virtuelle pure. Cette classe abstraite ne peut pas avoir d'objets mais on peut définir des classes filles de `var_alea`, par exemple `uniforme` et `gaussienne` qui possèdent une méthode **`operator()`**. Dans ce cas, ces classes `uniforme` et `gaussienne` ne sont plus des classes abstraites.

*Classe abstraite et classes filles*

```
struct var_alea {
   virtual double operator()() const = 0; // virtuelle pure
};
struct uniforme : public var_alea {
   double operator()() const {
      // code qui renvoie une réalisation uniforme sur [0,1]...
   }
};
struct gaussienne : public var_alea {
   double operator()() const {
      // code qui renvoie une réalisation gaussienne N(0;1)...
   }
};
```

On peut alors écrire une fonction `moyenne_empirique` qui effectue la moyenne empirique de `n` réalisations d'une variable aléatoire. Le code de cette fonction sera par exemple

*Classe abstraite et classes filles*

```
double moyenne_empirique(int n, var_alea const & X) {
   double result;
   for (int i = 0; i < n; ++i)
      result += X(); // appel de operator() d'après le type dynamique
   return result;
};
```

Les appels suivants sont alors possibles :

*Appels possible de la fonction polymorphe `moyenne_empirique()`*

```
moyenne_empirique(1e6, uniforme());
gaussienne G;
moyenne_empirique(1e6, G);
```

# Chapitre 7

# Programmation générique et Librairie standard STL

La programmation générique en C++ repose sur le mécanisme appelé **template**. Ce mécanisme permet de définir des modèles de fonctions ou de classes (ou structures...) qui seront instanciés à la compilation. Ces modèles peuvent prendre plusieurs paramètres de 2 sortes : type (désigné avec le mot-clé **typename**) ou constante entière (désigné avec **int**). L'instanciation est effectuée par le compilateur et consiste en la création d'une version effective de la fonction ou de la classe (avec un ou plusieurs types donnés et/ou une ou plusieurs constantes données).

Pour définir un modèle il faut ajouter le mot-clé **template** suivi d'une liste de paramètres génériques devant la déclaration et la définition de la fonction ou de la classe.

*Quelques syntaxes possibles*

```
template <typename T>
// T type générique

template <int N>
// N est une constante entière

template <typename F, typename G, int N>
// F et G sont des types génériques et N une constante entière

template <int N = 10, typename F = double>
// N est une constante entière qui prend la valeur par défaut 10
// F est un type générique qui prend la valeur par défaut double
```

## 7.1 Fonctions génériques

On a déjà vu les fonctions génériques dans la partie 6.5.

### 7.1.1 Spécifier une instanciation

On reprend l'exemple de la fonction générique `min<T>` qui renvoie le minimum entre 2 variables ou objets pour lesquels existent l'opérateur `<` (et le constructeur de copie).

*Fonction générique `min<T>`*

```
template <typename T> T min(T a, T b) {
   return a < b ? a : b.
};
```

On a vu qu'il était alors possible d'appeler cette fonction générique instanciée avec les types **double**, **int**, p_adic, complex<**double**>. Par contre, l'appel suivant min("abc", "def") n'est pas possible car les constantes "abc" et "def" sont de type **char** * et il n'existe pas d'opérateur < pour comparer ces constantes. Une solution simple dans ce cas précis serait d'appeler min(std::string("abc"), std::string("dev")) mais cela nécessite la création de 2 objets std::string.

Plus généralement, on est dans un cas où l'on souhaite définir une instanciation particulière de la fonction min<T> avec le type **char** *. Cela est possible en définissant la fonction min<**char***> précédée de **template** <> pour indiquer qu'il s'agit d'une instance d'une fonction générique. On écrira donc le code :

*Définition de l'instance min<**char***>*

```
template <> char * min<char*>(char const * a, char const * b) {
   return (strcmp(a, b) < 0) ? a : b;
};
```

Il est aussi possible de surcharger la fonction min sans préciser qu'il s'agit d'une instance de la fonction générique min<T>. Cette solution est cependant moins souhaitable. Dans ce cas on écrira simplement

*Surcharge de la fonction min*

```
char * min(char const * a, char const * b) {
   return (strcmp(a, b) < 0) ? a : b;
};
```

### 7.1.2 Un exemple de template récursif

Le paramètre du modèle peut aussi être une constante entière. Voici un exemple d'utilisation d'une telle fonction générique paramétrée par un entier N.

*Une fonction générique et récursive !*

```
template <int N>
inline int fact(){
   return N*fact<N-1>(); // appel explicite de l'instance fact<N-1>()
}

template <> // définition de l'instance fact<0>()
inline int fact<0>(){
   return 1;
}
```

Un appel de fact<5>() correspond à la création par le compilateur des 5 fonctions fact<k> (pour k de 1 à 5). Comme les fonctions sont **inline**, l'évaluation de fact<5>() correspond exactement à l'évaluation de 5*4*3*2*1.

De nombreuses bibliothèques utilisent ce mécanisme, aussi appelé métaprogrammation, pour créer un code optimisé à la compilation : le temps de compilation est bien plus long mais le code produit peut-être très rapide. Ces aspects dépassent le cadre de ce cours.

## 7.2 Classes génériques

Le mécanisme de **template** est le même pour une classe générique. La seule différence avec les fonctions génériques est qu'il faut toujours préciser (explicitement) l'instanciation utilisée de la classe

générique. Prenons l'exemple d'une classe `vecteur` générique qui dépend de 2 paramètres : un type `T` et un entier `d`. On peut la déclarer de la façon suivante :

*Une classe `vecteur` générique*

```
template <typename T, int d>
class vecteur {
   public:
      T operator[](int i) const { return data[i]; }
      T & operator[](int i) { return data[i]; }
      vecteur & operator+=(T k);
      vecteur & operator+=(vecteur const & v);
      double norme() const;
   private:
      T data[d];
};
```

A l'intérieur de la définition de la classe, on peut utiliser le nom `vecteur` à la place de `vecteur<T,d>`. Par exemple les opérateurs d'incrément-affectation `+=` renvoie un `vecteur` : cette écriture est possible uniquement dans la définition de la classe (et dans une méthode).

### Méthode d'une classe générique

De la même façon à l'intérieur de la définition d'une méthode et dans la signature, on peut utilise la notation `vecteur` à la place de la notation complète `vecteur<T,d>`. Voici des définitions possibles pour les 3 méthodes de cette classe générique. Notez que la ligne **`template`** doit précéder chaque définition.

*Définition des méthodes de la classe `vecteur` générique*

```
template <typename T, int d>
vecteur<T,d> vecteur<T,d>::operator+=(T k) {
   for (T * pt = data; pt != data+d; ++pt)
      *pt += k;
   return *this;
}

template <typename T, int d>
vecteur<T,d> vecteur<T,d>::operator+=(vecteur const & v) {
   T * pt = data, * v_pt = v.data;
   while (pt != data+d)
      *pt++ += *v_pt++;
   return *this;
}

template <typename T, int d>
double vecteur<T,d>::norme() const {
   double result;
   for (T const * pt = data; pt != data+d; ++pt)
      result += (*pt) * (*pt);
   return result;
}
```

### Fonction globale et classe générique

Dans le cas d'une fonction globale, il faut toujours préciser le nom complet de la classe générique `vecteur<T,d>`.

*Définition d'une fonction générique*

```
template<typename T, int d>
vecteur<T,d> operator+(vecteur<T,d> const & v,
                       vecteur<T,d> const & w) {
   vecteur<T,d> result;
   for (int i = 0; i < d; ++i)
      result[i] = v[i] + w[i];
   return result;
}
```

### Fonction amie d'une classe générique

Pour déclarer une fonction amie d'une classe générique c'est un petit peu plus subtil. En fait on peut s'y prendre de 2 façons différentes : déclarer comme amie une fonction générique (toute une classe de fonction) ou bien une instance particulière. La première façon est plus facile syntaxiquement et c'est celle que l'on utilise dans les TP de ce cours.

#### Fonction générique amie d'une classe générique

*Fonction générique amie d'une classe générique*

```
template <typename T>
class A {
   public:
      //...
      template <typename S>   // ici S doit être différent de T
      friend double f(A<S> const & x);
};
```

Il s'agit de la même syntaxe qu'habituellement : on fait précéder le prototype de la fonction générique par le mot-clé **`friend`**. Mais celui-ci est placé après le **`template <typename S>`** qui indique que la fonction amie (ici `f`) est générique. Notez que `S` est un nouveau paramètre différent du paramètre `T` (qui existe déjà). Dans ce cas, la classe `A<T>` est amie avec toutes les instances possibles de la fonction générique `f`, toutes les instances `f<S>`.

#### Instance d'une fonction générique amie d'une classe générique

Pour déclarer l'instance `f<T>` comme amie de la classe générique `A<T>` il faut d'abord déclarer la fonction générique `f` (lignes 4 à 8) avec un paramètre quelconque : `S` ou `T` ou `CEQUEVOUSVOULEZ`. Mais la fonction générique `f<S>` prend pour argument une référence constante `x` sur un objet de la classe générique `A<S>` : il faut donc que la classe soit déclarer avant (lignes 1 et 2).

Enfin une fois ces déclarations faîtes, on peut définir la classe générique `A` (lignes 10 à 15) en indiaquant que la fonction `f<T>` (l'instance de la fonction générique avec le paramètre `T`) est amie de la classe `A<T>`.

*Instance d'une fonction générique amie d'une classe générique*

```
template <typename T>
class A; // uniquement la déclaration de la classe A générique

template <typename S>  // définition de la fonction f générique
                       // ici S peut être T
double f(A<S> const & x) {
   // ...
};

template <typename T>
class A {
   public:
      //...
      friend double f<T>(A const & x);  // seule f<T> est amie
};
```

## 7.3 STL

Fin du cours pas encore rédigée mais au programme de l'examen final

### 7.3.1 Conteneurs séquentiels : `vector<T>` et `list<T>`

### 7.3.2 Itérateurs

Définition d'un itérateur, utilisation, `iterator`, `const_iterator`...

### 7.3.3 Algorithmes

Cf. fiche donnée en cours

## Aide-mémoire pour la Standard Template Library

### Itérateurs

Définition des itérateurs :

| | |
|---|---|
| *InpIt* | Input iterator (lecture) |
| *OutIt* | Output iterator (écriture) |
| *ForIt* | Forward iterator (lecture/écriture + déplacement avant) |
| *BiDirIt* | Bidirectional iterator (lecture/écriture + dépl. avant/arrière) |
| *RandIt* | Random access iterator (lecture/écriture + dépl. libre) |

Deux fonctions globales agissant sur les itérateurs :

| | | | |
|---|---|---|---|
| `int` | `distance` | (*InpIt* first, *InpIt* last) | renvoie le nombre d'élts entre 2 itérateurs |
| `void` | `advance` | (*InpIt* it, `int` n) | avance l'itérateur it de n élts (ou recule si n< 0) |

### Objets fonctionnels

Notations utilisés pour les objets fonctionnels :

| | | |
|---|---|---|
| *Gen* | `generator<R>` | |
| *Fun* | `unary_function<T, R>` | negate, identity |
| *BFun* | `binary_function<T1, T2, R>` | plus, minus, multiplies, divides, modulus |
| *Pred* | `unary_function<T, bool>` | logical_and, logical_or, logical_not |
| *BPred* | `binary_function<T, T, bool>` | equal_to, not_equal_to |
| | | less, less_equal |
| | | greater, greater_equal |

### Conteneurs :

| | | | |
|---|---|---|---|
| Conteneurs : `X` ∈ {`vector`, `list`, `deque`, `set`, `multiset`, `map`, `multimap`} | | | |
| | `X` | () | constructeur par défaut |
| | `X` | (`X` `const` &x) | constructeur de clônage |
| `X &` | `operator=` | (`X` `const` &x) | opérateur d'affectation |
| `cst_iter` | `begin` | () `const` | ↩ itérateur de lecture placé au début du conteneur |
| `cst_iter` | `end` | () `const` | ↩ itérateur placé après le dernier élt |
| `cst_iter` | `rbegin` | () `const` | ↩ itérateur inverse placé sur le dernier élt |
| `cst_iter` | `rend` | () `const` | ↩ itérateur inverse placé avant le premier élt |
| `unsigned` | `size` | () `const` | ↩ nombre d'éléments du conteneur |
| `unsigned` | `max_size` | () `const` | ↩ nombre maximal d'élts |
| `bool` | `empty` | () `const` | ↩ `true` si aucun élément |
| `void` | `clear` | () | détruit le contenu, taille mise à 0 |
| `void` | `swap` | (`X` &x) `const` | échange le contenu avec celui de x |
| Conteneurs ordonnés : `CO` ∈ {`vector`, `list`, `deque`} | | | |
| | `CO` | (`unsigned` n, `T` &val) | constructeur de remplissage, taille n avec val |
| | `CO` | (*InpIt* first, *InpIt* last) | constructeur à partir d'une plage d'itérateurs |
| `T &` | `front` | () `const` | ↩ premier élt du conteneur |
| `T &` | `back` | () `const` | ↩ dernier élt du conteneur |
| `void` | `push_back` | (`const` `T` &val) | ajoute un élt à la fin du conteneur |
| `void` | `pop_back` | () | retire le dernier élt du conteneur |
| `void` | `resize` | (`unsigned` s, `T` c) | redimensionne le conteneur, quitte à ajouter des élts c |
| `CO::iter` | `insert` | (`CO::iter` before, `const` `T` &val) | ↩ iterateur sur l'élt inséré après before de valeur val |
| `void` | `insert` | (`CO::iter` before, `unsigned` n, `const` `T` &val) | insertion de n élts après before |
| | | (`CO::iter` before, *InpIt* first, *InpIt* last) | insertion de [first, last[ après before |
| `CO::iter` | `erase` | (`CO::cst_iter` position) | ↩ itérateur placé après l'élt supprimé position |
| `CO::iter` | `erase` | (`CO::cst_iter` first, `CO::cst_iter` last) | ↩ itérateur placé après les élts supprimés [first,last[ |
| Méthodes spécifiques à la classe `vector` | | | |
| `T` | `operator[]` | (`unsigned` i) `const` | ↩ le ième élément (lecture) |
| `T &` | `operator[]` | (`unsigned` i) | ↩ référence sur le ième élément (écriture) |
| Méthodes spécifiques à la classe `deque` | | | |
| `T` | `operator[]` | (`unsigned` i) `const` | ↩ le ième élément (lecture) |
| `T &` | `operator[]` | (`unsigned` i) | ↩ référence sur le ième élément (écriture) |
| `void` | `push_front` | (`const` `T` &val) | ajoute un élt au début du conteneur |
| `void` | `pop_front` | () | retire le premier élt du conteneur |
| Méthodes spécifiques à la classe `list` | | | |
| `void` | `push_front` | (`const` `T` &val) | ajoute un élt en début de liste |
| `void` | `pop_front` | () | retire le premier élt de la liste |
| `void` | `reverse` | () | inverse l'ordre des élts de la liste |
| `void` | `remove` | (`const` `T` &val) | efface tous les élts de la liste égaux à val |
| `void` | `remove_if` | (*Pred* pred) | efface tous les élts vérifiants pred (résultat de pred est `true`) |
| `void` | `sort` | (*Cmp* cmp) | trie les élts du plus petit au plus grand (opérateur < ou foncteur `Cmp`) |
| `void` | `unique` | (*BPred* pred) | remplace tous les blocs d'élts égaux (ou `pred`) par un seul d'entre eux |
| `void` | `merge` | (`list` &x, *Cmp* cmp) | fusionne la liste courante (triée) avec `x` et reste triée |
| `void` | `splice` | (`list::iter` pos, `list` &x, `iter` itx) | déplace les élts de la liste `x` dans la liste courante |
| | | (`list::iter` pos, `list` &x, `iter` firstx, `iter` lastx) | |

## Algorithmes :

| | | | |
|---|---|---|---|
| | | Fonctions définies dans <algorithm> | |
| Fonctions qui ne modifient pas l'organisation des données | | | |
| *Fun* | `for_each` | (*InpIt* first, *InpIt* last, *Fun* f) | Applique fonction f. ↩ f |
| *InpIt* | `find` | (*InpIt* first, *InpIt* last, **const** T &val) | ↩ premier élt égal à val, ou last |
| *InpIt* | `find_if` | (*InpIt* first, *InpIt* last, *Pred* pred) | ↩ premier élt vérifiant pred, ou last |
| *ForIt*1 | `find_end` | (*ForIt*1 first1, *ForIt*1 last1, *ForIt*2 first2, *ForIt*2 last2, *Pred* pred) | ↩ dernier élt où apparaît le motif [first2 ;last2[ dans [first1 ;last1[ (similaire à `search` en partant de la fin) |
| *ForIt*1 | `find_first_of` | (*ForIt*1 first1, *ForIt*1 last1, *ForIt*2 first2, *ForIt*2 last2, *Pred* pred) | ↩ premier élt de [first1 ;last1[ présent dans [first2 ;last2[ |
| *ForIt* | `adjacend_find` | (*ForIt* first, *ForIt* last, *Pred* pred) | ↩ premier élt égal à son successeur |
| (int) | `count` | (*ForIt* first, *ForIt* last, **const** T &val) | ↩ nb d'élts égaux à val |
| (int) | `count_if` | (*ForIt* first, *ForIt* last, *Pred* pred) | ↩ nb d'élts vérifiants pred |
| **bool** | `equal` | (*InpIt*1 first1, *InpIt*1 last1, *InpIt*2 first2, *Pred* pred) | Test whether the elements in two ranges are equal |
| *ForIt*1 | `search` | (*ForIt*1 first1, *ForIt*1 last1, *ForIt* first2, *ForIt* last2, *BPred* pred) | ↩ premier élt où apparaît le motif [first2 ;last2[ dans [first1 ;last1[ |
| *ForIt* | `search_n` | (*ForIt* first1, *ForIt* last1, *Size* count, **const** T &val, *BPred*pred) | ↩ premier élt où apparaît count valeur de val |
| Fonctions qui modifient l'organisation des données | | | |
| *OutIt* | `copy` | (*InpIt* first, *InpIt* last, *OutIt* result) | Copy range of elements |
| *BiDirIt*2 | `copy_backward` | (*BiDirIt*1 first, *BiDirIt*1 last, *BiDirIt*2 result) | Copy range of elements backwards |
| **void** | `swap` | (T & a, T & b) | Exchange values of two objects |
| *ForIt*2 | `swap_ranges` | (*ForIt*1 first, *ForIt*1 last, *ForIt*2 result) | Exchange values of two ranges |
| **void** | `iter_swap` | (*ForIt*1 a, *ForIt*2 b) | Exchange values of objects pointed by two iterators |
| *OutIt* | `transform` | (*InpIt* first, *InpIt* last, *OutIt* result, *UFun* op)<br>(*InpIt*1 first1, *InpIt*1 last1, *InpIt*2 first2, *OutIt* result, *BFun* op) | Apply function to range |
| **void** | `replace` | (*ForIt* first, *ForIt* last, **const** T &old, **const** T &new) | Replace value in range |
| **void** | `replace_if` | (*ForIt* first, *ForIt* last, *Pred* pred, **const** T &new) | Replace values in range |
| *OutIt* | `replace_copy` | (*InpIt* first, *InpIt* last, *OutIt* result, **const** T &old, **const** T &new) | Copy range replacing value |
| *OutIt* | `replace_copy_if` | (*InpIt* first, *InpIt* last, *OutIt* result, *Pred* pred, **const** T &new) | Copy range replacing value |
| **void** | `fill` | (*ForIt* first, *ForIt* last, **const** T &val) | Fill range with value |
| **void** | `fill_n` | (*OutIt* first, *Size* n, **const** T&val) | Fill sequence with value |
| **void** | `generate` | (*ForIt* first, *ForIt* last, *Gen* gen) | Generate values for range with function |
| **void** | `generate_n` | (*OutIt* first, *Size* n, *Gen* gen) | Generate values for sequence with function |
| *ForIt* | `remove` | (*ForIt* first, *ForIt* last, **const** T &val) | Remove value from range |
| *ForIt* | `remove_if` | (*ForIt* first, *ForIt* last, *Pred* pred) | Remove elements from range |
| *OutIt* | `remove_copy` | (*ForIt* first, *ForIt* last, *OutIt* result, **const** T &val) | Copy range removing value |
| *OutIt* | `remove_copy_if` | (*ForIt* first, *ForIt* last, *OutIt* result, *Pred* pred) | Copy range removing values |
| *ForIt* | `unique` | (*ForIt* first, *ForIt* last) | Remove consecutive duplicates in range |
| *OutIt* | `unique_copy` | (*ForIt* first, *ForIt* last, *OutIt* result, *BPred* pred) | Copy range removing duplicates |
| *BiDirIt* | `reverse` | (*BiDirIt* first, *BiDirIt* last) | Reverse range |
| *OutIt* | `reverse_copy` | (*BiDirIt* first, *BiDirIt* last, *OutIt* result) | Copy range reversed |
| **void** | `rotate` | (*ForIt* first, *ForIt* middle, *ForIt* last) | Rotate elements in range |
| *OutIt* | `rotate_copy` | (*ForIt* first, *ForIt* middle, *ForIt* last, *OutIt* result) | Copy rotated range |
| **void** | `random_shuffle` | (*RandIt* first, *RandIt* last, *Rand* &rand) | Rearrangle elements in range randomly |
| *BiDirIt* | `partition` | (*BiDirIt* first, *BiDirIt* last, *Pred* pred) | Partition range in two |
| *BiDirIt* | `stable_partition` | (*BiDirIt* first, *BiDirIt* last, *Pred* pred) | Divide range in two groups - stable ordering |
| Min/max and sorting | | | |
| **const** T & | `min` | (**const** T &a, **const** T &b, *Cmp* cmp) | ↩ le minimum entre a et b |
| **const** T & | `max` | (**const** T &a, **const** T &b, *Cmp* cmp) | ↩ le maximum entre a et b |
| *ForIt* | `min_element` | (*ForIt* first, *ForIt* last, *Cmp* cmp) | ↩ plus petit élt de [first, last[ |
| *ForIt* | `max_element` | (*ForIt* first, *ForIt* last, *Cmp* cmp) | ↩ plus grand élt de [first, last[ |
| **void** | `sort` | (*RandIt* first, *RandIt* last, *Cmp* cmp) | trie les élts de [first, last[ |
| **void** | `stable_sort` | (*RandIt* first, *RandIt* last, *Cmp* cmp) | ordre relatif des élts de même valeur est préservé |
| **void** | `partial_sort` | (*RandIt* first, *RandIt* middle, *RandIt* last, *Cmp* cmp) | trie les (middle-first) plus petits élts |
| *RandIt* | `partial_sort_copy` | (*ForIt* first, *ForIt* last, *RandIt* resfirst, *RandIt* reslast, *Cmp* cmp) | copie avant le tri |
| **void** | `nth_element` | (*RandIt* first, *RandIt* nth, *RandIt* last, *Cmp* cmp) | nth est un pivot (élts à gauche <, et à droite >, non triés) |
| Binary search and merge (operating on sorted ranges) | | | |
| *ForIt* | `lower_bound` | (*ForIt* first, *ForIt* last, **const** T &val, *Cmp* cmp) | ↩ premier élt plus grand que val |
| *ForIt* | `upper_bound` | (*ForIt* first, *ForIt* last, **const** T &val, *Cmp* cmp) | ↩ premier élt plus petit que val |
| **bool** | `binary_search` | (*ForIt* first, *ForIt* last, **const** T &val, *Cmp* cmp) | ↩ **true** si val ∈ [first, last[ |
| *OutIt* | `merge` | (*ForIt*1 first1, *ForIt*1 last1, *ForIt*2 first2, *ForIt*2 last2, *OutIt* result, *Cmp* cmp) | |
| **void** | `inplace_merge` | (*BiDirIt* first, *BiDirIt* middle, *BiDirIt* last, *Cmp* cmp) | Merge consecutive sorted ranges |
| | | Fonctions définies dans <numerical> | |
| T | `accumulate` | (*InpIt* first, *InpIt* last, T init, *BFun*op) | ↩ somme (ou op) de init et de tous les élts de [fist, last[ |
| T | `inner_prod` | (*InpIt*1 first1, *InpIt*1 last1, *InpIt*2 first2, T init, *BFun* op1, *BFun* op2 ) | ↩ produit scalaire entre [first1, last1[ et [first2, last2[ |
| *OutIt* | `partial_sum` | (*InpIt* first, *InpIt* last, *OutIt*res, *BFun* op) | *result = *first et *(result+1) = op(*result, *(first+1)) ... |
| *OutIt* | `adjacent_difference` | (*InpIt* first, *InpIt* last, *OutIt*res, *BFun* op) | ↩ *result = *first et différence (ou op) de 2 élts adjacents |

*Version provisoire de ces notes de cours. Elles seront mises à jour au cours du semestre.*
*Ne pas imprimer à chaque nouvelle version !*
*Merci de m'indiquer les erreurs et les points obscurs...*

# Annexe A

# Préprocesseur C, compilation séparée

Le préprocesseur C applique un traitement aux fichiers sources avant compilation. Il peut-être utilisé avec différents langages de programmation car il est indépendant de la syntaxe propre du langage. Les lignes qui commencent par un **#** sont adressées au préprocesseur. Ces lignes sont appelées des directives de compilation.

## A.1 Directives de compilation

### A.1.1 Inclusion `#include`

La directive **`#include`** permet d'inclure un fichier d'en-tête contenant des déclarations de fonctions, de types, d'autres directives de compilation, etc. Il y a 2 syntaxe possible

- **`#define`** `<ficher.h>` : où le `fichier.h` entre chevrons sera recherché dans les répertoires systèmes : c'est donc un fichier d'en-tête du C ou d'une librairie installée sur l'OS,
- **`#define`** `"ficher.h"` : où le `fichier.h` entre guillemets doubles sera recherché à partir du répertoire local contenant le fichier source `.c` : c'est un fichier que vous avez créé ou celui d'une librairie installée en local.

Comme les déclarations (de fonctions, de types, etc.) doivent être uniques, un fichier `.h` doit être inclus *une seule fois*.

### A.1.2 Déclaration `#define`

La directive **`#define`** permet de remplacer des symboles par d'autres avant la compilation. Elle est utilisée pour déclarer des constantes globales ou des macros. Une macro peut-être vu comme un remplacement avec argument(s). Les macros ne sont absolument pas des fonctions, ne respectent pas les mécanismes d'appels de fonction, ne sont pas vérifiées à la compilation, et peuvent être dangereuses.

Les syntaxes possibles sont les suivantes :

*Syntaxe de déclaration `#define`*

```
#define CONSTANTE expr
#define MACRO1(x) (expr(x))          // macro à 1 paramètre
#define MACRO2(x,y) (expr(x,y))      // macro à 2 paramètre
```

Pour comprendre la dangerosité d'une macro, on considère le programme suivant :

*Exemple de déclaration `#define` : fichier `test.c`*

```
#define PI 3.14159
#define PP(x) ((x) > 0 ? (x) : 0)

double partie_positive(double x) {
   return x > 0 ? x : 0;
}

int main(void) {
   double a1 = 2*PI, a2 = a1;
   double b1 = PP(a1++);
   double b2 = partie_positive(a2++);
   return 0;
}
```

Après le passage au préprocesseur, on obtient le code suivant :

*Après le préprocesseur : `gcc -E -P test.c`*

```
double partie_positive(double x) {
 return x > 0 ? x : 0;
}

int main(void) {
 double a1 = 2*3.14159, a2 = a1;
 double b1 = ((a1++) > 0 ? (a1++) : 0);
 double b2 = partie_positive(a2++);
 return 0;
}
```

Comparer les valeurs de `b1` et de `b2` à la fin du programme.

### A.1.3 Directives conditionnelles

## A.2 Compilation séparée

## A.3 Makefile

# Annexe B

# Extraits de la librairie standard du C

La librairie standard est composée de nombreuses fonctions préprogrammmées qui permettent une manipulation plus simple de la mémoire, des entrées/sorties (ligne de commande, fichiers), des chaînes de caractères, des erreurs, etc. De plus, il existe aussi quelques fonctions mathématiques.

Les fichiers d'en-têtes qui contiennent la déclaration de ces fonctions sont les suivants :

— `stdio.h`
— `stdlib.h`
— `math.h`
— `float.h`
— `limits.h`
— `string.h`
— `time.h`
— d'autres : `assert.h`, `ctype.h`, `errno.h`, `locale.h`, `setjmp.h`, `signal.h`, `stdarg.h`, `stddef.h`

Ces fonctions sont très bien documentées dans des *pages de manuel* qui devraient être accessible dans tout environnement de programmation. C'est la documentation principale pour le programmeur (bien avant d'aller chercher des informations sur internet) qu'il faut savoir utiliser. Pour chercher la page de manuel correspondant à une fonction de la librairie standard en `C`, par exemple `printf`, on tapera en ligne de commande

```
$ man 3 printf
```

Si ces pages ne sont pas installées sur votre système vous pouvez utiliser ces 2 moteurs de recherche :

— http://linux.die.net/man/ (documentation Linux)
— http://www.freebsd.org/cgi/man.cgi? (documentation FreeBSD)

Certaines pages sont mieux écrites dans la documentation FreeBSD.

## B.1 `stdio.h`

A faire.

## NOM

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf − Formatage des sorties

## SYNOPSIS

```
#include <stdio.h>

int printf(const char * format, ...);
int fprintf(FILE *stream, const char * format, ...);
int sprintf(char *str, const char * format, ...);
int snprintf(char *str, size_t size, const char * format, ...);

#include <stdarg.h>

int vprintf(const char * format, va_list ap);
int vfprintf(FILE *stream, const char * format, va_list ap);
int vsprintf(char *str, const char * format, va_list ap);
int vsnprintf(char *str, size_t size, const char * format, va_list ap);
```

Exigences de macros de test de fonctionnalités pour la glibc (voir **feature_test_macros**(7)) :

**snprintf**(), **vsnprintf**() : _BSD_SOURCE || _XOPEN_SOURCE >= 500 || _ISOC99_SOURCE ; ou *cc −std=c99*

## DESCRIPTION

Les fonctions de la famille **printf**() produisent des sorties en accord avec le *format* décrit plus bas. Les fonctions **printf**() et **vprintf**() écrivent leur sortie sur *stdout*, le flux de sortie standard. **fprintf**() et **vfprintf**() écrivent sur le flux *stream* indiqué. **sprintf**(), **snprintf**(), **vsprintf**() et **vsnprintf**() écrivent leurs sorties dans la chaîne de caractères *str*.

Les fonctions **snprintf**() et **vsnprintf**() écrivent au plus *size* octets (octet nul (« \0 ») inclus) dans *str*.

Les fonctions **vprintf**(), **vfprintf**(), **vsprintf**(), **vsnprintf**() sont équivalentes aux fonctions **printf**(), **fprintf**(), **sprintf**(), **snprintf**(), respectivement, mais elles emploient un tableau *va_list* à la place d'un nombre variable d'arguments. Ces fonctions n'appellent pas la macro *va_end*. Aussi, la valeur de *va_arg* est-elle indéfinie après l'appel. Voir **stdarg**(3).

Ces huit fonctions créent leurs sorties sous le contrôle d'une chaîne de *format* qui indique les conversions à apporter aux arguments suivants (ou accessibles à travers les arguments de taille variable de **stdarg**(3)).

## VALEUR RENVOYÉE

Si elles réussissent, ces fonctions renvoient le nombre de caractères imprimés, sans compter l'octet nul « \0 » final dans les chaînes. Les fonctions **snprintf**() et **vsnprintf**() n'écrivent pas plus de *size* octets (y compris le « \0 » final). Si la sortie a été tronquée à cause de la limite, la valeur de retour est le nombre de caractères (sans le « \0 » final) qui auraient été écrits dans la chaîne s'il y avait eu suffisamment de place. Ainsi, une valeur de retour *size* ou plus signifie que la sortie a été tronquée. (Voir aussi la section NOTES plus bas). Si une erreur de sortie s'est produite, une valeur négative est renvoyée.

### Chaîne de format

Le format de conversion est indiqué par une chaîne de caractères, commençant et se terminant dans son état de décalage initial. La chaîne de format est composée d'indicateurs : les caractères ordinaires (différents de **%**), qui sont copiés sans modification sur la sortie, et les spécifications de conversion, qui sont mises en correspondance avec les arguments suivants. Les spécifications de conversion sont introduites par le caractère **%**, et se terminent par un *indicateur de conversion*. Entre eux peuvent se trouver (dans l'ordre), zéro ou plusieurs *attributs*, une valeur optionnelle de *largeur minimal de champ*, une valeur optionnelle de *précision*, et un éventuel *modificateur de longueur*.

Les arguments doivent correspondre correctement (après les promotions de types) avec les indicateurs de conversion. Par défaut les arguments sont pris dans l'ordre indiqué, où chaque « * » et chaque indicateur de conversion réclament un nouvel argument (et où l'insuffisance en arguments est une erreur). On peut aussi

préciser explicitement quel argument prendre, en écrivant, à chaque conversion, « %m$ » au lieu de « % », et « *m$ » au lieu de « * ». L'entier décimal m indique la position dans la liste d'arguments, l'indexation commençant à 1. Ainsi,

```
printf("%*d", width, num);
```

et

```
printf("%2$*1$d", width, num);
```

sont équivalents. La seconde notation permet de répéter plusieurs fois le même argument. Le standard C99 n'autorise pas le style utilisant « $ », qui provient des Spécifications Single Unix. Si le style avec « $ » est utilisé, il faut l'employer pour toutes conversions prenant un argument, et pour tous les arguments de largeur et de précision, mais on peut le mélanger avec des formats « %% » qui ne consomment pas d'arguments. Il ne doit pas y avoir de sauts dans les numéros des arguments spécifiés avec « $ ». Par exemple, si les arguments 1 et 3 sont spécifiés, l'argument 2 doit aussi être mentionné quelque part dans la chaîne de format.

Pour certaines conversions numériques, un caractère de séparation décimale (le point par défaut) est utilisé, ainsi qu'un caractère de regroupement par milliers. Les véritables caractères dépendent de la localisation **LC_NUMERIC**. La localisation POSIX utilise « . » comme séparateur décimal, et n'a pas de caractère de regroupement. Ainsi,

```
printf("%'.2f", 1234567.89);
```

s'affichera comme «\[u00A0]1234567.89 » dans la localisation POSIX, «\[u00A0]1 234 567,89 » en localisation fr_FR, et « 1.234.567,89 » en localisation da_DK.

**Caractère d'attribut**

Le caractère % peut être éventuellement suivi par un ou plusieurs attributs suivants :

**#** indique que la valeur doit être convertie en une autre forme. Pour la conversion **o** le premier caractère de la chaîne de sortie vaudra zéro (en ajoutant un préfixe 0 si ce n'est pas déjà un zéro). Pour les conversions **x** et **X** une valeur non nulle reçoit le préfixe « 0x » (ou « 0X » pour l'indicateur **X**). Pour les conversions **a**, **A**, **e**, **E**, **f**, **F**, **g**, et **G** le résultat contiendra toujours un point décimal même si aucun chiffre ne le suit (normalement, un point décimal n'est présent avec ces conversions que si des décimales le suivent). Pour les conversions **g** et **G** les zéros en tête ne sont pas éliminés, contrairement au comportement habituel. Pour les autres conversions, cet attribut n'a pas d'effet.

**0** indique le remplissage avec des zéros. Pour les conversions **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, et **G**, la valeur est complétée à gauche avec des zéros plutôt qu'avec des espaces. Si les attributs **0** et **−** apparaissent ensemble, l'attribut **0** est ignoré. Si une précision est fournie avec une conversion numérique (**d**, **i**, **o**, **u**, **x**, et **X**), l'attribut **0** est ignoré. Pour les autres conversions, le comportement est indéfini.

**−** indique que la valeur doit être justifiée sur la limite gauche du champ (par défaut elle l'est à droite). Sauf pour la conversion **n**, les valeurs sont complétées à droite par des espaces, plutôt qu'à gauche par des zéros ou des blancs. Un attribut **−** surcharge un attribut **0** si les deux sont fournis.

**' '** (une espace) indique qu'une espace doit être laissée avant un nombre positif (ou une chaîne vide) produit par une conversion signée

**+** Un signe (+ ou −) doit toujours être imprimé avant un nombre produit par une conversion signée. Par défaut, un signe n'est utilisé que pour des valeurs négatives. Un attribut **+** surcharge un attribut « espace » si les deux sont fournis.

Les cinq caractères d'attributs ci-dessus sont définis dans le standard C, les spécifications SUSv2 en ajoute un :

' Pour les conversions décimales (**i**, **d**, **u**, **f**, **F**, **g**, **G**) indique que les chiffres d'un argument numérique doivent être groupés par milliers en fonction de la localisation. Remarquez que de nombreuses versions de **gcc**(1) n'acceptent pas cet attribut et déclencheront un avertissement (warning). SUSv2 n'inclue pas *%'F*.

La glibc 2.2 ajoute un caractère d'attribut supplémentaire.

**I** Pour les conversions décimales (**i**, **d**, **u**) la sortie emploie les chiffres alternatifs de la localisation s'il y en a. Par exemple, depuis la glibc 2.2.3, cela donnera des chiffres arabes pour la localisation perse (« fa_IR »).

**Largeur de champ**

Un nombre optionnel ne commençant pas par un zéro, peut indiquer une largeur minimale de champ. Si la valeur convertie occupe moins de caractères que cette largeur, elle sera complétée par des espaces à gauche (ou à droite si l'attribut d'alignement à gauche a été fourni). À la place de la chaîne représentant le nombre décimal, on peut écrire « * » ou « *m$ » (*m* étant entier) pour indiquer que la largeur du champ est fournie dans l'argument suivant, ou dans le *m*-ième argument, respectivement. L'argument fournissant la largeur doit être de type *int*. Une largeur négative est considéré comme l'attribut « − » vu plus haut suivi d'une largeur positive. En aucun cas une largeur trop petite ne provoque la troncature du champ. Si le résultat de la conversion est plus grand que la largeur indiquée, le champ est élargi pour contenir le résultat.

**Précision**

Une précision éventuelle, sous la forme d'un point (« . ») suivi par un nombre. À la place de la chaîne représentant le nombre décimal, on peut écrire « * » ou « *m$ » (m étant entier) pour indiquer que la précision est fournie dans l'argument suivant, ou dans le m-ième argument, respectivement. L'argument fournissant la précision doit être de type *int*. Si la précision ne contient que le caractère « . », ou une valeur négative, elle est considérée comme nulle. Cette précision indique un nombre minimum de chiffres à faire apparaître lors des conversions **d**, **i**, **o**, **u**, **x**, et **X**, le nombre de décimales à faire apparaître pour les conversions **a**, **A**, **e**, **E**, **f** et **F**, le nombre maximum de chiffres significatifs pour **g** et **G**, et le nombre maximum de caractères à imprimer depuis une chaîne pour les conversions **s** et **S**.

**Modificateur de longueur**

Ici, une conversion entière correspond à **d**, **i**, **o**, **u**, **x** ou **X**.

**hh** La conversion entière suivante correspond à un *signed char* ou *unsigned char*, ou la conversion **n** suivante correspond à un argument pointeur sur un *signed char*.

**h** La conversion entière suivante correspond à un *short int* ou *unsigned short int*, ou la conversion **n** suivante correspond à un argument pointeur sur un *short int*.

**l** (elle) La conversion entière suivante correspond à un *long int* ou *unsigned long int*, ou la conversion **n** suivante correspond à un pointeur sur un *long int*, ou la conversion **c** suivante correspond à un argument *wint_t*, ou encore la conversion **s** suivante correspond à un pointeur sur un *wchar_t*.

**ll** (elle-elle) La conversion entière suivante correspond à un *long long int*, ou *unsigned long long int*, ou la conversion **n** suivante correspond à un pointeur sur un *long long int*.

**L** La conversion **a**, **A**, **e**, **E**, **f**, **F**, **g**, ou **G** suivante correspond à un argument *long double*. (C99 autorise %LF mais pas SUSv2).

**q** (« quad » BSD 4.4 et Linux sous libc5 seulement, ne pas utiliser) Il s'agit d'un synonyme pour **ll**.

**j** La conversion entière suivante correspond à un argument *intmax_t* ou *uintmax_t*.

**z** La conversion entière suivante correspond à un argument *size_t* ou *ssize_t*. (La bibliothèque libc5 de Linux proposait l'argument **Z** pour cela, ne pas utiliser).

**t** La conversion entière suivante correspond à un argument *ptrdiff_t*.

Les spécifications SUSv2 ne mentionnent que les modificateurs de longueur **h** (dans **hd**, **hi**, **ho**, **hx**, **hX**, **hn**),

**l** (dans **ld**, **li**, **lo**, **lx**, **lX**, **ln**, **lc**, **ls**) et **L** (dans **Le**, **LE**, **Lf**, **Lg**, **LG**).

**Indicateur de conversion**

Un caractère indique le type de conversion à apporter. Les indicateurs de conversion, et leurs significations sont :

**d**, **i** L'argument *int* est converti en un chiffre décimal signé. La précision, si elle est mentionnée, correspond au nombre minimal de chiffres qui doivent apparaître. Si la conversion fournit moins de chiffres, le résultat est rempli à gauche avec des zéros. Par défaut la précision vaut 1. Lorsque 0 est converti avec une précision valant 0, la sortie est vide.

**o**, **u**, **x**, **X**
L'argument *unsigned int* est converti en un chiffre octal non signé (**o**), un chiffre décimal non signé (**u**), un chiffre hexadécimal non signé (**x** et **X**). Les lettres **abcdef** sont utilisées pour les conversions avec **x**, les lettres **ABCDEF** sont utilisées pour les conversions avec **X**. La précision, si elle est indiquée, donne un nombre minimal de chiffres à faire apparaître. Si la valeur convertie nécessite moins de chiffres, elle est complétée à gauche avec des zéros. La précision par défaut vaut 1. Lorsque 0 est converti avec une précision valant 0, la sortie est vide.

**e**, **E** L'argument réel, de type *double*, est arrondi et présenté avec la notation scientifique [−]c**.**ccc**e**±cc dans lequel se trouve un chiffre avant le point, puis un nombre de décimales égal à la précision demandée. Si la précision n'est pas indiquée, l'affichage contiendra 6 décimales. Si la précision vaut zéro, il n'y a pas de point décimal. Une conversion **E** utilise la lettre **E** (plutôt que **e**) pour introduire l'exposant. Celui-ci contient toujours au moins deux chiffres. Si la valeur affichée est nulle, son exposant est 00.

**f**, **F** L'argument réel, de type *double*, est arrondi, et présenté avec la notation classique [-]ccc**.**ccc, où le nombre de décimales est égal à la précision réclamée. Si la précision n'est pas indiquée, l'affichage se fera avec 6 décimales. Si la précision vaut zéro, aucun point n'est affiché. Lorsque le point est affiché, il y a toujours au moins un chiffre devant.

SUSv2 ne mentionne pas **F** et dit qu'il existe une chaîne de caractères représentant l'infini ou NaN. Le standard C99 précise « [−]inf » ou « [−]infinity » pour les infinis, et une chaîne commençant par « nan » pour NaN dans le cas d'une conversion **f**, et les chaînes « [−]INF » « [−]INFINITY » « NAN* » pour une conversion **F**.

**g**, **G** L'argument réel, de type *double*, est converti en style **f** ou **e** (ou **F** ou **E** pour la conversion **G**) La précision indique le nombre de décimales significatives. Si la précision est absente, une valeur par défaut de 6 est utilisée. Si la précision vaut 0, elle est considérée comme valant 1. La notation scientifique **e** est utilisée si l'exposant est inférieur à −4 ou supérieur ou égal à la précision demandée. Les zéros en fin de partie décimale sont supprimés. Un point décimal n'est affiché que s'il est suivi d'au moins un chiffre.

**a**, **A** (C99 mais pas SUSv2). Pour la conversion **a**, l'argument de type *double* est transformé en notation hexadécimale (avec les lettres abcdef) dans le style [−]**0x**h**.**hhhh**p**±d; Pour la conversion **A**, le préfixe **0X**, les lettres ABCDEF et le séparateur d'exposant **P** sont utilisés. Il y a un chiffre hexadécimal avant la virgule, et le nombre de chiffres ensuite est égal à la précision. La précision par défaut suffit pour une représentation exacte de la valeur, si une représentation exacte est possible en base 2. Sinon, elle est suffisamment grande pour distinguer les valeurs de type *double*. Le chiffre avant le point décimal n'est pas spécifié pour les nombres non normalisés, et il est non nul pour les nombres normalisés.

**c** S'il n'y a pas de modificateur **l**, l'argument entier, de type *int*, est converti en un *unsigned char*, et le caractère correspondant est affiché. Si un modificateur **l** est présent, l'argument de type *wint_t* (caractère large) est converti en séquence multi-octet par un appel à **wcrtomb**(3), avec un état de conversion débutant dans l'état initial. La chaîne multi-octet résultante est écrite.

**s** S'il n'y a pas de modificateur **l**, l'argument de type *const char* * est supposé être un pointeur sur un tableau de caractères (pointeur sur une chaîne). Les caractères du tableau sont écrits jusqu'à l'octet nul « \0 » final, non compris. Si une précision est indiquée, seul ce nombre de caractères sont écrits. Si une précision est fournie, il n'y a pas besoin d'octet nul. Si la précision n'est pas

donnée, ou si elle est supérieure à la longueur de la chaîne, l'octet nul final est nécessaire.

Si un modificateur **l** est présent, l'argument de type *const wchar_t* * est supposé être un pointeur sur un tableau de caractères larges. Les caractères larges du tableau sont convertis en une séquence de caractères multi-octets (chacun par un appel de **wcrtomb**(3), avec un état de conversion dans l'état initial avant le premier caractère large), ceci jusqu'au caractère large nul final compris. Les caractères multi-octets résultants sont écris jusqu'à l'octet nul final (non compris). Si une précision est fournie, il n'y a pas plus d'octets écrits que la précision indiquée, mais aucun caractère multi-octet n'est écrit partiellement. Remarquez que la précision concerne le nombre *d'octets* écrits, et non pas le nombre de *caractères larges* ou de *positions d'écrans*. La chaîne doit contenir un caractère large nul final, sauf si une précision est indiquée, suffisamment petite pour que le nombre d'octets écrits la remplisse avant la fin de la chaîne.

**C** (dans SUSv2 mais pas dans C99) Synonyme de **lc**. Ne pas utiliser.

**S** (dans SUSv2 mais pas dans C99) Synonyme de **ls**. Ne pas utiliser.

**p** L'argument pointeur, du type *void* * est affiché en hexadécimal, comme avec **%#x** ou **%#lx**.

**n** Le nombre de caractères déjà écrits est stocké dans l'entier indiqué par l'argument pointeur de type *int* *. Aucun argument n'est converti.

**m** (Extension glibc.) Afficher la sortie de *strerror(errno)*. Aucun argument n'est nécessaire.

**%** Un caractère « % » est écrit. Il n'y a pas de conversion. L'indicateur complet est « %% ».

## CONFORMITÉ

Les fonctions **fprintf**(), **printf**(), **sprintf**(), **vprintf**(), **vfprintf**(), et **vsprintf**() sont conformes à C89 et C99. Les fonctions **snprintf**() et **vsnprintf**() sont conformes à C99.

En ce qui concerne la valeur de retour de **snprintf**(), SUSv2 et C99 sont en contradiction : lorsque **snprintf**() est appelée avec un argument *size=0* SUSv2 précise une valeur de retour indéterminée, inférieure à 1, alors que C99 autorise *str* à être NULL dans ce cas, et réclame en valeur de retour (comme toujours) le nombre de caractères qui auraient été écrits si la chaîne de sortie avait été assez grande.

La bibliothèque libc4 de Linux connaissait les 5 attributs standard du C. Elle connaissait les modificateurs de longueur h, l, L et les conversions cdeEfFgGinopsuxX, où F était synonyme de f. De plus, elle acceptait D, O, U comme synonymes de ld, lo et lu. (Ce qui causa de sérieux bogues par la suite lorsque le support de %D disparut). Il n'y avait pas de séparateur décimal dépendant de la localisation, pas de séparateur des milliers, pas de NaN ou d'infinis, et pas de %m$ ni *m$.

La bibliothèque libc5 de Linux connaissait les 5 attributs standard C, l'attribut « ' », la localisation, %m$ et *m$. Elle connaissait les modificateurs de longueur h, l, L, Z, q, mais acceptait L et q pour les *long double* et les *long long int* (ce qui est un bogue). Elle ne reconnaissait plus FDOU, mais ajoutait le caractère de conversion **m**, qui affiche *strerror(errno)*.

La bibliothèque glibc 2.0 ajouta les caractères de conversion C et S.

La bibliothèque glibc 2.1 ajouta les modificateurs de longueur hh, t, z, et les caractères de conversion a, A.

La bibliothèque glibc 2.2. ajouta le caractère de conversion F avec la sémantique C99, et le caractère d'attribut I.

## NOTES

L'implémentation des fonctions **snprintf**() et **vsnprintf**() de la glibc se conforme au standard C99, et se comporte comme décrit plus haut depuis la glibc 2.1. Jusqu'à la glibc 2.0.6, elles renvoyaient −1 si la sortie avait été tronquée.

## BOGUES

Comme **sprintf**() et **vsprintf**() ne font pas de suppositions sur la longueur des chaînes, le programme appelant doit s'assurer de ne pas déborder l'espace d'adressage. C'est souvent difficile. Notez que la longueur des chaînes peut varier avec la localisation et être difficilement prévisible. Il faut alors utiliser **snprintf**() ou **vsnprintf**() à la place (ou encore **asprintf**(3) et **vasprintf**(3)).

La libc4.[45] de Linux n'avait pas **snprintf**(), mais proposait une bibliothèque libbsd qui contenait un **snprintf**() équivalent à **sprintf**(), c'est-à-dire qui ignorait l'argument *size*. Ainsi, l'utilisation de **snprintf**() avec les anciennes libc4 pouvait conduire à de sérieux problèmes de sécurité.

Un code tel que **printf(** *foo***);** indique souvent un bogue, car *foo* peut contenir un caractère « % ». Si *foo* vient d'une saisie non sécurisée, il peut contenir « %n », ce qui autorise **printf**() à écrire dans la mémoire, et crée une faille de sécurité.

**EXEMPLE**

Pour afficher pi avec cinq décimales :

```
#include <math.h>
#include <stdio.h>
fprintf (stdout, "pi = %.5f\n", 4 * atan (1.0));
```

Pour afficher une date et une heure sous la forme « Sunday, July 3, 23:15 », ou *jour_semaine* et *mois* sont des pointeurs sur des chaînes :

```
#include <stdio.h>
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
        jour_semaine, mois, jour, heure, minute);
```

De nombreux pays utilisent un format de date différent, comme jour-mois-année. Une version internationale doit donc être capable d'afficher les arguments dans l'ordre indiqué par le format :

```
#include <stdio.h>
fprintf(stdout, format,
        jour_semaine, mois, day, hour, min);
```

où le *format* dépend de la localisation et peut permuter les arguments. Avec la valeur :

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

on peut obtenir « Dimanche, 3 juillet, 23:15 ».

Pour allouer une chaîne de taille suffisante et écrire dedans (code correct aussi bien pour glibc 2.0 que glibc 2.1) :

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
char *
make_message(const char *fmt, ...)
{
   /* Supposons que 100 octets suffisent. */
   int n, size = 100;
   char *p, *np;
   va_list ap;

   if ((p = malloc(size)) == NULL)
      return NULL;
   while (1) {
      /* Essayons avec l'espace alloué. */
      va_start(ap, fmt);
      n = vsnprintf(p, size, fmt, ap);
      va_end(ap);
      /* Si ça marche, renvoyer la chaîne. */
      if (n > -1 && n < size)
```

```
            return p;
        /* Sinon réessayer avec plus de place */
        if (n > -1)    /* glibc 2.1 */
            size = n+1; /* ce qu'il fallait */
        else           /* glibc 2.0 */
            size *= 2;  /* deux fois plus */
        if ((np = realloc(p, size)) == NULL) {
            free(p);
            return NULL;
        } else {
            p = np;
        }
    }
}
```

## VOIR AUSSI

**printf**(1), **asprintf**(3), **dprintf**(3), **scanf**(3), **setlocale**(3), **wcrtomb**(3), **wprintf**(3), **locale**(5)

## TRADUCTION

Ce document est une traduction réalisée par Christophe Blaess <http://www.blaess.fr/christophe/> le 10 novembre 1996 et révisée le 17 juillet 2008.

L'équipe de traduction a fait le maximum pour réaliser une adaptation française de qualité. La version anglaise la plus à jour de ce document est toujours consultable via la commande : « **LANG=C man 3 printf** ». N'hésitez pas à signaler à l'auteur ou au traducteur, selon le cas, toute erreur dans cette page de manuel.

## B.2 math.h

Voici une liste des fonctions de la librairie mathématique. Sur certaines machines, il faut utiliser l'option `-lm` de `gcc` lors de l'édition de lien pour inclure le code de ces fonctions.

— `acos`, `acosh`, `asin`, `asinh`, `atan`, `atanh`, `atan2`
— `cos`, `cosh`, `sin`, `sinh`, `tan`, `tanh`
— `erf`, `erfc` (fonction d'erreur)
— `pow`, `exp`, `exp2`, `expm1`
— `log`, `log10`, `log2`, `log1p`
— `tgamma`, `lgamma` (fonction Gamma)
— `j0`, `j1`, `jn`, `y0`, `y1`, `yn` (fonctions de Bessel)
— `sqrt`, `cbrt`, `hypot`, `fma`
— `fabs`, `fdim`, `fmax`, `fmin`, `copysign`, `signbit`
— `ceil`, `floor`, `round`, `trunc`, `fmod`, `modf`, ...
— `isfinite`, `isinf`, `isnan`, `isnormal`, `fpclassify`
— `isgreater`, `isless`, `isgreaterequal`, `islessequal`, ...

Les 3 pages suivantes sont les pages de manuel de `floor`, `erf` et `tgamma`.

## NOM

floor, floorf, floorl − Le plus grand entier inférieur ou égal à x

## SYNOPSIS

**#include <math.h>**

**double floor(double** *x***);**
**float floorf(float** *x***);**
**long double floorl(long double** *x***);**

Utilisez *−lm* à l'édition de liens pour lier avec la bibliothèque mathématique.

## DESCRIPTION

Ces fonctions renvoient la valeur *x* arrondie par défaut à l'entier le plus proche.

## VALEUR RENVOYÉE

La valeur entière arrondie. Si *x* est entier ou infini, la valeur renvoyée est *x* lui-même.

## ERREURS

Seules les erreurs **EDOM** et **ERANGE** peuvent se produire. Si *x* est NaN, NaN est renvoyée, et *errno* peut contenir **EDOM**.

## CONFORMITÉ

La fonction **floor**() est conforme à SVr4, BSD 4.3, C89. Les autres fonctions proviennent de C99.

## NOTES

Les spécifications SUSv2 et POSIX.1-2001 contiennent un passage sur les débordements (qui peuvent remplir *errno* avec **ERANGE** ou déclencher une exception). En pratique, aucun débordement ne peut se produire sur les machines actuelles, ce qui rend inutile cette gestion d'erreur. Plus précisément, le débordement ne peut se produire que si la valeur maximale de l'exposant est plus petite que le nombre de bits de la mantisse. Pour les nombres réels 32 bits et 64 bits obéissant à la norme IEEE-754, la valeur maximale de l'exposant est 128 (respectivement 1024) et le nombre de bits de la mantisse est 24 (respectivement 53).

## VOIR AUSSI

**ceil**(3), **lrint**(3), **nearbyint**(3), **rint**(3), **round**(3), **trunc**(3)

## TRADUCTION

Ce document est une traduction réalisée par Christophe Blaess <http://www.blaess.fr/christophe/> le 22 octobre 1996 et révisée le 17 juillet 2008.

L'équipe de traduction a fait le maximum pour réaliser une adaptation française de qualité. La version anglaise la plus à jour de ce document est toujours consultable via la commande : « **LANG=C man 3 floor** ». N'hésitez pas à signaler à l'auteur ou au traducteur, selon le cas, toute erreur dans cette page de manuel.

## NOM

erf, erff, erfl, erfc, erfcf, erfcl − Fonctions d'erreur et fonctions d'erreur complémentaires

## SYNOPSIS

**#include <math.h>**

**double erf(double** *x***);**
**float erff(float** *x***);**
**long double erfl(long double** *x***);**

**double erfc(double** *x***);**
**float erfcf(float** *x***);**
**long double erfcl(long double** *x***);**

Utilisez *−lm* à l'édition de liens pour lier avec la bibliothèque mathématique.

Exigences de macros de test de fonctionnalités pour la glibc (voir **feature_test_macros**(7)) :

Pour toutes les fonctions décrites ci-dessus : _BSD_SOURCE || _SVID_SOURCE || _XOPEN_SOURCE || _ISOC99_SOURCE ; ou *cc −std=c99*

## DESCRIPTION

La fonction **erf**() renvoie la fonction d'erreur de *x*, ainsi définie :

erf(x) = 2/sqrt(pi)* intégrale de 0 à x de exp(−t*t) dt

La fonction **erfc**() renvoie la fonction d'erreur complémentaire de *x*, qui vaut 1,0 − erf(x).

## CONFORMITÉ

SVr4, BSD 4.3, C99. Les variantes *float* et *long double* sont des demandes C99.

## VOIR AUSSI

**cerf**(3), **exp**(3)

## TRADUCTION

Ce document est une traduction réalisée par Christophe Blaess <http://www.blaess.fr/christophe/> le 23 octobre 1996 et révisée le 17 juillet 2008.

L'équipe de traduction a fait le maximum pour réaliser une adaptation française de qualité. La version anglaise la plus à jour de ce document est toujours consultable via la commande : « **LANG=C man 3 erf** ». N'hésitez pas à signaler à l'auteur ou au traducteur, selon le cas, toute erreur dans cette page de manuel.

## NOM

tgamma, tgammaf, tgammal − Véritables fonctions Gamma

## SYNOPSIS

**#include <math.h>**

**double tgamma(double** *x***);**
**float tgammaf(float** *x***);**
**long double tgammal(long double** *x***);**

Utilisez *−lm* à l'édition de liens pour lier avec la bibliothèque mathématique.

Exigences de macros de test de fonctionnalités pour la glibc (voir **feature_test_macros**(7)) :

**tgamma**(), **tgammaf**(), **tgammal**() : _XOPEN_SOURCE >= 600 || _ISOC99_SOURCE ; ou *cc −std=c99*

## DESCRIPTION

La fonction Gamma est définie ainsi :

Gamma(x) = intégrale de 0 à l'infini de tˆ(x−1) eˆ−t dt

Elle est définie pour tout réel sauf les entiers négatifs ou nuls. Pour un entier non négatif *m* on a

Gamma(m+1) = m!

et, plus généralement pour tout *x* :

Gamma(x+1) = x * Gamma(x)

De plus, pour toutes les valeurs de *x* en dehors des pôles, on peut écrire

Gamma(x) * Gamma(1 − x) = PI / sin(PI * x)

Ces fonctions renvoient la valeur de la fonction Gamma pour l'argument *x*. Le préfixe « t » signifie « true gamma » (« véritable fonction Gamma ») car il existe déjà une fonction **gamma**(3) qui retourne un autre résultat.

## ERREURS

Afin de vérifier les conditions d'erreur, vous devez mettre *errno* à zéro et appeler *feclearexcept(FE_ALL_EXCEPT)* avant d'invoquer ces fonctions. En retour, si *errno* est non nul ou si **fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** est non nul, une erreur s'est produite.

Une erreur d'échelle survient si *x* est trop grand. Une erreur de pôle survient si *x* est nul. Une erreur de domaine (ou erreur de pôle) survient si *x* est un entier négatif.

## CONFORMITÉ

C99.

## VOIR AUSSI

**gamma**(3), **lgamma**(3)

## TRADUCTION

Ce document est une traduction réalisée par Thierry Vignaud <tvignaud AT mandriva DOT com> en 2002 et révisée le 17 juillet 2008.

L'équipe de traduction a fait le maximum pour réaliser une adaptation française de qualité. La version anglaise la plus à jour de ce document est toujours consultable via la commande : « **LANG=C man 3 tgamma** ». N'hésitez pas à signaler à l'auteur ou au traducteur, selon le cas, toute erreur dans cette page de manuel.

## B.3 stdlib.h

A faire.

## NOM

malloc, calloc, free, realloc − Allocation et libération dynamiques de mémoire

## SYNOPSIS

**#include <stdlib.h>**

**void *calloc(size_t** *nmemb***, size_t** *size***);**
**void *malloc(size_t** *size***);**
**void free(void** **ptr***);*
**void *realloc(void** **ptr***, **size_t** *size***);**

## DESCRIPTION

**calloc**() alloue la mémoire nécessaire pour un tableau de *nmemb* éléments de taille *size* octets, et renvoie un pointeur vers la mémoire allouée. Cette zone est remplie avec des zéros. Si *nmemb* ou *size* vaut 0, **calloc**() renvoie soit NULL, soit un pointeur unique qui pourra être passé ultérieurement à **free**() avec succès.

**malloc**() alloue *size* octets, et renvoie un pointeur sur la mémoire allouée. Le contenu de la zone de mémoire n'est pas initialisé. Si *size* vaut 0, **malloc**() renvoie soit NULL, soit un pointeur unique qui pourra être passé ultérieurement à **free**() avec succès.

**free**() libère l'espace mémoire pointé par *ptr*, qui a été obtenu lors d'un appel antérieur à **malloc**(), **calloc**() ou **realloc**(). Si le pointeur *ptr* n'a pas été obtenu par l'un de ces appels, ou s'il a déjà été libéré avec *free(ptr)*, le comportement est indéterminé. Si *ptr* est NULL, aucune tentative de libération n'a lieu.

**realloc**() modifie la taille du bloc de mémoire pointé par *ptr* pour l'amener à une taille de *size* octets. **realloc**() conserve le contenu de la zone mémoire minimum entre la nouvelle et l'ancienne taille. Le contenu de la zone de mémoire nouvellement allouée n'est pas initialisé. Si *ptr* est NULL, l'appel est équivalent à *malloc(size)*, pour toute valeur de *size*. Si *size* vaut zéro, et *ptr* n'est pas NULL, l'appel est équivalent à *free(ptr)*. Si *ptr* n'est pas NULL, il doit avoir été obtenu par un appel antérieur à **malloc**(), **calloc**() ou **realloc**(). Si la zone pointée était déplacée, un *free(ptr)* est effectué.

## VALEUR RENVOYÉE

**calloc**() et **malloc**() renvoient un pointeur sur la mémoire allouée, qui est correctement alignée pour n'importe quel type de variable. Si elles échouent, elles renvoient NULL. NULL peut également être renvoyé par un appel réussi à **malloc**() avec un argument *size* égal à zéro, ou par un appel réussi de **realloc**() avec *nmemb* ou *size* égal à zéro.

**free**() ne renvoie pas de valeur.

**realloc**() renvoie un pointeur sur la mémoire nouvellement allouée, qui est correctement alignée pour n'importe quel type de variable, et qui peut être différent de *ptr*, ou **NULL** si la demande échoue. Si *size* vaut zéro, realloc renvoie NULL ou un pointeur acceptable pour **free**(). Si **realloc**() échoue, le bloc mémoire original reste intact, il n'est ni libéré ni déplacé.

## CONFORMITÉ

C89, C99.

## NOTES

Normalement, **malloc**() alloue la mémoire du tas et ajuste la taille de celui-ci suivant les besoins en utilisant **sbrk**(2). Lors de l'allocation de blocs mémoire plus grands que **MMAP_THRESHOLD** octets, l'implémentation **malloc**() de la glibc alloue la mémoire comme une projection anonyme privée avec **mmap**(2). **MMAP_THRESHOLD** vaut 128 Ko par défaut, mais peut être ajustée avec **mallopt**(3). Les allocations effectuées avec **mmap**(2) ne sont pas affectées par la limite de ressources **RLIMIT_DATA** (voir **getrlimit**(2)).

Le standard Unix98 réclame que **malloc**(), **calloc**() et **realloc**() remplissent *errno* avec **ENOMEM** en cas d'échec. La glibc suppose qu'il en est ainsi (et les versions glibc de ces routines le font). Si vous utilisez une implémentation personnelle de **malloc**() qui ne définit pas *errno*, certaines routines de bibliothèques peuvent échouer sans donner de raison dans *errno*.

Lorsqu'un programme se plante durant un appel à **malloc**(), **calloc**(), **realloc**() ou **free**(), ceci est presque

toujours le signe d'une corruption du tas. Ceci survient généralement en cas de débordement d'un bloc mémoire alloué, ou en libérant deux fois le même pointeur.

Les versions récentes de la bibliothèque C de Linux (libc postérieures à 5.4.23) et de la bibliothèque glibc 2.x incluent une implémentation de **malloc**() dont on peut configurer le comportement à l'aide de variables d'environnement. Quand la variable **MALLOC_CHECK_** existe, les appels à **malloc**() emploient une implémentation spéciale, moins efficace mais plus tolérante à l'encontre des bogues simples comme le double appel à **free**() avec le même argument, ou lors d'un débordement de tampon d'un seul octet (bogues de surpassement d'une unité, ou oubli d'un octet nul final d'une chaîne). Il n'est toutefois pas possible de pallier toutes les erreurs de ce type, et l'on risque de voir des fuites de mémoire se produire.

Si la variable **MALLOC_CHECK_** vaut zéro, toutes les corruptions du tas détectées sont ignorées silencieusement ; si elle vaut 1, un message de diagnostique est affiché sur *stderr*. Si cette variable vaut 2, la fonction **abort**(3) est appelée immédiatement. Si cette variable vaut 3, un message de diagnostique est affiché sur *stderr* et le programme abandonne. L'utilisation d'une valeur **MALLOC_CHECK_** non nulle est particulièrement utile car un crash pourrait sinon se produire ultérieurement, et serait très difficile à diagnostiquer.

## BOGUES

Par défaut, Linux suit une stratégie d'allocation optimiste. Ceci signifie que lorsque **malloc**() renvoie une valeur non nulle, il n'y a aucune garantie que la mémoire soit véritablement disponible. C'est vraiment un bogue craignos. Dans le cas où le système manque de mémoire, un ou plusieurs processus seront tués par l'infâme exterminateur de gestion mémoire (« OOM killer ». Dans le cas où Linux est utilisé dans des circonstances où il n'est pas souhaitable de perdre soudainement des processus lancés aléatoirement, et si de plus la version du noyau est suffisamment récente, on peut désactiver ce comportement en utilisant une commande du style :

```
# echo 2 > /proc/sys/vm/overcommit_memory
```

Voir également les fichiers *vm/overcommit−accounting* et *sysctl/vm.txt* dans le répertoire de la documentation du noyau.

## VOIR AUSSI

**brk**(2), **mmap**(2), **alloca**(3), **posix_memalign**(3)

## TRADUCTION

Ce document est une traduction réalisée par Christophe Blaess <http://www.blaess.fr/christophe/> le 4 novembre 1996 et révisée le 17 juillet 2008.

L'équipe de traduction a fait le maximum pour réaliser une adaptation française de qualité. La version anglaise la plus à jour de ce document est toujours consultable via la commande : « **LANG=C man 3 malloc** ». N'hésitez pas à signaler à l'auteur ou au traducteur, selon le cas, toute erreur dans cette page de manuel.

## NOM

qsort − Trier une table

## SYNOPSIS

**#include <stdlib.h>**

**void qsort(void *****base***, size_t** *nmemb*, **size_t** *size*,
**int(****compar***)(const void *, const void *));**

## DESCRIPTION

La fonction **qsort**() trie une table contenant *nmemb* éléments de taille *size*. L'argument *base* pointe sur le début de la table.

Le contenu de la table est trié en ordre croissant, en utilisant la fonction de comparaison pointée par *compar*, laquelle est appelée avec deux arguments pointant sur les objets à comparer.

La fonction de comparaison doit renvoyer un entier inférieur, égal, ou supérieur à zéro si le premier argument est respectivement considéré comme inférieur, égal ou supérieur au second. Si la comparaison des deux arguments renvoie une égalité (valeur de retour nulle), l'ordre des deux éléments est indéfini.

## VALEUR RENVOYÉE

La fonction **qsort**() ne renvoie pas de valeur.

## CONFORMITÉ

SVr4, BSD 4.3, C89, C99.

## NOTES

Parmi les routines de la bibliothèque utilisables comme argument *compar*, on a **alphasort**(3) et **versionsort**(3). Pour comparer des chaînes de caractères C, la fonction de comparaison peut appeler **strcmp**(3), comme dans l'exemple ci-dessous.

## EXEMPLE

Pour un exemple d'utilisation, voir l'exemple de la page **bsearch**(3).

Un autre exemple est le suivant qui trie les chaînes fournies comme argument sur la ligne de commande :

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

static int
cmpstringp(const void *p1, const void *p2)
{
   /* Les arguments de cette fonction sont des « pointeurs de
      pointeurs sur des char », mais les arguments de strcmp(3)
      sont des « pointeurs sur des char », d'où le transtypage
      et l'utilisation de l'astérique */

   return strcmp(* (char * const *) p1, * (char * const *) p2);
}

int
main(int argc, char *argv[])
{
   int j;

   assert(argc > 1);
```

```
    qsort(&argv[1], argc − 1, sizeof(char *), cmpstringp);

    for (j = 1; j < argc; j++)
        puts(argv[j]);
    exit(EXIT_SUCCESS);
}
```

## VOIR AUSSI

**sort**(1), **alphasort**(3), **strcmp**(3), **versionsort**(3)

## TRADUCTION

Ce document est une traduction réalisée par Christophe Blaess <http://www.blaess.fr/christophe/> le 5 novembre 1996 et révisée le 17 juillet 2008.

L'équipe de traduction a fait le maximum pour réaliser une adaptation française de qualité. La version anglaise la plus à jour de ce document est toujours consultable via la commande : « **LANG=C man 3 qsort** ». N'hésitez pas à signaler à l'auteur ou au traducteur, selon le cas, toute erreur dans cette page de manuel.

RANDOM(3) Manuel du programmeur Linux RANDOM(3)

## NOM

random, srandom, initstate, setstate − Générateur de nombres aléatoires

## SYNOPSIS

**#include <stdlib.h>**

**long int random(void);**

**void srandom(unsigned int** *seed***);**

**char *initstate(unsigned int** *seed***, char *****state***, size_t** *n***);**
**char *setstate(char *****state***);**

Exigences de macros de test de fonctionnalités pour la glibc (voir **feature_test_macros**(7)) :

**random**(), **srandom**(), **initstate**(), **setstate**() : _SVID_SOURCE || _BSD_SOURCE || _XOPEN_SOURCE >= 500

## DESCRIPTION

La fonction **random**() utilise une fonction non linéaire pour engendrer des nombres pseudo-aléatoires entre 0 et **RAND_MAX**. La période de ce générateur est très grande, approximativement 16 * ((2ˆ31) − 1).

La fonction **srandom**() utilise son argument comme « graine » pour engendrer une nouvelle séquence de nombre pseudo-aléatoires qui seront fournis lors des appels à **random**(). Ces séquences sont reproductibles en invoquant **srandom**() avec la même graine. Si aucune graine n'est fournie, la fonction **random**() utilise automatiquement une graine originale de valeur 1.

La fonction **initstate**() permet d'initialiser une table d'états *state* pour l'utiliser avec **random**(). La taille *n* de la table est utilisée par **initstate**() pour déterminer le niveau de sophistication du générateur de nombre aléatoires. Plus grande est la table d'état, meilleurs seront les nombres aléatoires. *seed* est la graine utilisée pour l'initialisation, indiquant un point de départ pour la séquence de nombres, et permet de redémarrer au même endroit.

La fonction **setstate**() modifie la table d'états utilisée par la fonction **random**(). La table d'état *state* est alors utilisée comme générateur de nombres aléatoires jusqu'au prochain appel de **initstate**() ou **setstate**(). *state* doit d'abord être initialisée avec **initstate**() ou être le résultat d'un appel précédent à **setstate**().

## VALEUR RENVOYÉE

La fonction **random**() renvoie une valeur entre 0 et **RAND_MAX**. La fonction **srandom**() ne renvoie pas de valeur. Les fonctions **initstate**() et **setstate**() renvoient un pointeur sur la table d'états précédente, ou NULL en cas d'erreur.

## ERREURS

**EINVAL**

Une table d'états de moins de 8 octets a été fournie à **initstate**().

## CONFORMITÉ

BSD 4.3, POSIX.1-2001.

## NOTES

Actuellement, les valeurs optimales *n*, pour la taille de la table d'états sont 8, 32, 64, 128, et 256 octets. Les autres valeurs seront arrondies à la taille la plus proche. Essayer d'utiliser moins de 8 octets déclenche une erreur.

Cette fonction ne devrait pas être utilisée dans les cas où plusieurs threads utilisent **random**() et où le comportement doit être reproductible. Utilisez **random_r**(3) pour celà.

La génération de nombres alétoires est un sujet complexe. *Numerical Recipes in C: The Art of Scientific Computing* (William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling; New York: Cambridge University Press, 2007, 3rd ed.) fournit une excellente discussion sur les problèmes pratiques de génération de noms aléatoires dans le chapitre 7 (Random Numbers).

Pour une discussion plus théorique qui traite également beaucoup de problèmes pratiques en profondeur, voir le chapitre 3 (Random Numbers) du livre de Donald E. Knuth *The Art of Computer Programming*, volume 2 (Seminumerical Algorithms), 2nd ed.; Reading, Massachusetts: Addison-Wesley Publishing Company, 1981.

## VOIR AUSSI

**drand48**(3), **rand**(3), **srand**(3)

## TRADUCTION

Ce document est une traduction réalisée par Christophe Blaess <http://www.blaess.fr/christophe/> le 7 novembre 1996 et révisée le 17 juillet 2008.

L'équipe de traduction a fait le maximum pour réaliser une adaptation française de qualité. La version anglaise la plus à jour de ce document est toujours consultable via la commande : « **LANG=C man 3 random** ». N'hésitez pas à signaler à l'auteur ou au traducteur, selon le cas, toute erreur dans cette page de manuel.

*Version provisoire de ces notes de cours. Elles seront mises à jour au cours du semestre.*