Introduction à la programmation impérative – Éléments de langage C (T1 – PG109)

Année 2020 – 2021

Guillaume Mercier

Table des matières

IV La quintessence de C

Avant-propos

Organisation de l'enseignement

> **Rappel** : en vertu du règlement pédagogique signé par vos soins, l'assiduité aux cours est *obligatoire*.

Objectifs du cours de PG109

Pré-requis

Les pré-requis pour ce cours sont les suivants :

1. de la curiosité ;
2. de la logique ;
3. de la rigueur ;
4. un cerveau en état de marche ;
5. un minimum d'envie d'apprendre des choses ;
6. un minimum d'implication dans ce cours.

Outils et logiciels

En ce qui concerne les outils et logiciels dont vous aurez besoin pour ce cours, en voici une liste raisonnable :

— un éditeur de texte
— un compilateur (gcc *a priori*)
— des terminaux pour lancer vos programmes
— et surtout : le manuel (`man`). Le manuel sera votre meilleur ami ce semestre. Et je vous rappelle que cela n'a **aucun** sens de chercher sur internet, parce que l'information que vous allez récupérer ne correspond pas forcément au système pour lequel vous allez développer vos programmes. De plus, le jour où vous n'aurez plus/pas de réseau, le manuel sera toujours disponible.

En particulier, vous devrez avoir le réflexe de regarder le manuel dès que sera mentionné un prototype ou un nom de fonction durant ce cours.

Comment utiliser ce document ?

Premièrement, il ne faut **pas** apprendre ce document par cœur. Ensuite, ce document n'a pas pour vocation a être lu linéairement. Certains points seront obscurs à la première lecture, mais ils s'éclairciront par la suite : n'hésitez pas à revenir sur les points laissés en suspens au besoin. Certaines parties sont plus pour votre culture générale et ne seront pas forcément abordés durant les cours : c'est normal donc pas d'inquiétudes à avoir à ce sujet. Enfin, n'hésitez pas à compléter ce document par d'autres lectures : il y a encore beaucoup de points que je n'ai pas abordés.

Conventions

Les conventions suivantes ont été adoptées dans ce document :

Codes d'exemple

Du code entre deux lignes horizontales :

```
1 Rien dans ce programme pour le moment :
2 rassurez-vous, ca va vite venir...
```

programme_vide.c

est récupérable sur mon compte dans le répertoire `/net/ens/mercier/PG109/Exemples/` pour être compilé et testé. Le nom du programme est mentionné avant la seconde ligne horizontale au centre de la page (ici, `programme_vide.c`). Toutes les lignes sont numérotées pour une meilleure lisibilité et analyse du code.

Points remarquables et exercices

Signale un point notable à retenir ou auquel il faut prêter attention.

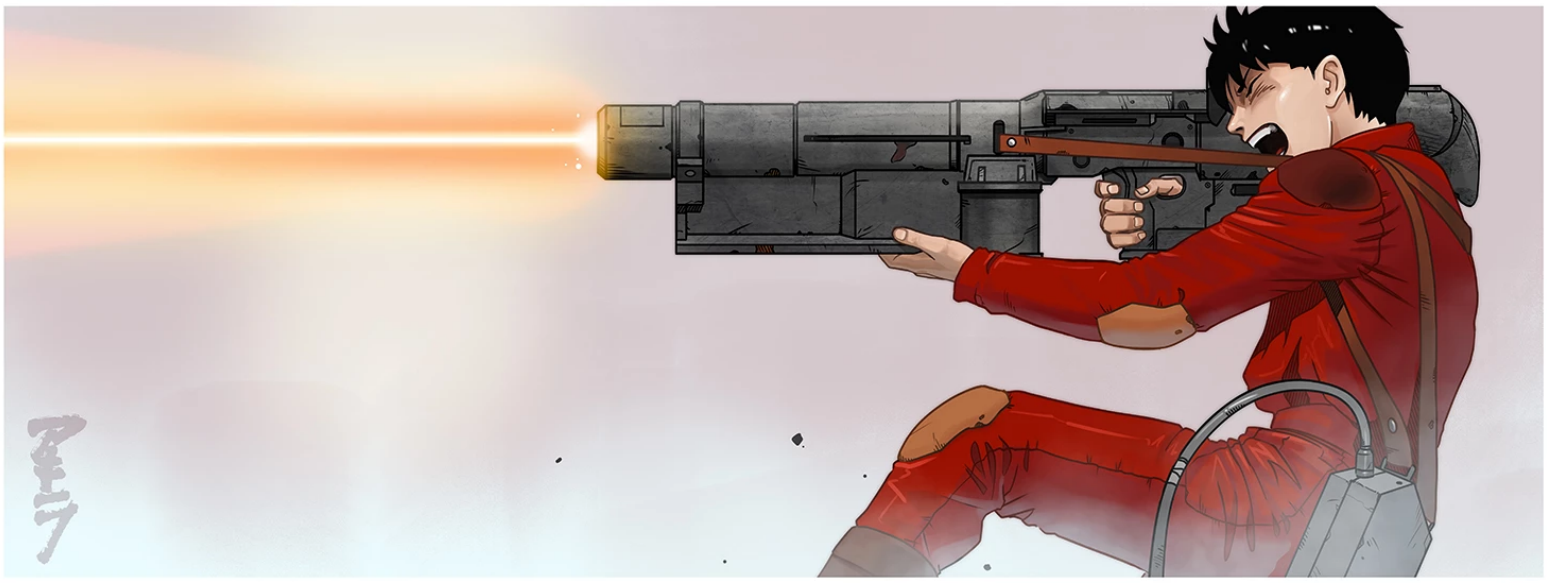
Indique un petit exercice/manipulation à faire après avoir lu le paragraphe correspondant.

Indique un exercice de niveau intermédiaire.

Indique un exercice de niveau avancé.

Ce cours est approuvé par l'Institut Banzaï.

Premiers pas en C



1. Vue d'ensemble du langage C

C is all about memory management.

Leonard Shelby

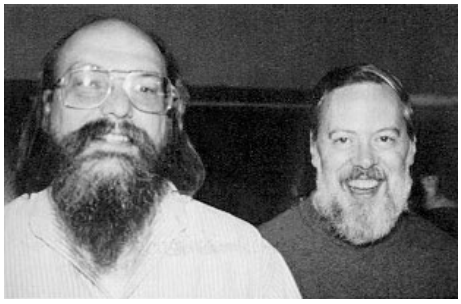
C is easy to learn but hard to master.

Nolan Bushnell

C est le langage le plus classe du monde.

Georges Abitbol

C est un langage de programmation qui a été mis au point en 1978 par Kenneth Thomson et Dennis Ritchie (Brian Kernighan a apporté son concours plus tard). C a été nommé ainsi car il est le successeur du langage B, lui-même créé par Dennis Ritchie et Kenneth Thomson.



Thomson et Ritchie, la *bromance* qui a bouleversé l'Amérique (et révolutionné l'informatique).

Thomson et Ritchie ont également créé le système d'exploitation UNIX et ce langage (le C) a d'abord été mis au point afin de pouvoir implémenter ce système (UNIX) avec autre chose que du langage assembleur pour bénéficier d'une **portabilité**, c'est-à-dire la capacité de réutiliser un code sur une nouvelle architecture matérielle (le processeur, essentiellement) ou sur un système d'exploitation différent. En effet, le langage assembleur étant spécifique à un type de processeur donné, un programme écrit dans un tel langage doit être intégralement réécrit lorsque l'on souhaite déployer le programme sur un autre type de processeur. [1].

1.1 Caractéristiques du langage C

C est un exemple de langage suivant le paradigme dit **impératif**, c'est-à-dire qu'une application ou un programme écrit dans ce langage (le code source) prend la forme d'une succession d'*instructions*, qui seront exécutées dans l'ordre où elles sont écrites [2].

De plus, un code écrit en C passe usuellement à travers une chaîne de **compilation** pour produire un exécutable. Plus précisément le code source (concrètement, il s'agit de texte) va être transformé en code exécutable par le biais d'un outil externe, que l'on appelle un compilateur (cf. Section 2). De façon générale, il existe une autre façon d'exécuter un programme écrit dans un langage de programmation, c'est de l'**intepréter**. Dans ce cas, le code source est passé à ce que l'on appelle un interpréteur (qui est aussi un programme) , qui va se charger d'exécuter les instructions du programme. Certains langages font plutôt l'objet d'une compilation (c'est le cas de C) et d'autres langages sont plutôt interprétés (ex : Lisp, Scheme). Il existe des langages qui font un peu les deux : en Java, par exemple, le code-source est compilé dans ce qu'on appelle du *bytecode* et c'est ce *bytecode* qui est interprété par la *Java Virtual Machine*, qui joue le rôle d'interpréteur. Voici quelques différences entre compilation en assembleur/binaire (cas de C) et interprétation :

— une fois que le programme est compilé, c'est-à-dire qu'il est sous forme exécutable, il n'y a plus besoin du compilateur pour exécuter le programme. En revanche, dans le cas d'un programme interprété, l'interpréteur est nécessaire pour *chaque* exécution du programme ;
— un programme compilé ne peut s'exécuter que sur les processeurs/systèmes pour lesquels il a été compilé au départ. En cas de changement de système ou d'architecture, une recompilation est indispensable. Dans le cas d'un programme interprété, il est exécutable sur n'importe quel processeur/système à partir du moment où l'interpréteur est disponible pour ce couple ;

1. En fait, c'est surtout l'architecture du processeur et son jeu d'instructions qui sont déterminants. Une nouvelle génération de processeurs peut voir le jour sans que le jeu d'instructions soit beaucoup modifié : dans ce cas, un programme écrit en assembleur n'aura sans doute pas à être intégralement réécrit, mais plus vraisemblablement modifié à la marge.

2. Ce n'est pas forcément le cas en pratique, notamment dans le cas de processeurs qui font de l'exécution *out of order*, mais nous allons considérer cela comme vrai dans ce cours (et cette année plus généralement).

— un programme compilé s'exécute généralement plus rapidement qu'un programme interprété car il n'y a pas besoin d'intermédiaire lors de l'exécution.

Quant au **typage** des variables, il est :

— statique : la vérification du typage est effectuée au moment de la compilation (par opposition à une typage dynamique, où cette vérification se fait au moment de l'exécution proprement dite) ;
— explicite : c'est à vous en tant qu'utilisateur/utilisatrice d'indiquer les types que vous allez utiliser dans le programme ou l'application, dans les déclarations de variables ou de fonctions ;
— faible : C autorise des *conversions* de type lors de l'exécution d'un programme.

Enfin, C peut être considéré comme un langage relativement *simple*, dans la mesure où le nombre de mots-clefs et de constructions syntaxiques est peu élevé. De plus, il reste relativement proche de l'assembleur et du matériel (il est d'ailleurs possible d'intégrer du code assembleur dans un code-source en C, ce qui le rend non portable). Cette proximité du langage avec le système s'exprime notamment au niveau de la gestion mémoire : en C, il incombe [3] à l'utilisateur/utilisatrice d'effectuer cette gestion explicitement. Aucun mécanisme de type ramasse-miettes (*garbage collector)* n'existe et la possibilité de fuites mémoire n'est pas exclue. Cela laisse une grande liberté au programmeur pour écrire du code très efficace : C est rarement battu en terme de vitesse d'exécution dans les tests comparatifs avec d'autres langages.

La mémoire utilisée par le programme est ce qu'on appelle de la **mémoire virtuelle** . Cette mémoire est stockée dans la **mémoire physique** de la machine (la RAM) au fur et à mesure de l'exécution du programme. Le pourquoi de l'existence de cette correspondance entre mémoire virtuelle et mémoire physique, ainsi que ses modalités fera l'objet d'une partie du cours de systèmes d'exploitation en S8 (IF207). Il est cependant important de comprendre que la mémoire utilisée par un programme est intégralement libérée par le système quand ce même programme se termine. Néanmoins, cela ne veut pas dire que vous ne devez pas apporter un soin méticuleux à votre gestion de la mémoire. Nous y reviendrons ultérieurement (cf. section 8.5). Il existe de plus des outils spécifiques qui pourront vous aider à ce niveau, comme gdb ou encore Valgrind, dont nous verrons l'utilisation en cours de PG110 au semestre prochain.

1.2 Pourquoi apprendre le C ?

C est un langage relativement ancien, mais il reste toujours très utilisé, notamment pour le développement d'applications scientifiques, pour le développement de systèmes (par exemple, le système GNU/Linux est écrit en C [4]). Il est utilisé en raison de sa rapidité d'exécution et de ses performances. C'est un langage prisé dans les systèmes embarqués ou encore en traitement de signaux et d'images.

Au-delà de sa proximité avec le système, C est le pilier d'un ensemble de langages qui en sont des dérivés ou des extensions :

— C++ : à l'origine, il s'agit d'extensions du langage C permettant d'utiliser le **paradigme objet** dans des programmes C. Avec le temps, et les révisions successives de ces langages, C et C++ ont eu tendance à diverger au point de devenir partiellement incompatibles. Vous pourrez trouver une introduction aux versions récentes de C++ à cet URL : `https://changkun.de/modern-cpp/pdf/modern-cpp-tutorial-en-us.pdf` ;
— Objective-C : il s'agit d'une autre version objet de C, dont le développement à débuté dans les années 80, et plus ou moins en parallèle avec C++. Objective-C fût utilisé pendant un

3. voire décombe
4. Avec certaines parties en assembleur…

temps dans les systèmes d'exploitation d'Apple : macOS et son dérivé iOS, basés sur la bibliothèque de classes Cocoa. Ce langage a également été employé pour le développement d'applications mobiles sur smartphone de marque Apple avant d'être remplacé par Swift ;
— C# (C sharp) : est une autre version de C orientée objet et commercialisée par Microsoft depuis 2002. Elle est destinée aux développements sur la plateforme de Microsoft appelée .NET.

Quelle version de C ?

Depuis sa création, le langage C a connu plusieurs révisions, les plus importantes étant celle de 1989 (C89, ou ANSI C) qui a vu le langage devenir un standard, celle de 1999 (C99) qui a introduit notamment la déclaration de variables à la volée, celle de 2011 (C11) et celle de 2018 (C18) qui est une version corrigeant certains problèmes de C11 sans rajouter de nouvelles fonctionnalités.

1.3 Premier programme

Voici le code source de votre premier programme C (du moins, dans ce cours) :

```c
#include␣<stdio.h>

int␣main(void)
{
printf("Hello,␣world!\n");
return␣0;
}
```

hello_world.c

Les espaces sont volontairement indiqués dans le code ci-dessus. Bien entendu, il ne faut **pas** écrire ces caractères (␣) dans vos propres programmes !

1. Écrivez ce programme à l'aide d'un **éditeur de texte** approprié (en particulier un logiciel de **traitement de texte** n'est **pas** un bon choix)
2. Compilez ce programme à l'aide la commande suivante : `gcc hello_world.c`
3. Exécutez ce programme en le lançant avec la commande suivante : `./a.out`

Analyse du code du programme `hello_world`

Que fait ce programme ? Ainsi que vous avez pu le constater lors de son exécution, la **chaîne de caractères** `"Hello, world!"` est affichée à l'écran de l'ordinateur. Regardons d'un peu plus près le code de ce programme :

Ligne 1 : `#include <stdio.h>` cette ligne commence avec le caractère `#` ce qui indique qu'il s'agit d'une directive de *preprocessing* (cf. section 2.1.1). En effet **toutes** ces directives commencent obligatoirement par ce caractère. La directive `#include` permet de prendre le contenu du fichier dont le nom est donné entre les symboles < et > (`stdio.h` ici, ce qui signifie ***st**andard **i**nput **o**utput*) et de le copier dans le fichier source .c. Dans le cas de notre programme, cela va permettre d'utiliser la fonction `printf` qui permet d'afficher quelque chose à l'écran. Schématiquement, vous allez réutiliser des fonctions (des morceaux de code) pré-existantes, un peu comme des outils et pour ce faire, il est impératif de déclarer cette utilisation. C'est le

but de la **directive** `#include <truc>`, que vous pouvez interpréter comme : «*Je vais utiliser dans mon programme la boîte à outils* `truc`».

Ligne 2 : Cette ligne vide n'est pas nécessaire mais permet une meilleure lisibilité du code source. C est un langage pour lequel les espaces et les sauts de lignes ne sont pas nécessaires pour la compilation [5]. De même, le nombre de colonnes pour écrire le code n'est pas limité. Il est donc possible d'écrire l'intégralité d'un programme C (et ce, quelle que soit sa taille) sur une unique ligne ! Maintenant, vous vous doutez bien que la lisibilité d'un tel programme sera très mauvaise et il est important d'aérer vos codes afin d'en faciliter la lecture, surtout quand vous allez travailler à plusieurs sur un projet (par exemple).

Ligne 3 : `main()` est ce qu'on appelle une **fonction** en langage C. Pour simplifier, les fonctions permettent d'organiser votre code en entités logiques de taille réduite afin de faciliter la réutilisation ainsi que sa modularité. Une fonction en C peut accepter ou non des **arguments** et la liste de ceux-ci est située entre les parenthèses (`(` et `)`) suivant le nom de la fonction (ici, `main`. Dans notre exemple, la fonction ne prend pas de paramètres et la liste est donc vide. Un programme C possède toujours **au moins** une fonction, c'est la fonction `main`, qui est le **point d'entrée du programme**. En effet, quand un programme C débute son exécution, c'est cette fonction `main` qui est exécutée la première. Ainsi, s'il est possible d'écrire l'intégralité du code dans cette fonction `main`, cela est fortement déconseillé, là encore pour des questions de lisibilité, de réutilisation et de maintenance du code du programme. Nous reviendrons plus longtemps sur les fonctions au Chapitre6.

Ligne 4 : l'accolade ouvrante `{` permet de délimiter le **corps de la fonction** `main`, c'est-à-dire l'ensemble des instructions qui constituent la fonction. Dans notre exemple, ce corps n'est constitué que d'une unique instruction.

Ligne 5 : l'unique instruction de la fonction `main` est un **appel** à la fonction `printf`, qui prend en argument un unique paramètre, à savoir la chaîne de caractères `Hello, world!`. Cette chaîne est délimitée par les caractères `"` et `"`. Notez la présence d'un caractère spécial, `\n`, qui ne s'affiche pas et qui permet un retour à la ligne lors de l'affichage.

Ligne 6 : l'accolade fermante `}` indique la fin de la fonction `main`.

1.4 Conseils de développement en C

Développer des programmes en C demande un peu de rigueur et de discipline. Pour ce faire, il est important d'énoncer un certain nombre de règles qui vont vous aider à gagner en efficacité dans le développement de vos programmes en C. Ces règles n'ont toutefois rien d'obligatoire mais je vous recommande fortement de vous y conformer, en particulier la première.

Conseil 1 **Indentez** vos programmes. L'indentation est l'action d'insérer des espaces en début de ligne afin que la structure du programme apparaisse plus clairement. En effet, si vous alignez toutes vos lignes de code sur la gauche, cela devient rapidement illisible. Quelques espaces suffisent, car trop d'indentation nuit également à la lisibilité du code. Par exemple, il est courant d'indenter légèrement le corps de la fonction afin de le faire ressortir. Dans le cas de notre programme `hello_world`, l'indentation du corps donnerait la chose suivante :

```
1 #include␣<stdio.h>
2
3 int␣main(void)
4 {
5 ␣␣printf("Hello,␣world!\n");
```

5. Ce qui n'est pas le cas d'autres langages comme Fortran, par exemple.

```
6  ␣␣return␣0;
7  }
```

hello_world_avec_indentation.c

Vous noterez la présence de deux espaces devant l'appel à `printf` à la ligne 5.

Conseil 2 **Nommez** vos variables et vos fonctions de façon explicite afin que la lecture et la compréhension du code en soient facilitées. Par exemple, des variables nommées `x`, `y` et `z` sont moins explicites que des variables nommées `longueur`, `largeur` ou `profondeur`. De même, nommer une fonction `truc` sera sans doute moins pertinent que de la nommer `calcul_aire`. Il est par ailleurs inutile (et contre-productif) d'utiliser des identificateurs (des noms) trop longs.

Conseil 3 **Commentez** vos programmes. Vous allez vous rendre compte qu'il est plus difficile de relire du code que d'en écrire. Vous êtes encouragés à commenter vos codes, c'est-à-dire placer des indications concernant ce que vous avez développé. Il existe deux moyens d'insérer des commentaires dans un programme en C : tout ce que vous écrirez entre `/*` et `*/` ne sera pas pris en compte par le compilateur. Cela permet de d'écrire des commentaires sur plusieurs lignes. Depuis C99, il est aussi possible d'utiliser les commentaires de C++ : toute ligne débutant par `//` n'est pas prise en compte par le compilateur. Veuillez noter que dans ce cas, **chaque** ligne doit être commentée individuellement. Enfin, ne confondez pas commentaires et paraphrase : il est inutile de réécrire en français ce que le code fait. En revanche, commenter des points délicats est plus qu'indiqué.

Conseil 4 **Testez** régulièrement vos programmes. Le développement, c'est 10% d'écriture et 90% de corrections d'erreurs. Afin de détecter ces dernières plus facilement [6], il est indispensable de ne pas attendre d'avoir écrit une grosse quantité de code. Par exemple, il est conseillé de faire des tests à chaque fois que vous écrirez une nouvelle fonction. Et si cette fonction est elle-même très longue, alors n'hésitez pas à faire des tests intermédiaires.

6. Nous verrons au semestre prochain dans le cours de PG110 des outils pour vous y aider.

2. Création d'un programme C exécutable

2.1 Compilation : notion et phases

Le compilateur est un **programme** indispensable lorsqu'on développe des programmes en C (en programmation classique ou système). Le compilateur sert à transformer du **texte** (le code source) en **programme exécutable** (code exécutable par le microprocesseur de la machine à laquelle le programme que vous écrivez est destiné).

En règle générale, il y aura toujours un compilateur C d'installé sur les machines sur lesquelles vous travaillerez (le compilateur `cc` sera au moins présent). Nous allons utiliser un autre compilateur, `gcc`. `gcc` est un compilateur libre et *open source* ; c'est un des nombreux projets GNU. `gcc` est un programme complexe d'utilisation qui accepte un nombre conséquent d'options (tapez la commande `man gcc` pour vous en convaincre !). Bien entendu, il n'est pas question pour vous de connaître le rôle de **toutes** ces options, mais nous en verrons quelques-unes lors des TP et manipulations pratiques.

La compilation d'un programme écrit en C se décompose en quatre phases :

1. la phase de **preprocessing** (production d'une nouvelle version du code source)
2. la phase de compilation proprement dite (production du code **assembleur**)
3. la phase d'assemblage (production du code **objet**)
4. la phase d'édition de lien (production du programme **exécutable**)

Afin de vous donner une idée concrète de ces phases, reprenons le programme `Hello_World.c` :

~

```c
#include <stdio.h>

int main(void)
{
  printf("Hello, world!\n");
  return 0;
}
```

hello_world_avec_indentation.c

Voyons ce que donnent les phases compilations de ce programmes les unes après les autres avec ce programme très simple d'exemple.

2.1.1 Phase I : le *preprocessing*

Lors de cette phase, le **préprocesseur** du compilateur C va transformer le code source original en un **nouveau** code source. Toutes les lignes commençant par le caractère # sont celles qui vont être traitées par le pré-processeur. Dans notre programme d'exemple, une seule ligne est concernée.

Avec `gcc`, il est possible de voir le résultat de cette phase en lui passant l'option `-E` (pour *expand*).

1. Tapez la commande `gcc -E hello_world.c > new_hello_world.c` (quand cette option est passée au compilateur, ce dernier s'arrête après la phase de *preprocessing*).
2. Regardez le nouveau code source (dans le fichier `new_hello_world.c`). Comparez la taille du fichier d'origine avec celle du fichier produit. Que constatez-vous ?
3. Le fichier produit est-il dépendant du processeur ?
4. Le fichier produit est-il dépendant du système d'exploitation ?

2.1.2 Phase II : la compilation

Lors de cette phase, le code source résultant de du preprocessing est traduit en code **assembleur**. Cette phase va produire un fichier `hello_world.s`.

1. Tapez la commande `gcc -S hello_world.c` (quand cette option est passée au compilateur, ce dernier s'arrête après la phase de **compilation**).
2. Regardez le code assembleur produit c'est à dire le contenu du fichier `hello_world.s`. Connaissant le rôle du programme d'origine, que reconnaissez-vous dans le code assembleur ?
3. Le fichier produit est-il dépendant du système d'exploitation ?

Bien entendu, le code assembleur généré est spécifique au processeur cible. Pour vous en convaincre, vous pouvez refaire cette manipulation en changeant de type de processeur. Notez bien que ce fichier contient toujours du **texte**, mais un texte différent de celui du programme source (`hello_world.c`). En l'état, ce code n'est toujours pas compréhensible par le processeur de l'ordinateur sur lequel vous travaillez, encore moins exécutable.

2.1.3 Phase III : l'assemblage

Lors de cette phase, le code assembleur produit lors de la phase précédente (`hello_world.s`) est traduit en code **objet** : c'est le fichier `hello_world.o`.

1. Tapez la commande `gcc -c hello_world.s` (quand cette option est passée compilateur, ce dernier s'arrête après la phase d'**assemblage**).

2. Regardez le code objet produit, contenu dans le fichier `hello_world.o`). Que constatez-vous ?

À partir de ce moment, le contenu du fichier objet est du code **binaire**, compréhensible par le processeur de la machine, ce qui n'était pas le cas auparavant. Si vous utilisez un programme permettant de lire du binaire (et oui, ça existe !), vous verrez que le fichier objet contient quelque chose d'assez ressemblant à l'assembleur : tapez la commande `nm hello_world.o` et comparez le résultat avec le contenu du fichier `hello_world.s` pour vous en convaincre.

2.1.4 Phase IV : l'édition de liens

Bien que ce code soit *a priori* exécutable par l'ordinateur, ce n'est pas le cas en pratique car le programme fait appel à la fonction `printf` dont le code est situé dans une **bibliothèque** fournie par le système d'exploitation UNIX.

1. Tapez la commande `gcc -o hello_world hello_world.o` et exécutez le programme correspondant.
2. Fonctionne-t-il ?

Cette commande récupère automatiquement la bibliothèque qui contient le code de `printf`, sans que vous ayez besoin de spécifier son emplacement. Cette bibliothèque est la `libc.a` (le `.a` signifie **archive**, c'est donc une bibliothèque statique). Elle est indispensable pour le développement de programmes en langage C (les curieux peuvent exécuter la commande `gcc -v -o hello_world hello_world.o` pour voir ce qui se passe vraiment lors de cette phase !).

2.2 En pratique

Par la suite, vous pourrez compiler le code source (`mon_prog.c`) directement en code exécutable (`mon_prog`), sans avoir besoin de faire la compilation phase par phase. Ceci se fait directement de la façon suivante :

```
gcc toto.c
```

Cette commande produit un fichier exécutable appelé `a.out`. C'est le nom **par défaut** d'un fichier exécutable produit par `gcc`. Vous pouvez spécifier le nom de l'exécutable au moment de la compilation à l'aide de l'option `-o` (pour *output*) ainsi :

```
gcc -o mon_prog prog.c
```

Bien entendu, le choix du nom est libre, il n'est pas obligatoire d'appeler `mon_prog` un exécutable correspondant à un fichier de nom `mon_prog.c` (cf. l'exemple ci-dessus)

Attention cependant à l'utilisation de l'option `-o` : le nom choisi pour le programme exécutable doit se trouver **immédiatement** après l'option. En particulier, si vous vous trompez dans l'ordre et que vous mettez juste après `-o` le nom du fichier source, ce dernier sera écrasé par `gcc` !

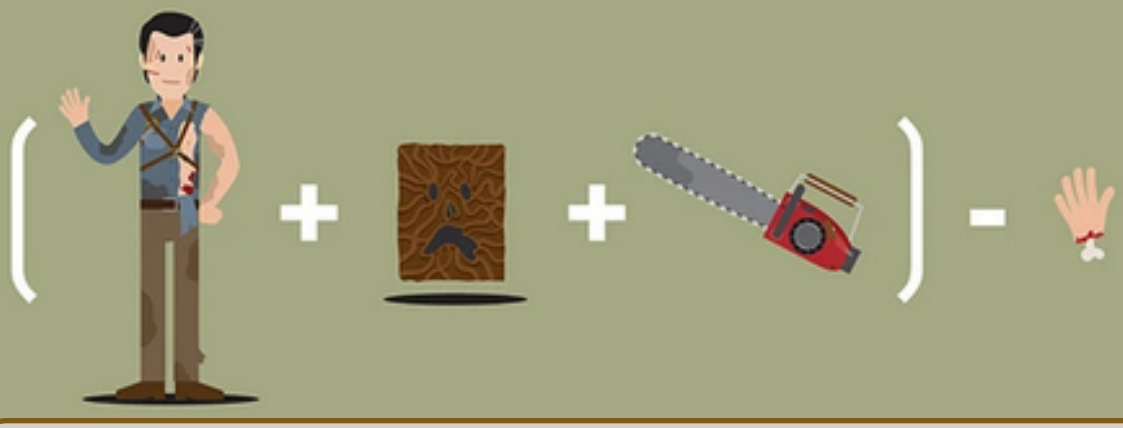
Un autre type de quatrième phase (*spoiler : à la fin, les fourmis gagnent*).

2.3 Lancement et exécution d'un programme en C

Une fois que le programme exécutable a été généré suite à la compilation du code-source, vous devez l'exécuter. Pour cela, tapez le nom de l'exécutable dans un terminal puis appuyez sur "Entrée". À ce moment, une commande appelée `exec` est appelée, qui va déclencher l'exécution du programme. De façon très schématique, tout se passe comme si le programme était chargé dans la mémoire de la machine et que l'exécution débutait avec l'appel à la fonction `main` du programme.

Un code source n'est que du **texte** : en particulier, il ne s'agit **pas** de code exécutable. Donc, pour lancer un programme, vous devez d'abord créer un **exécutable** puis le lancer en tant que commande. Lancer le fichier source directement dans le terminal ne sert à rien (ex. : `./hello_world.c`). De même, passer le fichier source en mode exécutable (avec une commande du style `chmod +x`) ne créera **jamais** un programme exécutable. Un programme exécutable est avant tout constitué d'instructions binaires compréhensibles par le processeur cible.

II Construction de programmes basiques



3. De l'expression vers l'instruction

3.1 Notion d'expression

C est un langage, qui possède une grammaire et une syntaxe. Un programme **syntaxiquement correct** est un programme qui est donc formellement correct du point de vue du langage tel qu'il est défini formellement. Cela signifie qu'à la compilation, il n'y aura pas d'erreurs. Cependant, l'absence de détection d'erreur à la compilation ne signifie pas que le programme est correct du point de vue de l'utilisateur/utilisatrice : en particulier, il est susceptible de produire un résultat ou un effet différent de celui attendu [1].

La première étape est donc l'apprentissage de l'écriture d'un code syntaxiquement correct. Le code de votre programme sera composé d'**instructions**, qui sont elles-mêmes le résultat de la composition d'**expressions** entre-elles.

3.1.1 Caractérisation d'une expression

En C, une expression est à la fois :

— **évaluable**, c'est-à-dire qu'elle produit un résultat qui possède une valeur ;
— **typée**, c'est-à-dire que ce résultat possède un type.

3.1.2 Expressions de base

Il existe en C des expressions de base, qui peuvent être ensuite utilisées pour former des expressions composéés. Ces expressions de base sont :

— les **identificateurs** : les identificateurs sont en fait des noms. Cela concerne les noms de variables, les noms de fonctions, mais aussi les mots-clefs du langage lui-même (qui sont donc réservés et inutilisables). Un identificateur est composé de lettres et de chiffres. Il doit obligatoirement commencer par une lettre, sachant que l'*underscore* (_) compte comme une lettre. Les majuscules et les minuscules sont considérées comme distinctes ;
— les **constantes** : cela recouvre aussi bien les constantes numériques (entières et flottantes),

1. C'est pour cela que vous êtes encouragés à effectuer des tests régulièrement pour vérifier que vos développements correspondent bien à vos attentes.

les caractères constants et les énumérations (cf. section 4.2.3). Par exemple, `12`, `5.67`, `-13` et `'z'` sont des constantes pour le langage C ;

— les **chaînes de caractères constantes** , c'est-à-dire celles qui sont délimitées par les symboles `"` et `"`, sont des expressions de base en C (par exemple : `"coucou"` est une expression de base) ;

— les **expressions parenthèsées** (par exemple `(` *expression* `)`) sont des expressions de base en C ;

++ une **sélection générique** est une expression de base (cf. section 4.1.5) dont le type et la valeur sont déterminés par l'assocation générique sélectionnée.

La table 3.1 donne la liste complète des mots-clefs du langage C [2]. On remarquera que cette liste est relativement réduite (une trentaine de nots-clefs) et qu'y figurent les identificateurs des **types de base**, c'est-à-dire : `char`, `short`, `int`, `long`, `float`, `double` et `void` (qui n'est pas tout à fait un type de base).

TABLE 3.1: Liste des mots-clefs du langage C

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

3.1.3 Expressions composées élémentaires

Ces expressions de base peuvent être composées pour former des expressions plus complexes. Cette composition est possible notamment grâce à des opérateurs, notamment arithmétiques. Afin de fixer les idées, si `1` et `2` sont deux expressions de base, alors `1 + 2` est une expression composée. L'ensemble des opérateurs disponibles en C est donné par la table 3.2 en section 3.2.4. Les sections suivantes donnent quelques exemples simples (mais importants) d'expressions composées.

Déclaration de variable

Nous verrons plus en détails les variables dans le Chapitre 4. La déclaration d'une variable est une expression composée. En effet une déclaration de variable nécessite au moins deux identificateurs : un nom de type (rappel : C est un langage typé) et un nom de variable :

nom-de-type *nom-de-variable*

Exemples

`int x` $\Longrightarrow$ déclaration d'une variable appelée `x`, de type entier

`char bidule` $\Longrightarrow$ déclaration d'une variable appelée `bidule`, de type caractère

Affectation de variable

Affecter une valeur constante à une variable est également une expression composée, faissant intervenir deux expressions de base (le nom de la variable et la constante numérique) et l'opérateur

2. Liste valable pour C89 et C99, qui a rajouté le mot-clef `restrict`. En revanche, C11 a introduit de nouveaux mots-clefs, comme `alignof`, `_Atomic` ou `_Generic` par exemple.

d'affection =. Par exemple, `x = 33` est une expression composée. Pour rappel, une expression doit être évaluable et typée. Dans le cas de l'affectation, le type du résultat sera le même que le type de la variable stockant la valeur. Quand à la valeur de l'expression, elle est égale à la valeur de ce qui est affecté (valeur d'une constante ou résultat d'une autre expression). En particulier la valeur de l'expression `x = 0` est 0 tandis que celle de l'expression `x = 1` est 1. Notez qu'il est habituel de combiner déclaration et affectation, par exemple : `int y = 2`


Dans le cas de variables statiques (cf section 4.4.4), la combinaison déclaration + affectation constitue même l'**unique moyen** de les initialiser.

Appel de fonction

Un appel de fonction (nous reverrons les fonctions plus en détail au chapitre 6) est également une expression composée, qui fait intervenir au minimum une expressions de base (le nom de la fonction) et l'opérateur unaire d'appel de fonction `()`.


Attention à ne pas confondre *appel* de fonction et *pointeur* de fonction (cf. section 8.4.3). Un appel de fonction est une expression composée tandis qu'un pointeur de fonction est juste une expression de base (le nom de la fonction).

3.2 Composition d'expressions plus complexes

Les instructions utilisent des expressions qui doivent être évaluées afin de retourner une valeur logique (booléenne) : vrai ou faux.


Il n'y a(vait) pas de booléens en C[3]. Une valeur nulle (0) est considérée comme logiquement fausse. Toute valeur non nulle est considérée comme logiquement vraie.

La composition d'expressions pour former des expressions logiques est possible avec :
- les opérateurs d'égalité `==` et de différence `!=`
- les opérateurs de comparaison `>`, `<`, `<=`, `>=`
- les opérateurs logiques : OU (`||`), ET (`&&`), négation (`!`)


Attention à ne pas confondre :
- l'opérateur d'égalité logique `==` avec l'opérateur d'affectation `=`
- l'opérateur OU logique `||` avec l'opérateur OU bit-à-bit `|`
- l'opérateur ET logique `&&` avec l'opérateur ET bit-à-bit `&`.

Exemples

```
x > 0 && x <= 100 || x < 0
x == 12 && y != 0
```

3.2.1 Cas des opérateurs d'affectation combinée

Ils prennent la forme suivante : *opérateur*=. Comme l'opérateur d'affectation simple (i.e =) leur associativité va de droite à gauche.

3. Jusqu'à C99, qui introduit le mot-clef `_Bool` mais cela ne change rien fondamentalement pour ce qui nous intéresse.

Exemples

```
x *= 2
y += (x + 3)
```

Cas particulier : si *opérateur* est l'addition ou la soustraction et que la valeur ajoutée ou retranchée est 1, alors on utilise plutôt les opérateurs ++ et -, mais qui sont beaucoup plus prioritaires (cf. section 3.2.4). De plus, la position de l'opérateur par rapport à celle de la variable ne produit pas le même résultat.

Quel sera l'affichage du code suivant ?

```c
#include <stdio.h>

int main(void)
{
  int a = 0;
  ++a;
  printf("Valeur de la variable a : %i\n",a);
  a++;
  printf("Valeur de la variable a : %i\n",a);
  printf("Valeur de la variable a : %i\n",a++);
  printf("Valeur de la variable a : %i\n",++a);
  return 0;
}
```

prepost.c

On parle de **pré-**incrémentation (resp. décrémentation) ou **post-**incrémentation (resp. décrémentation).

3.2.2 Cas des opérateurs bit-à-bit (*bitwise operators*)

Le langage fournit des opérateurs qui permettent de manipuler directement les bits d'une variable, mais **uniquement de type entier**, c'est-à-dire de type `char`, `short`, `int`, `long` et `long long` (signé ou non). Ces opérateurs sont les suivants :

&	ET bit-à-bit
|	OU bit-à-bit
^	OU exclusif bit-à-bit
<<	Décalage de bits vers la gauche
>>	Décalage de bits vers la droite
~	Complément à un (inversion de bits)

Attention à ne pas faire la confusion entre les versions logiques et les versions bit-à-bit des opérateurs ET et OU. En effet, examinez l'exemple suivant :

```c
#include <stdio.h>

int main(void)
{
```

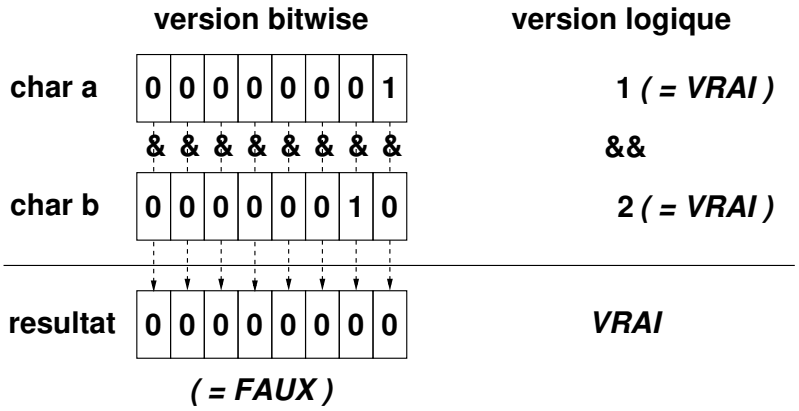
```
5   char a = 1, b = 2;
6   printf("Valeur bitwise : %hhi\n", a &  b );
7   printf("Valeur logique : %hhi\n", a && b );
8   return 0;
9 }
```

logique_vs_bitwise.c

L'affichage de ce programme sera :

```
Valeur bitwise : 0
Valeur logique : 1
```

Un dessin vaut mieux qu'un long paragraphe :



De la même façon `~x` (inverse bit-à-bit de `x`) est différent de `!x` (inversion logique, c'est-à-dire négation, de `x`).

Les opérateurs de décalage de bit, << et >>, sont indépendants de la représentation binaires des entiers (i.e *big endian* vs. *little endian*). Par exemple, << est une multiplication par 2 dans le cas *big endian* et un division par 2 dans le cas *little endian*. L'utilisation de ces opérateurs est portable du point de vue du langage mais le résultat est dépendant du système/processeur utilisé.

3.2.3 Cas de l'opérateur de succession

L'opérateur de succession (la virgule : ,) garantit une évaluation de ses opérandes de gauche à droite et la valeur de cette expression composée est celle de l'opérande de droite. Ainsi dans le cas de l'expression composée suivante :

$$expression_1 \quad , \quad expression_2$$

$expression_1$ est d'abord évaluée, puis $expression_2$. Le résultat de l'évaluation de cette expression composée est égal au résultat de l'évaluation de l'expression $expression_2$. Cet opérateur est surtout utilisé dans les boucles `for` (cf. section 5.2.2).

Les virgules que l'on trouve dans :
- les déclarations multiples de variables (cf. section 4.3.1) ;
- les énumérations de constantes (cf. section 4.2.3) ;
- les listes de paramètres de fonctions (cf. section 6.4) ;
- les liste de sélection générique (cf. section 4.1.5).

ne sont **pas** des opérateurs de succession.

3.2.4 Priorité des opérateurs

Quand une expression est composée à l'aide de multiples opérateurs, l'ordre d'évaluation va dépendre de leurs priorités respectives [4]. Le tableau 3.2 indique la priorité des opérateurs dans le sens décroissant (de haut vers le bas). L'associativité indique le sens dans lequel les expressions

TABLE 3.2: Tableau récapitulatif des opérateurs et de leurs priorités relatives en C

Opérateur(s)	Associativité	Commentaire
`()`	de gauche à droite	Forçage de la priorité

`()`
`[] -> .`
`++ --`
	de gauche à droite	Appel de fonction
		
		Opérateurs suffixés

`& * + - ~ !`
`sizeof`
`++ --`
`(type)`
	de droite à gauche	Opérateurs unaires
		
		Opérateurs préfixés
		Conversion de type (*cast*)

`* / %`	de gauche à droite	Opérateurs arithmétiques binaires
`+ -`	de gauche à droite	Opérateurs arithmétiques binaires
`<< >>`	de gauche à droite	Décalage de bits
`< <= > >=`	de gauche à droite	Opérateurs de comparaison
`== !=`	de gauche à droite	Opérateurs de comparaison
`&`	de gauche à droite	ET bit-à-bit
`^`	de gauche à droite	OU exclusif bit-à-bit
`|`	de gauche à droite	OU bit-à-bit
`&&`	de gauche à droite	ET logique
`||`	de gauche à droite	OU logique
`?:`	de gauche à droite	Opérateur ternaire
`= += -= *= /= %=` `&= |= ^= <<= =>>`	**de droite à gauche**	Opérateurs d'affectation
`,`	de gauche à droite	Succession (boucle `for`)

sont évaluées. Par exemple, si nous prenons l'expression composée

nom-de-variable = *expression*

alors l'expression à droite de l'opérateur d'affectation est d'abord évaluée (selon un sens qui dépend des opérateurs présents au sein de cette expression).

3.3 Instructions

Les instructions sont les éléments de base d'un code en langage C. Ce sont, pour ainsi dire, les "phrases" de C. Écrire des programmes corrects en C, cela revient à écrire des phrases correctes, à la fois syntaxiquement et grammaticalement. Ces instructions sont les éléments indispensables pour l'écriture d'un code-source C compilable en un programme exécutable sur un système. Les sections suivantes décrivent les différents types d'instructions.

4. Comme en mathématiques où la multiplication est prioritaire sur l'addition, ce qui fait que l'expression $1 + 2 \times 3$ peut être évaluée de façon univoque en 7 et non pas 9.

3.3.1 Instructions simples (dérivées d'expressions)

La façon la plus simple (ou naturelle) pour obtenir des instructions et de les dériver à partir d'expressions (on supposera que ces dernières sont correctement formées, c'est-à-dire syntaxiquement correctes). Pour créer une instruction simple il suffit de rajouter un caractère de terminaison à la fin d'une expression (simple ou composée). En C, ce caractère est le point-virgule : `;`
Par exemple, une expression de déclaration de variable peut être transformée en instruction ainsi :

`int toto = 33` $\Longrightarrow$ `int toto = 33 ;`

Il existe une instruction particulière, à savoir l'instruction qui ne fait rien (*no operation* ou nop) : il suffit juste de placer le caractère de terminaison d'instruction `;` Cette instruction vide est assez couramment utilisée dans les programmes [5].

3.3.2 Instructions groupées (blocs d'instructions)

Il est possible de regrouper des instructions afin de former des **blocs**. Ces blocs sont délimités par des accolades ouvrante `{` et fermante `}`. Il est inutile de placer un caractère de terminaison d'instructions (`;`) après l'accolade fermante de bloc. Ces blocs affectent la visibilité/portée des variables d'un programme : un identificateur déclaré à l'intérieur du bloc va masquer un autre identificateur identique mais déclaré à l'extérieur du bloc (cf. section 4.4.5).

3.3.3 Instructions de branchement

Les instructions de branchement prennent la forme suivante :

1. `if (` *expression* `)` *instruction*
2. `if (` *expression* `)` *instruction* `else` *instruction*
3. `switch (` *expression* `)` *instruction*

Nous reviendrons plus en détails sur ces instructions en section 5.1.

3.3.4 Instructions d'itération

Les instructions d'itération prennent la forme suivante :

1. `while (` *expression* `)` *instruction*
2. `do` *instruction* `while (` *expression* `) ;`
   Vous noterez la présence du terminateur d'instruction après la parenthèse fermante.
3. `for (` $expression_{optionnelle}$ `;` $expression_{optionnelle}$ `;` $expression_{optionnelle}$ `)` *instruction*
   Les expressions sont optionnelles, ce qui signifie qu'il est possible de mettre des instructions vides. Il s'agit là d'une des méthodes pour réaliser une boucle infinie (`for(;;)`)

Attention, une erreur classique consiste à remplacer les caractères de terminaison d'instruction (`;`) par l'opérateur de succession (`,`). Or, cela produit quelque chose de syntaxiquement correct mais qui ne fera sans doute pas ce que vous souhaitez [6] !

Nous reviendrons sur ces instructions en section 5.2

3.3.5 Instructions étiquetées

Il est possible de placer une **étiquette** (*label*) devant une ou des instructions. Cela permet de mieux contrôler le flot d'exécution du programme en conjonction avec les instructions de saut de la section suivante (et notamment le fameux `goto`). Ces instructions sont de la forme suivante :

5. Si, si.
6. Je viens de vous éviter de perdre deux heures de votre vie à chercher ce bug. Ne me remerciez pas, c'est cadeau.

1. *nom-de-label* : *instruction(s)*
2. `case` *expression-constante* : *instruction(s)*
3. `default` : *instruction(s)*

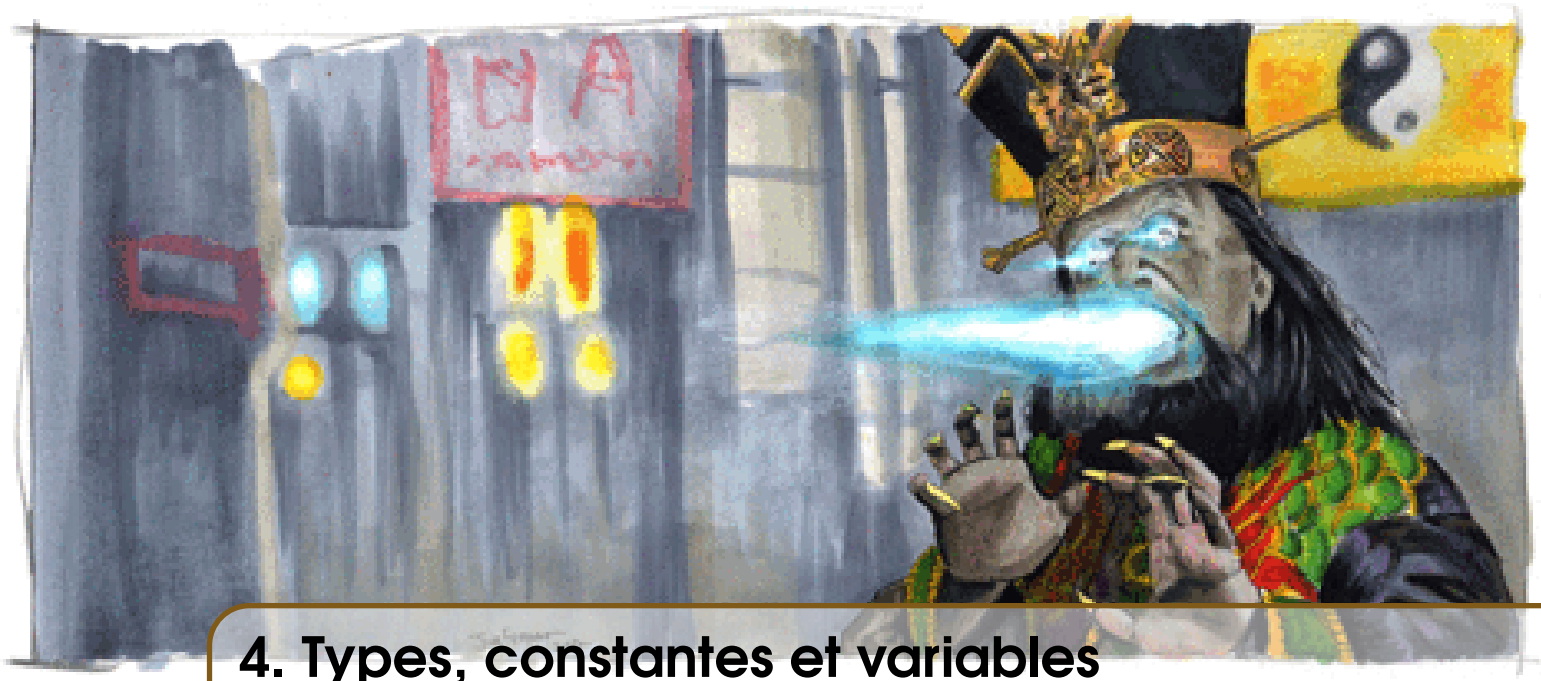
Une étiquette est donc soit l'un des mots-clefs `case` ou `default` ou bien un identificateur défini par l'utilisateur ou l'utilisatrice. Nous reviendrons sur ce mécanisme d'étiquetage en section 5.3.2.

3.3.6 Instructions de saut

Les instructions de saut sont les suivantes :

1. `goto` *identificateur* ;
   L'identificateur utilisé pour l'instruction `goto` doit obligatoirement correspondre à une étiquette qui est placée dans la *même* fonction où se trouve l'instruction de saut `goto`[7].
2. `continue` ;
   Cette instruction ne peut apparaître qu'à l'intérieur d'une instruction d'itération (plus exactement, à l'intérieur d'un bloc d'instructions dans une instruction itérative `for` ou `while`) et permet de passer à l'itération suivante.
3. `break` ;
   Cette instruction ne peut apparaître que dans une instruction d'itération ou bien dans une instruction `switch`. Elle termine l'instruction en cours (en général il s'agit plutôt d'un bloc d'instructions) et passe à la suivante.
4. `return` $expression_{optionnelle}$;
   Cette instruction de saut est celle qui vous allez utiliser le plus fréquemment car c'est celle qui permet de sortir d'une fonction. La fonction renvoie l'évaluation de l'expression qui suit le mot-clef `return`. Cette expression est optionnelle car il est possible de ne pas retourner de résultat : on parlera alors non pas de fonction mais de procédure (cf. Chapitre 6).

7. Il est possible d'effectuer des sauts non locaux, en sauvegardant et restaurant le contexte d'exécution à l'aide des fonctions `setjmp` et `longjmp`. L'utilisation de ces fonctions n'est cependant pas triviale.



4. Types, constantes et variables

4.1 Les types en C

Ainsi que la table 3.1 l'indique, un certain nombre de mots-clefs du langage désignent des types de base. Ces types peuvent se ranger en deux catégories : les types entiers et les types flottants. Ces types vont donc servir à caractériser des variables stockant des nombres dont la nature correspond à ces types. En C89 le type booléen n'existe pas, mais il existe en depuis C99. Dans les versions antérieures de C, on utilise donc une variable de type entier pour stocker une information booléenne.

4.1.1 Types de base

Types entiers

Le tableau suivant indique les types entiers de base existants en C :

Type	**Taille**	**Affichage avec** `printf`
`char`	plus petite unité adressable dans la machine	%c (%hhi pour valeur numérique i.e., code ASCII du caractère)
`short int` / `short`	au moins 16 bits	%hi %hd
`int`	au moins 16 bits	%i %d
`long int` / `long`	au moins 32 bits	%li %ld
`long long int` / `long long` (depuis C99)	au moins 64 bits	%lli %lld

Tous ces types sont des versions signées par défaut (sauf `char`) et des versions non signées existent également (il faut mettre le mot-clef `unsigned` en tête de déclaration et remplacer le `i` ou le `d` par un `u` pour l'affichage).

Les tailles en bits des différents types varient selon le **modèle de données**. Les modèles existants sont les suivants :

Modèle	`short int`	`int`	`long int`	`long long int`	`size_t`, pointeurs
LLP64	16	32	32	64	64
LP64	16	32	64	64	64
ILP64	16	64	64	64	64
SILP64	64	64	64	64	64

Il est très probable que les systèmes que vous allez utiliser auront pour modèle LP64.

Types flottants

Le tableau suivant indique les types flottants de base existants en C :

Type	Taille	Affichage avec `printf`
`float`	32 bits le plus souvent	%f %g %e %a %F %G %E %A
`double`	64 bits le plus souvent	%lf %lg %le %la %lF %lG %lE %lA
`long double`	au moins un `double`	%Lf %Lg %Le %La %LF %LG %LE %LA

`f` ou `F` affiche en notation déciamle, `e` ou `E` affiche en notation scientifique (avec des exposants) et `g` ou `G` choisit l'affichage le plus adapté. `a` ou `A` affiche en hexadécimal. Ces types n'existent pas en version non signée.

En raison d'arrondis, l'opérateur de comparaison `==` n'est pas utilisable avec des nombres flottants : `if (a == b)` $\Longrightarrow$ `if (fabs(a - b) <` *seuil* `)`

Rencontre du troisième type : le cas `void`

Il existe un autre type de base, un peu particulier : le type `void`. Ce type n'est pas un véritable type de données et il n'est pas possible de déclarer des variables de ce type. `void` est utilisé pour les types de retour de certaines fonctions (cf. section 6.3.2) ou pour déclarer des pointeurs génériques (cf. section 8.1.3). De plus, ce type a une taille nulle (i.e `sizeof(void)`$= 0$).

4.1.2 Types définis par l'utilisateur

Outre ces types de base, il est possible pour l'utilisateur/utilisatrice de définir de nouveaux noms de types et de nouveaux types également.

Nouveaux noms de types

C'est le mot-clef `typedef` qui est employé pour définir un nouveau nom de type :

`typedef` *ancien-nom-de-type nouveau-nom-de-type*

Il est assez courant (mais pas obligatoire) de terminer le nouveau nom avec `_t` car cela permet de bien fixer le fait que ce nom désigne un type de données. De plus, les éditeurs de texte sont capables de détecter ce suffixe et de mettre ce nom avec le bon code couleur.

Nouveaux types de données

Pour créer de nouveaux types de données, les mots-clefs `struct` et `union` sont utilisables. Nous verrons cela plus en détails dans le chapitre 9 qui leur est consacré.

4.1.3 Conversions de types

Des conversions de types se produisent quand des expressions utilisent des types hétérogènes pour les constantes, variables ou résultats d'évaluation de sous-expressions. Comme les expressions sont elles-mêmes typées (du moins le résultat de leur évaluation), il faut homogénéiser tous ces élements en effectuant ces **conversions de types** .

Conversion automatique

Le langage C dispose de règles implictes (automatiques) de conversion qui évitent au programmeur ou à la programmeuse de devoir effectuer lui-même ou elle-même ces conversions explicitement. En général, le "petit" type (i.e celui qui occupe le moins de bits) est converti en le "gros" type (i.e celui qui occupe le plus de bits). Dans le cas de types signés, les règles sont les suivantes :

— D'abord Si l'une des opérandes est de type `long double`, alors l'autre opérande est convertie en `long double` ;
— Ensuite, si l'une des opérandes est de type `double`, alors l'autre opérande est convertie en `double` ;
— Ensuite, si l'une des opérandes est de type `float`, alors l'autre opérande est convertie en `float` ;
— Ensuite, convertir les types `char` et `short` en `int` ;
— Enfin, si l'une des opérandes est de type `long`, alors l'autre opérande est convertie en `long`.

Dans le cas non signé, le résultat des conversions dépend de l'architecture du processeur utilisé.

Conversion explicite

Si l'on doit effectuer soi-même une conversion (par exemple pour forcer le type d'un argument de fonction) il faut utiliser l'opérateur unaire de conversion (opération *cast*) `()` :

(*nom-de-type*) *expression*

Exemple de conversion explicite :

```
int   x = 23;                /* x est de type int          */
float y = (float) x + 2.34 ; /* calcul flottant utilisant
                                la variable entiere x      */
```

Une opération de conversion ne modifie pas la valeur d'une variable mais seulement son type. Il ne s'agit pas d'une opération de conversion de données, d'un format vers un autre (par exemple, convertir la chaîne de caractères "123" en la valeur numérique 123).

4.1.4 ++ Qualificateurs de types

Les mots-clefs `const`, `restrict` et `volatile` sont des qualificateurs de type , qui sont susceptibles de modifier le comportement des objets déclarés avec un el type qualifié. Qualifier un type sert souvent à donner des indications au compilateur pour qu'il procède à des optimisations (ou non) lors de la génération de code. Le rôle de ces qualificateurs est le suivant :

— `const` : une variable qualifiée avec ce mot-clef peut être initialisée au moment de sa déclaration, mais n'est plus modifiable par la suite. Ce qualificateur est souvent employé pour des arguments de fonction afin d'en expliciter un peu plus le rôle (cf. section 8.4.4) ;
— `restrict` : ce qualificateur n'est utilisable que pour des pointeurs et indique que le pointeur qualifié ne possède pas d'*alias*, c'est-à-dire qu'il n'existe pas un autre pointeur ayant pour valeur la même adresse ;


Il vous incombe en tant que programmeur ou programmeuse de garantir cette absence d'aliasing. Le système et le compilateur ne le feront pas à votre place.

— `volatile` : une variable dont le type est qualifié par `volatile` est susceptible d'être modifiée par un évènement extérieur au programme et doit être lue en mémoire à chaque accès par le programme. L'utilisation de ce mot-clef permet de désactiver les optimisations (parfois agressives) du compilateur.


Nous allons utiliser principalement les types entiers dans ce cours, et réserver l'utilisation des flottants pour certains cas particuliers. Par défaut, vous pouvez donc déclarer vos variables comme étant du type `int`.

4.1.5 ++ Sélection générique (par le type)

L'opération de sélection générique permet d'effectuer une sélection d'expression sur la base du type d'une autre expression passée en argument à l'opérateur. Il devient ainsi possible d'avoir des actions différenciées sur cette base de typage. Il s'agit d'une fonctionnalité introduite depuis C11 et qui n'existait pas du tout auparavant [1]. Travailler sur/déterminer le type d'une variable offre des possbilités intéressantes pour produire un code plus générique et gagner en abstraction. Le nouveau mot-clef introduit est `_Generic` et la syntaxe est la suivante :

`_Generic` (*expression-d-affectation* , *liste-d-associations-generiques*)

La liste des associations génériques prend la forme suivante :

identificateur-type$_1$: *expression-d-affectation* ,
identificateur-type$_2$: *expression-d-affectation* ,
`default`$_{optionnelle}$: *expression-d-affectation*

Tous les identificateurs (noms) de types doivent être distincts et correspondre à des types de base, des types construits par l'utilisateur avec des structures (cf. Chapitre 9) ou des pointeurs sur ces types. Une expression d'affection peut être un nom de variable ou un nom de fonction. Il ne peut y avoir qu'une seule clause `default`, mais elle n'est pas obligatoire. Dans ce cas, le type de l'expression d'affecation prise en argument par la sélection doit avoir un type compatible avec exactement un type dans la liste d'association. De plus, cette expression prise en argument n'est pas évaluée, seule l'expression sélectionnée sur la base du type l'est. Voici un exemple qui permet d'afficher une chaîne de caractères correspondant au nom et au type de différentes variables :

```
1  #include <stdio.h>
2  #include <stddef.h>
3
4  #define typename(X) _Generic((X),                \
5      char          : #X " is char \n",            \
6      int           : #X " is int \n",             \
7      unsigned int  : #X " is uint \n",            \
8      char *        : #X " is char_ptr \n",        \
9      int *         : #X " is int_ptr \n",         \
10     unsigned int *: #X " is uint_ptr \n",        \
11     void *        : #X " is generic_ptr \n", \
12     default       : #X " is unknown \n")
```

1. Il y avait une extension GNU `typeof` de disponible avec gcc, mais non standard.

```
13
14 int main(void)
15 {
16   int truc = 33;
17   int *bidule = &truc;
18   void *machin = NULL;
19   float riri = 1.0;
20   char fifi = 'a';
21   float *loulou = &riri;
22
23   printf(typename(truc));
24   printf(typename(bidule));
25   printf(typename(machin));
26   printf(typename(riri));
27   printf(typename(fifi));
28   printf(typename(loulou));
29
30   return 0;
31 }
```

generic_type.c

Ce programme affiche :

```
truc is int
bidule is int_ptr
machin is generic_ptr
riri is unknown
fifi is char
loulou is unknown
```

Cet autre exemple montre comment il est possible d'implémenter une fonction générique :

```
#define fonction_polymorphe(X) _Generic((X),
                                    long double: fonction_lb,          \
                                    float      : fonction_f,           \
                                    default    : fonction_fallback) (X)
```

Selon le type de l'expression passée en argument à la macro (cf. Section 10.2) `fonction_polymorphe`, la fonction effectivement appelée sera différente.

4.2 Constantes

Dans le chapitre 3, nous avons vu que les constantes font partie des expressions. cela recouvre à la fois les constantes numériques, les caractères constants (qui sont déclarés entre *simple quotes* ' et ') et les chaînes de caractères constantes (qui sont déclarées entre *double quotes* " et ").

Deux *simple quotes* ne forment pas une *double quote* !

Il est usuel de déclarer les constantes au début du fichier source, après les diverses directives `#include`.

4.2.1 Typage

Si le type des constantes caractère et chaîne de caractères est évident, il est possible de typer les constantes numériques ; par défaut, une constante entière est considérée de type `int` (entier signé). Pour utiliser une constante entière non signée (`unsigned int`), il faut rajouter la lettre `u` ou `U` en fin de constante. Pour utiliser une constante de type `long`, il faut rajouter la lettre `l` ou `L` en fin de constante. Par exemple, si `1234` est un entier signé, `1234L` est un entier long, `1234U` est un entier non signé et `1234UL` est un entier long non signé.

Par défaut, les constantes entières sont décimales. Il est possible d'écrire des constantes octales et hexadécimales : toute constance préfixée avec un zéro `0` est octale (pas de chiffres `8` ni `9` ne doivent être présents) tandis qu'une constante hexadécimale est préfixée avec `0x` ou `0X`. Dans ce dernier cas, les chiffres possibles sont `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `a`, `b`, `c`, `d`, `e` et `f`

Les constantes flottantes (qui contiennent le caractère `.` pour marquer la virgule) possèdent par défaut le type `double`. Pour avoir des constantes de type `float`, il faut ajouter le suffixe `f` ou `F`. Le suffixe `l` ou `L` permet d'obtenir des constantes de type `long double`.

Exemples

`1234` $\Longrightarrow$ constante entière signée (valeur : 1234)
`010` $\Longrightarrow$ constante entière signée en octal (valeur : 8)
`0x23` $\Longrightarrow$ constante entière signée en hexadécimal (valeur : 35)
`34.0` $\Longrightarrow$ constante flottante signée (valeur : 34)

4.2.2 Nommage

Plutôt que d'utiliser des constantes directement dans le code, il est possible de les nommer et d'utiliser ce nom à la place. L'avantage de cette approche réside dans la meilleure maintenabilité du code produit. En effet, en cas de modification de la valeur de la constante, vous n'aurez à changer cette valeur qu'à l'endroit du nommage plutôt que dans l'ensemble du code, ce qui pourrait par ailleurs entraîner des erreurs en cas d'oubli de changement d'une valeur. La directive de *preprocessing* `#define` permet de déclarer une constante nommée, par exemple :

```
#define BIDULE 123456   /* définition d'une constante entière   */
#define TEXTE  "coucou" /* définition d'une chaîne constante    */
#define PI     3.14159  /* définition d'une constante flottante */
```

La substitution du nom par la valeur associée est réalisée lors de la phase de *preprocessing* (cf. section 10) et le code source compilé est celui où les noms ont déjà été remplacés par les valeurs.

Il est usuel de nommer les constantes à l'aide de majuscules uniquement. Ce n'est pas une obligation, mais une habitude de programmation très suivie par les développeurs et développeuses d'applications en C.

4.2.3 Énumération

Le langage C offre la possibilité de construire des énumérations de constantes avec le mot-clef `enum`. Une énumération est une liste de constantes entières, et sa déclaration se fait de la façon suivante :

`enum` *nom-enumeration* `{` *liste-de-constantes* `}` `;`

Par exemple, il est possible de définir une sorte de type booléen de cette façon :

```
enum booleen { FAUX, VRAI } ;
```

Si des valeurs ne sont pas spécifiées, le premier nom dans l'énumération vaut 0, le suivant 1 et ainsi de suite. L'exemple précédent est équivalent à :

```
enum booleen { FAUX = 0, VRAI = 1} ;
```

Il est aussi possible de définir uniquement le premier nom et tous les autres auront des valeurs calculées par le compilateur. Par exemple :

```
enum couleur { VERT = 2, BLEU, ROUGE, JAUNE } ;
```

Dans ce cas, les valeurs des constantes `BLEU`, `ROUGE` et `JAUNE` seront 3, 4 et 5 respectivement.

L'énumération est une alternative à la déclaration à l'aide de la directive `#define`, avec l'avantage que les valeurs sont générées automatiquement par le compilateur. Il est possible de déclarer des variables de type `enum` mais le compilateur n'effectue pas de vérification que les valeurs stockées dans ces variables sont cohérentes avec les valeurs des constantes déclarées, comme le montre le code du programme `enum.c` :

```c
#include<stdio.h>

enum boolean { FALSE, TRUE } ; /* FALSE vaut 0,
                                  TRUE  vaut 1 */
int main(void)
{
  enum boolean x;     /* x est une variable de type
                         enum boolean donc sa valeur
                         devrait etre soit 1 soit 0 */
  x = 2;              /* assignation licite */
  printf("valeur de x: %i\n",x);

  return 0;
}
```

enum.c

4.3 Notion de variable

Ce document a déjà mentionné à de nombreuses reprises le terme de *variable*. Cette section va nous donner l'opportunité de préciser ce qu'est une variable dans le cas du langage C, de faire la liste de ses propriétés et de les expliciter.

Définition

En C, une **variable** désigne un **moyen d'accéder à un emplacement mémoire contenant des données**. Une variable possède un certain nombre de propriétés, qui sont détaillées dans les sections suivantes. Afin de pouvoir être utilisée effectivement dans un programme, une variable doit être **déclarée** .

4.3.1 Déclaration

La déclaration a pour effet de déterminer les propriétés de la variable déclarée. Une variable ne peut pas être utilisée (*load* ou *store*) dans le code tant qu'elle n'est pas déclarée (cf. la notion de

visibilité ci-dessous). Ainsi que nous l'avons déjà vu dans la section consacrée aux expressions composées (cf. section 3.1.3), une instruction de déclaration possède la forme suivante :

nom-de-type *liste-de-noms-de-variables* ;

Exemples

```
int   ma_variable;  /* declaration d'une variable
                       entiere non initialisee */
char  truc = 'a';   /* declaration d'une variable
                       caractere initialisee avec 'a' */
float riri, fifi, loulou; /* declaration de trois
                             flottants non initialises */
```

Par défaut, les variables ne sont **pas** automatiquement initialisées en C (avec la valeur 0, par exemple). Il est possible qu'un compilateur particulier procède à une telle initialisation, mais ce comportement n'est pas standard et n'est donc pas portable.

4.3.2 Catégories

L'endroit dans le code où une variable est déclarée a son importance puisque qu'il détermine si la variable sera catégorisée en tant que variable **locale** ou variable **globale**. Une variable déclarée en dehors de tout corps de fonction est une variable globale au programme. Dans le cas contraire, la variable est considérée comme locale au bloc d'instructions où elle est déclarée (*a minima* un corps de fonction, mais pas forcément). C89 impose de déclarer les variables en **début de bloc**, avant tout autre type d'instructions. Depuis C99, cette contrainte est relâchée, et il est permis donc de déclarer des variables à la volée, c'est-à-dire n'importe où dans un bloc d'instructions. Bien entendu, il n'est toujours pas possible d'utiliser le même identificateur (nom) à l'intérieur d'un même bloc d'instructions (un corps de fonction en particulier).

Afin de minimiser les erreurs, je vous **impose de déclarer vos variables en début de bloc** [2] ! Il vous faudra donc (un peu) réfléchir à ce que vous faites quand vous allez coder.

De façon générale, il vaut mieux minimiser le nombre de variables globales utilisées dans un programme. En effet, la présence de variables globales complexifie la mise au point de code réentrant. Nous verrons des méthodes permettant d'éviter le recours à des variables globales. Il n'est cependant pas toujours possible de se passer de variables globales *dans tous les cas*.

4.4 Propriétés des variables

En plus du caractère de localité evoqué ci-dessus, les variables possèdent les propriétés suivantes :

1. un identificateur (nom) ;
2. un type ;
3. une valeur ;
4. un emplacement en mémoire ;
5. une visibilité (ou portée *scope*) ;
6. une durée de vie.

2. Ce sera même la première des règles que je vous impose : allez faire un tour page 135.

4.4.1 Identificateur (nom)

L'identification d'une variable suit quelques règles : le premier caractère du nom **doit** être une lettre (le caractère _ comptant comme une lettre dans ce cas). Les caractères suivants peuvent être des lettres, des chiffres ou bien _. Le compilateur fait la différence entre majuscules et minuscules [3]. Il est recommandé d'utiliser des noms explicites et d'une longueur raisonnable. Il est déconseillé d'utiliser des noms de variables avec uniquement des majuscules, car ce sont souvent les constantes qui sont nommées avec des majuscules ;

4.4.2 Type

Rappel : en C, une variable est **forcément** typée. En effet, le compilateur doit être capable de calculer la taille mémoire occupée par une variable. Or, cette taille est conditionnée par le type utilisé. Si, pendant l'analyse du code, le compilateur se trouve incapable de déterminer une taille, il générera une erreur. Ce type est indiqué au moment de la déclaration de la variable et ne pourra plus être modifié par la suite (effectuer une conversion explicite (cf. section 4.1.3) ne change pas le type de la variable) : en effet, le typage en C est statique. Les unions de types (cf. section 9.2) vont permettre de lever partiellement cette contrainte.

4.4.3 Valeur

La valeur d'une variable peut éventuellement dépendre de son type. En effet, comme la spécification d'un type induit une taille de stockage (un certain nombre de bits), les valeurs possibles sont donc limitées. Par exemple, dans le cas d'une variable de type entier `int`, elle aura une valeur comprise entre `INT_MIN` et`INT_MAX`. De plus, une conversion de type peut entraîner une modification de la valeur (attention en particulier à la conversion d'un type `unsigned` vers un type `signed`). Par ailleurs, je vous rappelle qu'en C, les variables ne sont pas initialisées par défaut.

4.4.4 Emplacement mémoire

Une variable est un moyen d'accéder à un emplacement dans la mémoire du programme (et *in fine* de la machine). Ainsi que nous l'avons déjà évoqué dans la remarque de la section 1.1, le schéma d'adressage de la mémoire du programme est appelé adressage virtuel tandis que le schéma d'adressage de la mémoire de la machine est appelé adressage physique. Nous verrons plus tard comment est effectuée la correspondance (la traduction) entre ces deux schémas d'adressage. Dans le cas d'un programme en C, les adresses des variables sont des adresses virtuelles (également appelées linéaires).

Zones de mémoire d'un programme

La mémoire virtuelle d'un programme est organisée en différentes zones qui effectuent un découpage logique du programme. Selon les propriétés des variables, la zone de stockage est appelée à changer. La Figure 4.1 montre cette organisation en zones, dans le cas d'un programme s'exécutant sur un système de type UNIX. En ce qui concerne les variables, ce sont les zones de pile (*stack*) et de données (*data*) qui sont utilisées pour leur stockage.

C'est la **classe de stockage** de la variable qui détermine sa zone de stockage. Cette classe est choisie avec la catégorie de la variable (locale ou globale) et avec l'utilisation d'un mot-clef idoine, choisi parmi : `auto`, `register`, `static` et `extern`. Ainsi, une variable globale (qui par défaut n'est pas de classe `static`), sera **systématiquement** stockée dans la zone de données du programme, qu'elle soit de classe `static` ou non. La classe `extern` n'est utilisable que pour les variables globales et indique juste que le programme va utiliser cette variable globale, mais qu'**elle est réellement déclarée dans un autre fichier source**. Si cette variable globale déclarée dans

3. Une source d'erreurs potentielle !

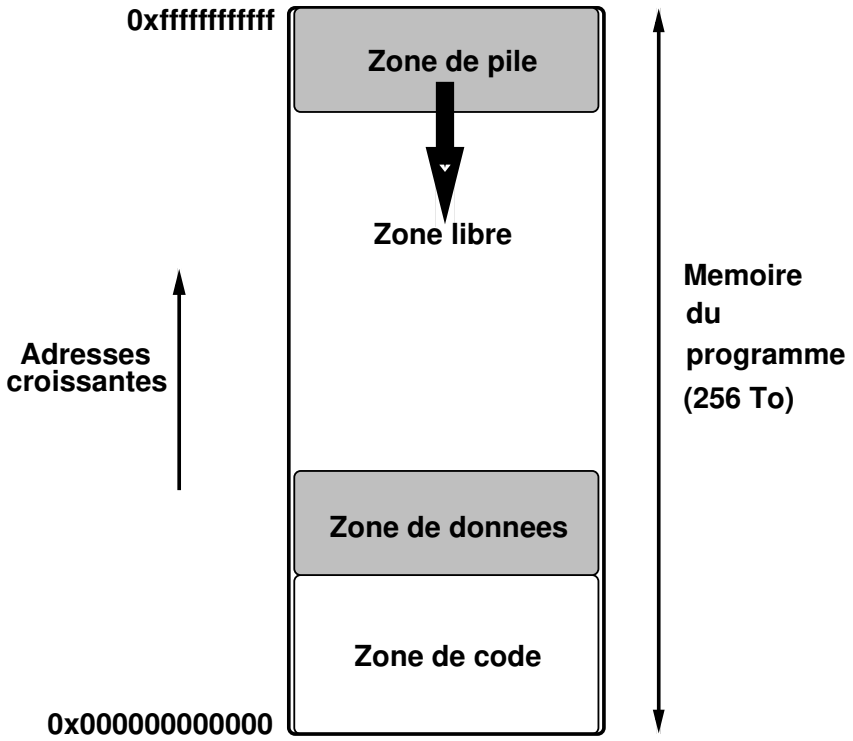


FIGURE 4.1: Organisation de la mémoire d'un programme en C (les tailles des différentes zones ne sont pas à l'échelle).

l'autre fichier est de classe `static`, alors le compilateur ne la trouvera pas et indiquera que le fichier courant référence une variable inconnue.

Quant aux variables locales, elles sont par défaut de classe automatique (mot-clef `auto`)[4] et stockées dans la zone de pile du programme. "Automatique" signifie que la mémoire stockant la variable est :

— automatiquement allouée au moment de la déclaration de la variable et
— automatiquement libérée quand le programme sort du bloc d'instructions où cette variable est déclarée.

La figure 4.2 montre le fonctionnement simplifié de la pile du programme dans le cas où la fonction `main` appelle une fonction `f` qui appelle elle-même une fonction `g` (voir le chapitre 6 pour plus d'informations sur les fonctions).

Une variable locale de classe `register` est similaire à une variable locale automatique, sauf que ce mot-clef indique qu'elle sera fréquemment accédée. Le compilateur va donc *essayer* de stocker cette variable dans un registre du processeur afin d'en accélérer les accès. Le mot-clef `register` est utilisable uniquement avec des variables automatiques et des arguments de fonctions. Une variable locale de classe statique (mot-clef `static`) est quant à elle stockée dans la zone données du programme.

Rappel : l'unique moyen d'initialiser une variable statique est de le faire au moment de sa déclaration (cf. section 3.1.3). Ainsi les codes suivants ne sont **pas** équivalents :

```
static int toto = 0;
```

vs.

```
static int toto;
toto = 0;
```

4. C'est pour cela que le mot-clef `auto` n'est jamais utilisé.

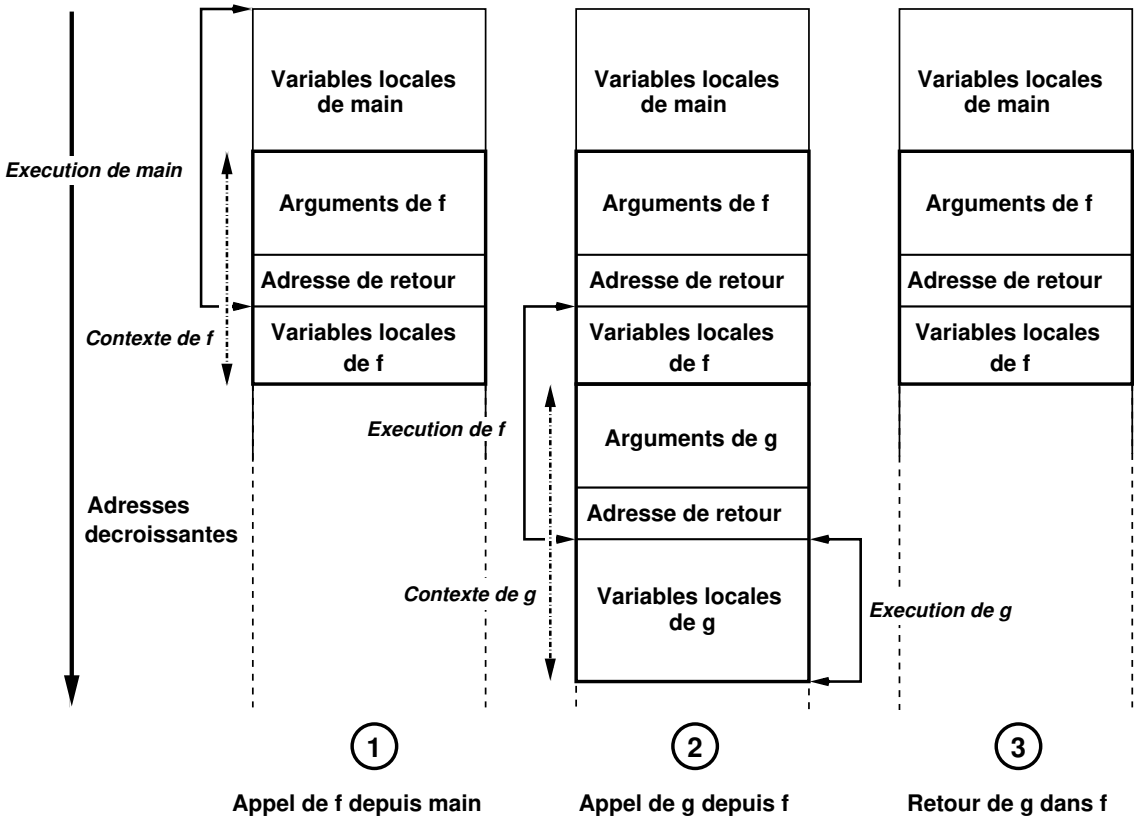


FIGURE 4.2: Organisation de la pile d'exécution d'un programme

Dans le premier cas, l'initialisation est effectuée **statiquement**, c'est-à-dire à la compilation tandis que dans le second cas, l'affectation est faite **dynamiquement**, c'est-à-dire au moment de l'exécution du programme. Dans ce cas, il ne s'agit pas d'une initialisation à proprement parler, mais une affectation, ce qui est un peu différent.

Taille de l'espace de stockage d'une variable

Ainsi que nous l'avons vu avec les types (cf. section 4.1), l'empreinte mémoire d'une variable dépend de son type et du système/processeur utilisé (par exemple, un entier (de type `int`) peut être stocké sur 32 ou 64 bits selon le modèle, cf). Il est possible de connaître la taille occupée par une variable ou un type de données avec l'opérateur `sizeof` [5]. Cet opérateur s'utilise ainsi :

`sizeof(` *nom-de-variable* `)` ou
`sizeof(` *nom-de-type* `)`


Afin d'éviter certaines erreurs malheureuses, je vous conseille d'utiliser exclusivement des noms de types avec l'opérateur `sizeof`.


Rappel : une déclaration de variable est correcte (du point de vue du langage) si le compilateur est capable de calculer son occupation en mémoire.

5. Contrairement à ce que l'on pourrait penser, `sizeof` n'est pas une fonction, mais bien un opérateur, cf. la liste des mots-clefs du langage, table 3.1.

4.4.5 Visibilité/portée

La visibilité (ou portée (*scope*)) détermine les endroits dans le code-source où une variable est effectivement utilisable (i.e accessible en lecture ou écriture). Une variable est visible **à partir de son point de déclaration** et prend fin :

— dans le cas d'une variable globale, cette visibilité est effective jusqu'à la fin du fichier source où la variable est déclarée [6] ;
— dans le cas d'une variable locale, la visibilité est effective jusqu'à la fin du bloc d'instructions où cette variable est déclarée.

Principe de masquage de variable

Pratiquement, une variable est visible dans sa portée, sauf là où elle est **masquée** par une autre variable **plus locale et possédant le même identificateur**.

Premier exemple :

```c
#include <stdio.h>

int x = 0;   /* variable globale  */

int main(void)
{
  int x = 2; /* variable locale a la fonction main*/
  printf("Valeur de x : %i\n",x);

  return 0;
}
```

masquage1.c

Ce programme affichera `Valeur de x : 2` car la variable globale `x` est masquée par la variable locale de même nom déclarée localement dans la fonction `main`.

Second exemple :

```c
#include <stdio.h>

int main(void)
{
  int x = 3;  /* variable locale a la fonction main */
  { /* debut d'un nouveau bloc d'instructions */
    int x = -1; /* variable locale au bloc  */
    printf("Valeur de x : %i\n", x);
  } /* fin bloc d'instructions */
```

6. Une telle variable globale est éventuellement visible dans des fichiers sources extérieurs si elle n'est pas de classe statique et si le mot-clef `extern` est utilisé.

```
10
11    return 0;
12  }
```

masquage2.c

Ce programme affichera `Valeur de x : -1` car la variable la plus globale (celle déclarée au début du bloc formant le corps de la fonction `main`) est masquée par la variable de même nom et déclarée dans le bloc d'instructions le plus interne contenu dans le corps de la fonction (qui est plus local, donc).

4.4.6 Durée de vie

La durée de vie correspond à l'intervalle de temps où une variable existe, c'est-à-dire réside en mémoire et est accessible (lecture ou écriture). Cette durée de vie varie selon la catégorie de la variable (locale ou globale) et sa classe de stockage (mot-clef `static` essentiellement).

— La durée de vie d'une variable globale (statique ou non) est égale à la durée d'exécution du programme lui-même.

— La durée de vie d'une variable locale statique (mot-clef `static`) est égale à la durée d'exécution du programme, ce qui signifie qu'à chaque exécution du bloc d'instructions où la variable est déclarée, on ne réutilise pas une nouvelle variable mais bien celle de départ avec pour valeur la dernière affectée. Un exemple est donné avec le code du programme `static.c` :

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int i = 0;
6    for(i = 0 ; i < 10 ; i++)
7      {
8        static int x = 0;
9        ++x;
10       printf("Valeur de x : %i\n",x);
11     }
12   return 0;
13 }
```

static.c

Ce programme affichera : `valeur de x : 1`, `valeur de x : 2`, ..., `valeur de x : 10` lors de son exécution.

— La durée de vie d'une variable locale automatique est égale à la durée d'exécution du bloc d'instructions au sein duquel la variable est déclarée. Le code du programme `automatic.c` montre ce comportement :

```
1  #include <stdio.h>
2
3  int main(void)
4  {
```

```
5   int i = 0;
6   for(i = 0 ; i < 10 ; i++)
7     {
8       int x = 0;
9       ++x;
10      printf("Valeur de x : %i\n",x);
11    }
12  return 0;
13 }
```

automatic.c

Le programme affichera 10 fois la valeur 1 pour la variable `x`.

4.4.7 Synthèse

La table 4.1 récapitule et résume ce qui est utile de connaître concernant les variables en termes de propriétés. Par défaut, une variable locale est de classe automatique (c'est la raison pour laquelle le mot-clef n'est presque jamais utilisé) tandis qu'une variable globale est non-statique.

TABLE 4.1: Comparaison des propriétés des variables

Catégorie	Classe de stockage	Visibilité*	Durée de vie	Emplacement
variable **globale**	statique	fichier source de déclaration	programme entier	zone de données du programme
	non-statique (par défaut)	autres fichiers sources (via `extern`)		
variable **locale**	statique	bloc d'instructions de déclaration	programme entier	zone de données du programme
	automatique (par défaut)		bloc d'instructions de déclaration	zone de pile du programme

* Une variable est visible à partir de son point de déclaration. Cette visibilité est éventuellement modifiée par les effets de masquage induits par la déclaration/présence de variables plus locales.

5. Contrôle du flot d'exécution

Les instructions de contrôle de flot d'exécution permettent de modifier l'ordre dans lequel les calculs sont effectués. Dans ce chapitre, nous allons examiner plus en détails deux types de constructions syntaxiques fondamentales en programmation :

— les constructions conditionnelles qui permettent d'introduire des embranchements dans le code. Le choix d'une branche ou d'une autre est basé sur le résultat de l'évaluation d'une expression ;

— les constructions itératives (les boucles) qui permettent de répéter l'exécution d'instructions en vue d'écrire des codes plus compacts [1].

5.1 Instructions de branchement

Ces instructions ont déjà été rapidement vues en section 3.3. Nous détaillons leur syntaxe, fonctionnement et utilisation dans cette section.

5.1.1 construction `if ...else...`

La première construction conditionnelle utilise le mot-clef `if` (obligatoirement) et éventuellement le mot-clef `else`. La syntaxe formelle est la suivante :

`if` (*expression*)
instruction

Le fonctionnement est le suivant : *expression* est évaluée de gauche à droite et cette évaluation s'arrête quand la véracité ou la fausseté de l'expression peut être déterminée avec certitude. Si la valeur de l'expression est vraie, alors *instruction* est exécutée.

Dès que la fausseté de l'expression est trouvée son évaluation prend fin. Il peut donc arriver qu'*expression* ne soit pas entièrement évaluée : attention donc aux

1. Règle de bon sens : moins vous écrirez de lignes de codes, moins vous produirez d'erreurs.

sous-expressions réalisant des effets de bord !

Par ailleurs, veuillez noter que :

— *expression* peut être une expression simple, mais sera le plus souvent une expression composée ;

— *instruction* peut être n'importe quel type d'instruction, pas uniquement une instruction simple. En particulier cela peut être un bloc d'instructions.

Je vous recommande de créer systématiquement un bloc d'instructions, même en cas d'instruction simple, en entourant votre instruction avec des accolades (`{` et `}`). Un bloc d'instructions ne se créé pas avec de l'indentation car C n'y est pas sensible. Une erreur courante consiste donc à écrire :

```
if (expression)
   instruction1
   instruction2
   instruction3
```

Mais dans cas, si *expression* est évaluée à "vrai", alors seule l'*instruction*$_1$ sera exécutée. *instruction*$_2$ et *instruction*$_3$ seront exécutées de toute façon, quel que soit le résultat de l'évaluation d'*expression*. Faites la comparaison avec la situation suivante :

```
if (expression)
 {
   instruction1
   instruction2
   instruction3
 }
```

Dans ce cas, si *expression* est évaluée à "vrai", alors les trois instructions du bloc suivant seront exécutées car elles appartiennent au bloc dépendant du `if`.

Afin de créer un véritable embranchement, c'est-à-dire indiquer l'instruction qui sera exécutée si jamais l'expression est évaluée à "faux", le mot-clef `else` est utilisé de la façon suivante :

```
if (expression)
   instruction1
else
   instruction2
```

À nouveau, *instruction*$_1$ et *instruction*$_2$ peuvent être n'importe quel type d'instruction, y compris des blocs d'instructions. La même recommandation que précédemment s'applique aussi dans ce cas. Enfin, un `else` est toujours associé au `if` le plus proche.

Voici un morceau de code volontairement non-indenté :

```
if ( n > 0 )
if ( a > b )
z = a ;
else
z = b;
```

Quelle est la valeur de `z` dans les cas suivants ?

1. `n` vaut 1, `a` vaut 2 et `b` vaut 3
2. `n` vaut 0, `a` vaut 2 et `b` vaut 3
3. `n` vaut 0, `a` vaut 3 et `b` vaut 2

Une erreur classique consiste à confondre l'opérateur de comparaison logique `==` avec l'opérateur d'affectation `=` dans les tests logiques. Or, une affectation est une expression qui renvoie une valeur, qui est la valeur affectée. Donc le test `if (x = 1)` sera toujours vrai et le test `if(x = 0)` sera toujours faux, ce qui n'est pas le cas de `if (x == 1)` ou `if (x == 0)`. Afin d'éviter ce genre de problème, je vous recommande d'inverser la variable et la constante dans le test :

```
if (const == variable)
```

plutôt que

```
if (variable == const)
```

En cas de confusion entre les opérateurs `==` et `=`, l'expression sera une affectation d'une variable dans une constante, ce qui est impossible et produira une erreur à la compilation [2].

Étant donné qu'*instruction*$_2$ est potentiellement une instruction conditionnelle, il est possible de "cascader"les `if` de cette façon :

`if (`*expression*$_1$`)`
  instruction$_1$
`else if (`*expression*$_2$`)`
  instruction$_2$
`else if (`*expression*$_3$`)`
  instruction$_3$
`else`$_{optionnel}$
  instruction$_{4-optionnel}$

Les différentes expressions sont évaluées dans l'ordre. Dès qu'une expression est évaluée à "vrai", alors l'instruction associée est exécutée et l'instruction conditionnelle globale (la casacade des `if`) prend fin.

Écrivez un programme qui affiche si une année (stockée dans une variable) est bissextile ou non (pour rappel, une année est bissextile si elle est divisible par 4 et non par 100 ou si elle est divisible par 400).

5.1.2 Construction `switch ...case`

Une cascade de `if` n'est pas toujours facile à lire et à mettre à jour. La construction `switch ...case` peut être utilisée à sa place. Sa syntaxe est la suivante :

2. En théorie des langages de programmation, une variable est ce qu'on appelle une *lvalue* et une constante est une `rvalue`. Une *lvalue* est affectable, une *rvalue* ne l'est pas. Bien entendu, une *lvalue* est aussi une *rvalue*, mais l'inverse est évidemment faux.

```
switch (expression) {
   case expression-constante_1 : instruction(s)_1
   case expression-constante_2 : instruction(s)_2
   default_optionnel : instruction(s)
}
```

Chaque *expression-constante* a un rôle d'étiquette, comme nous l'avons vu en section 3.3.5. *expression* est évaluée et si le résultat correspond à une des *expression-constante*$_i$, alors le programme continue son exécution par les *instruction(s)*$_i$ correspondantes, puis il exécute *instruction(s)*$_{i+1}$, *instruction(s)*$_{i+2}$, et ainsi de suite. Il s'agit d'une forme particulière de saut local (`goto`) qui permet d'obtenir un fonctionnement similaire à celui de la casacde des `if` mais qui autorise d'autres choses également.

Afin d'obtenir un comportement similaire à celui de la cascade des `if`, vous devez placer une instruction `break` à la fin de chaque série d'instructions *instruction(s)*$_i$ (cf. section 5.3.1 pour avoir le comportement du mot-clef `break`)[3]. De plus, je vous conseille de former un bloc avec chaque ensemble d'instructions à l'aide des accolades. C'est aussi valable pour les instructions de la clause par défaut :

```
switch (expression) {
   case expression-constante_1 :
      {
         instruction(s)_1
         break ;
      }
   case expression-constante_2 :
      {
         instruction(s)_2
         break ;
      }
   case expression-constante_3 :
      {
         instruction(s)_3
         break ;
      }
   default_optionnel :
      {
         instruction(s)_optionnelles
         break_optionnel ;
      }
}
```

En pratique, je ne veux pas voir de construction `switch ...case` écrite autrement que comme dans l'exemple donné ci-dessus.

5.1.3 Opérateur ternaire `?:`

L'opérateur ternaire n'est pas *stricto sensu* une instruction, mais permet d'écrire une instruction conditionnelle. La syntaxe de cet opérateur est la suivante :

expression$_1$ `?` *expression*$_2$ `:` *expression*$_3$

3. Même après la dernière série : on ne sait jamais, si vous devez rajouter des cas dans le futur, vous serez au moins couvert de ce côté-là.

Son fonctionnement est le suivant : $expression_1$ est évaluée complètement et si le résultat est "vrai", alors la valeur de l'exécution d'$expression_2$ est retournée, sinon c'est la valeur de l'exécution d'$expression_3$ qui est retournée.

Rappel : en C, une constante est une expression.

Voici un exemple d'utilisation de l'opérateur ternaire :

```
printf( x != 0 ? "coucou" : "hello" );
```

Ce code est plus compact que son équivalent écrit avec une instruction conditionnelle :

```
if (x != 0)
  printf("coucou");
else
  printf("hello");
```

Écrivez un programme qui affiche le maximum de deux variables, en utilisant l'opérateur ternaire `?:` .

5.2 Instructions itératives

Les autres instructions permettant un contrôle du flot d'exécution sont les instructions itératives. Il existe trois constructions syntaxiques, mais qui sont équivalentes fondamentalement. L'utilisation de l'une ou l'autre dépend de préférences personnelles et de situations particulières.

5.2.1 Boucle `while`

La première construction est celle de la boucle `while`. Sa syntaxe est la suivante :

`while` (*expression*)
  instruction

Le fonctionnement est le suivant : *expression* est évaluée et si la valeur du résultat est "vrai"(i.e. une valeur différente de zéro) alors *instruction* est exécutée et *expression* est évaluée de nouveau. Ce cycle se répète jusqu'à ce qu'*expression* soit évaluée à "faux" (zéro). Il suffit de mettre une constante non-nulle en tant qu'expression pour créer une boucle infinie : `while(1){ ...}`

Exemple

```
1 #include <stdio.h>
2
3 #define NLOOPS 10
4
5 int main(void)
6 {
7   int n = 0;
8   while( n < NLOOPS )
```

```
9       {
10        printf( "%i est ",n);
11        printf( n%2 ? "impair\n" : "pair\n") ;
12        ++n;
13      }
14    return 0;
15  }
```

while.c

À nouveau, je vous conseille de placer systématiquement un bloc d'instructions, même dans le cas où le corps de boucle n'est formé que d'une instruction simple (en encadrant l'instruction avec des accolades (`{` et `}`). Si vous devez rajouter d'autres instructions dans le corps de la boucle à l'avenir, vous éviterez une erreur potentielle.

Écrivez un programme utilisant une boucle `while` qui calcule la somme des `N` premiers entiers (`N` sera une constante du programme) et affiche le résultat.

Écrivez un programme qui compte (et affiche) le nombre de bits à 1 dans une variable entière.

5.2.2 Boucle `for`

La deuxième construction itérative utilise le mot-clef `for` et la syntaxe est la suivante :

`for (` *expression*$_1$ `;` *expression*$_2$ `;` *expression*$_3$ `)`
  instruction

Le fonctionnement est le suivant : le programme commence par exécuter *expression*$_1$ (qui sert d'initialisation et n'est exécutée qu'une unique fois), puis *expression*$_2$ est évaluée. Dans le cas où le résultat est "vrai" (non-nul, donc), alors le corps de la boucle for (*instruction*) est exécuté. Ce corps peut être formé de n'importe quel type d'instruction : simple, bloc d'instructions, de branchement, itérative, etc. Une fois ce corps exécuté, le programme passe à l'*expression*$_3$ qui est exécutée avant qu'*expression*$_2$ ne soit évaluée de nouveau. Ce cycle continue jusqu'à ce qu' *expression*$_2$ soit évaluée à "faux" (valeur nulle). En fait, cette construction avec le mot-clef `for` est équivalente à la construction suivante utilisant un mot-clef `while` :

expression$_1$ `;`
`while (`*expression*$_2$`) {`
  instruction
  expression$_3$ `;`
`}`

Si *expression*$_2$ est vide alors on considère que l'évalution est tout le temps "vraie". Il s'agit d'un autre moyen pour écrire une boucle infinie : `for(;;) {...}` Depuis C99, et la déclaration de

variables à la volée, il est licite pour $expression_1$ d'être une déclaration de variable combinée à une initialisation, comme le montre l'exemple ci-dessous :

```c
#include <stdio.h>

#define NLOOPS 10

int main(void)
{
  for(int i = 0 ; i < NLOOPS ; i++)
    {
      /* corps de la boucle */
    }
  return 0;
}
```

for_declaration_interne.c

Quand vous utilisez une fonctionnalité de C99, il est possible que vous ayiez à compiler votre code avec l'option `-std=c99`. Pour forcer le compilateur à passer en mode C ANSI (C89), vous pouvez utiliser l'option `-std=c89`.

Modifiez le programme de calcul des `N` premiers entiers et utilisez une boucle `for` en lieu et place de la boucle `while`.

Écrivez un programme qui trouve (et affiche) les années bissextiles entre 0 et l'année courante.

Avec l'opérateur de succession (cf. section 3.2.3), il est possible d'utiliser simultanément plusieurs index au sein de la même boucle, comme le montre l'exemple suivant :

```c
#include <stdio.h>

#define N 10

int main(void)
{
  for(int i=0,j=N ; i<=j ; i++,j--)
    {
      printf("valeur des indices : %i  %i\n",i,j);
    }
  return 0;
}
```

for_double_index.c

5.2.3 Boucle `do ...while`

Les constructions de boucle `while` et `for` commencent par tester la condition de sortie de boucle avant d'exécuter le corps de boucle. La construction `do ...while` permet une inversion en commençant par exécuter une première fois le corps de la boucle avant d'évaluer l'expression de sortie de boucle. Le corps de boucle est donc exécuté **au moins une fois**, ce qui est parfois utile dans certains programmes. La syntaxe de cette construction est la suivante :

```
do
   instruction
while (expression);
```

Vous remarquerez la présence du caractère de terminaison d'instruction (;) à la fin de la construction. Elle est équivalente à :

```
instruction_corps-de-boucle
while (expression)
   instruction_corps-de-boucle
```

Cependant, cette version entraîne une certaine duplication de code qui devra être évitée.

5.2.4 Diagrammes fonctionnels

Les figures 5.2 et 5.1 montrent une nouvelle fois que fondamentalement, il n'y a pas de différence entre une boucle `for` et une boucle `while`. Concrètement, il est toujours possible de remplacer l'une par l'autre et réciproquement. Cependant, utiliser une boucle `for` est souvent plus intuitif que d'utiliser une boucle `while` équivalente [4].

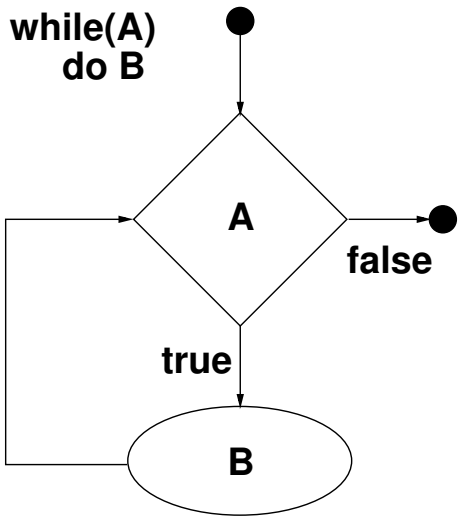


FIGURE 5.1: Diagramme fonctionnel d'une boucle `while`

4. C'est ce que l'on nomme du sucre syntaxique.

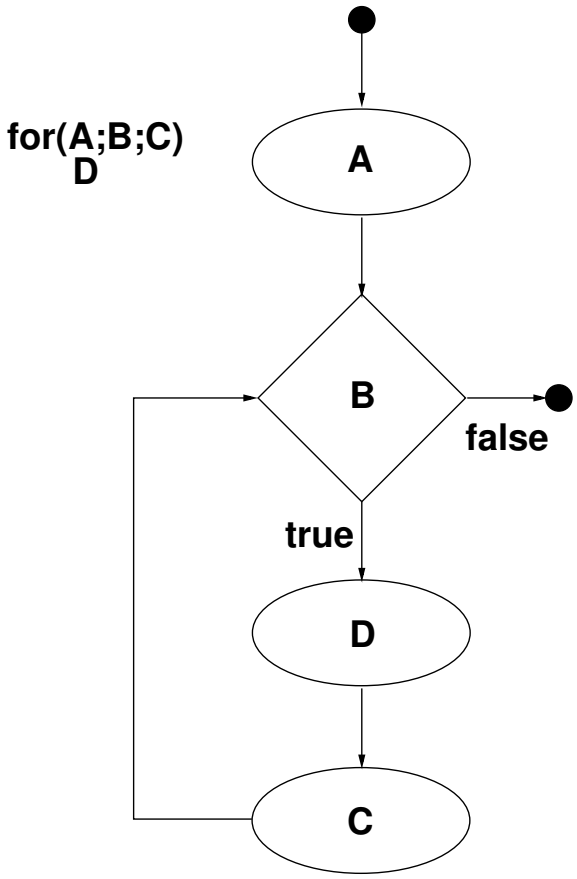


FIGURE 5.2: Diagramme fonctionnel d'une boucle `for`

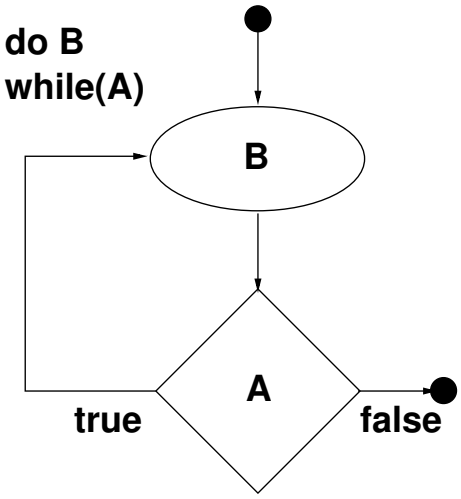


FIGURE 5.3: Diagramme fonctionnel d'une boucle `do ...while`

5.3 Instructions de saut

Les dernières instructions permettant un contrôle du flot d'un programme sont les instructions de saut. Basiquement, ces instructions permettent de sauter à un endroit déterminé dans le code et de continuer l'exécution à partir de ce point.

5.3.1 Saut depuis des boucles

La plupart des sauts concernent les corps de boucles, typiquement pour mettre fin à l'exécution d'une partie des instructions formant ce corps (par exemple, une situation est détectée qui implique d'abandonner l'exécution du reste de la boucle). Il existe pour ce faire deux mots-clefs : `break` et `continue`.

Instruction `break`

`break` permet d'interrompre l'exécution de la boucle courante ou d'un `switch` et d'en sortir, sans exécuter le reste du corps de la boucle **et des itérations** (s'il y en a). Par exemple :

```
while (expression_1)
   {
      while (expression_2)
         {
            instruction(s)_1
            if (une condition)
               break;
            instruction(s)_2
         }
      instruction(s)_3
   }
```

Si la condition du `if` est évaluée à "vrai", alors le `break` est exécuté, et *instruction(s)*$_2$ n'est pas exécuté car le flot programme va reprendre à partir d'*instruction(s)*$_3$.

Quel que soit le niveau d'imbrication des boucles, un `break` ne fera sortir le programme que de la boucle la plus interne dans laquelle le `break` se trouve. Si vous avez besoin de sortir de plus d'un niveau d'imbrication, il faut utiliser le saut local `goto` (cf. ci-dessous).

Instruction `continue`

`continue` n'est utilisable que dans un contexte de boucles : ce mot-clef permet de passer à l'itération suivante sans avoir à terminer l'actuelle. Il existe une subtile différence de fonctionnement entre une boucle `while` et une boucle `for` :

```
while (expression)
   {
      /* bla bla bla */
      if (une condition)
         continue;
      /* bla bla bla */
   }
```

Dans ce cas, le programme interrompt l'exécution du corps de la boucle et procède à l'évaluation

d'*expression*. Dans le cas d'une boucle `for` :

```
for ( expression1 ; expression2 ; expression3 )
   {
      /* bla bla bla */
      if (une condition)
         continue;
      /* bla bla bla */
   }
```

L'exécution de `continue` entraîne d'abord l'évaluation d'$expression_3$ (et réalise les effets de bords associés) **puis** l'évaluation d'$expression_2$.

5.3.2 Saut local avec `goto`

Dans le cas d'une imbrication de plusieurs boucles, il n'est pas possible d'utiliser `break` pour sortir de plus d'un niveau d'imbrication. La solution est alors d'utiliser une instruction plus générique de saut appelée `goto`, qui fonctionne de la façon suivante :

1. il faut d'abord poser une **étiquette** (cf. section 3.3.5) dans le code. Cette étiquette est juste un nom, suivi du caractère `:` . Cette étiquette marque juste un point de saut possible dans le code ;
2. il faut appeler `goto` avec le nom de l'étiquette voulue, ce qui va entraîner un saut sur l'endroit du code marqué par cette étiquette.

Exemple

```
main()
{
  int erreur;

  /*  Code .... */

  if(erreur > 0)
    goto fin_prog:

 /* Code .... */

fin_prog :
  printf("fin du programme\n");
  return 0;
}
```

`goto` est ce qu'on appelle une instruction de saut **local**, car l'étiquette référencée par `goto` doit avoir été posée dans le corps de la même fonction où se trouve le `goto`. En effet, `goto` ne sauvegarde pas le contexte d'exécution (cf. Figure 4.2) du programme (notamment l'état de sa pile et l'état des registres du processeur) et il faut donc rester dans la même fonction. Les sauts non-locaux (i.e entre fonctions différentes) sont possibles avec non pas des mots-clefs mais des fonctions particulières

du système UNIX, `setjmp` et `longjmp` qui permettent de sauvegarder/restaurer le contexte d'éxécution du programme.

`goto` est parfois utile pour éviter de dupliquer du code.

II Outils pour des programmes plus ambitieux

6. Fonctions et procédures

6.1 Syntaxe de déclaration

Jusqu'ici, la structure des programmes était relativement simple avec une unique fonction, le `main` et des déclarations annexes telles que les inclusions de fichiers d'entête de bibliothèques (les fichiers `.h`) et éventuellement des constantes et des variables globales. Un programme C se doit de contenir au moins cette fonction `main`, (dont le nom est imposé) afin d'être compilé pour obtenir un programme exécutable . Cependant, en vue de la création d'applications comportant un nombre important de lignes de code, il serait peu pratique que toutes ces lignes soient contenues dans la seule fonction `main` pour des raisons d'organisation, de lisibilité et de facilité de maintenance du code. De plus, isoler des fonctionnalités conceptuellement indépendantes au sein de fonctions permet de réutiliser facilement du code.

Il est impossible d'écrire des instructions en dehors du corps d'une fonction, à l'exception de la déclaration de variables globales (cf. section 4.3.2) ou de prototypes de fonctions (cf. section 6.1 ci-dessous).

À l'exécution d'un programme, c'est la fonction `main` qui est exécutée en premier (le point d'entrée du programme).

Dans ce chapitre, vous allez apprendre à déclarer et écrire des fonctions afin de mieux structurer vos codes applicatifs.

Une fonction est structurée en deux parties : le **prototype** et le **corps** :

type-de-retour nom-de-fonction (*liste-des-arguments*) ⟹ *prototype de la fonction*

```
{
   /* ici des instructions */          ⟹ corps de la fonction
}
```

Le corps de la fonction est un bloc d'instructions, identique à ceux que vous avez déjà rencontrés jusqu'à présent. Le prototype, lui, constitue en quelque sorte la "carte d'identité" de la fonction.

6.2 Visibilité

6.2.1 Portée dans le fichier de déclaration

La portée d'une variable dans un programme débute à son point de déclaration ; il est donc impossible de l'utiliser en dehors de sa portée car elle n'est pas (encore) visible. Ce comportement est identique pour les fonctions : afin d'utiliser une fonction (avec un appel, le plus souvent), il est indispensable qu'elle ait été déclarée auparavant dans le programme. Cependant, il n'est pas requis que l'intégralité de la fonction soit écrite : une déclaration au préalable de son prototype est suffisante. S'il est donc possible de dissocier un prototype du corps de la fonction, l'inverse n'est pas vrai car tout corps de fonction doit être immédiatement précédé de son prototype. Il est possible de déclarer un prototype de fonction localement dans un bloc d'instructions (par exemple dans le corps d'une *autre* fonction) mais en pratique cela est peu employé.

Exemples

Premier exemple

```c
#include <stdio.h>

int ma_fonction(int argument)
{
  int variable_locale = 0;
  variable_locale += 2*argument + 33;
  return variable_locale;
}

int main(void)
{
  int truc = 47;
  truc = ma_fonction(truc);
  printf("Valeur du resultat : %i\n",truc);
  return 0;
}
```

declaration_de_fonction.c

Dans cet exemple, la fonction `ma_fonction` est complètement déclarée (prototype et corps) avant son utilisation dans le `main`.

Second exemple

```c
#include <stdio.h>

int ma_fonction(int argument);

int main(void)
```

```
6  {
7    int truc = 73;
8    truc = ma_fonction(truc);
9    printf("Valeur du resultat : %i\n",truc);
10   return 0;
11 }
12
13 int ma_fonction(int argument)
14 {
15   int variable_locale = 0;
16   variable_locale += 2*argument + 33;
17   return variable_locale;
18 }
```

declaration_de_fonction2.c

Dans cet exemple, le prototype de la fonction `ma_fonction` est déclaré avant son utilisation dans le `main`. La fonction est complètement écrite (prototype + corps) dans le code postérieurement à son utilisation dans le `main`.

Une règle tacite d'organisation du code place la fonction `main` en tant que dernière fonction dans le code-source.

6.2.2 Visibilité dans un fichier externe

Tout comme les variables globales, les fonctions sont par défaut visibles en dehors du fichier source dans lequel elles sont déclarées. Afin de restreindre leur visibilité (et leur utilisation, donc) à leur fichier de déclaration, il faut préfixer leur prototype avec le mot-clef `static`.

Exemple

```
static int somme (int arg1, int arg2)
{
  return (arg1 + arg2);
}
```

La fonction `somme` ne sera plus utilisable en dehors du fichier où elle est déclarée.

6.3 Prototype

Nous allons examiner les éléments constitutifs du prototype, à savoir : l'identificateur (nom) de la fonction, le résultat retourné (sortie) et la liste des arguments (ou paramètres) qui sont pris en entrée.

6.3.1 Identification de la fonction

Les règles de nommage des fonctions sont les mêmes que celles de nommage des variables vues en section 4.4.1. Le langage C n'autorise pas la **surcharge** et par conséquent, des fonctions ne peuvent pas avoir le même nom s'il y a un risque de conflit au moment de la phase d'édition de lien de la compilation, même si leurs prototypes sont différents. En particulier, deux fonctions localisées dans des fichiers sources distincts peuvent porter des noms identiques si l'une des deux est statique car il n'y aura pas de conflit possible (c'est l'un des rôles du mot-clef `static` utilisé dans le cas des fonctions).

6.3.2 Résultat d'une fonction

Fonction vs. procédure

Il existe en fait deux catégories de fonctions :

— Les "vraies" fonctions qui renvoient un résultat (par exemple, une valeur entière, flottante, etc.). Dans ce cas, la dernière instruction exécutée dans le corps de la fonction sera l'instruction de saut `return` avec la valeur renvoyée : `retour` *expression* ;

— les fonctions qui ne renvoient pas de résultat, auquel cas le type retour est spécifique : il s'agit de `void` (rien). Une telle fonction est appelée une **procédure**. Dans ce cas il n'est pas obligatoire d'insérer l'instruction de saut `return` dans le corps de la procédure. Avoir un tel mécanisme de procédure peut sembler inutile, mais à la différence des mathématiques, les fonctions en C s'exécutent dans un environnement donné, que l'on appelle un **contexte d'exécution** . Une fonction ou une procédure n'est pas seulement un moyen de produire un résultat, mais aussi un éventuel moyen de produire ce que l'on appelle des **effets de bord** (*side effect*) c'est-à-dire apporter des modifications à cet environnement d'exécution ou à l'état de la machine qui exécute le programme. Par exemple, un affichage à l'écran ou une allocation de mémoire sont des effets de bord.

Exemple

L'exemple ci-dessous montre une déclaration de procédure ainsi que son utilisation dans le `main`. L'effet de bord consiste à effectuer l'affichage d'une chaîne de caractères à l'écran. On remarquera par ailleurs que `main` est une fonction qui renvoie un résultat de type entier `int`.

```c
#include <stdio.h>

void affichage(void)
{
  printf("Coucou\n");
}

int main(void)
{
  affichage();

  return 0;
}
```

procedure.c

Pratiquement, on trouve peu de "vraies" procédures dans les codes-sources, car si l'objectif est principalement de produire des effets de bord, le code de retour sera utilisé pour faire remonter d'éventuelles erreurs dans le programme (valeurs classiques : `1`, `0`, `-1`).

Une fonction qui spécifie un type de retour autre que `void` est **obligée** de retourner un résultat de ce type ; la présence de l'instruction de saut `return` est dès lors obligatoire. En revanche, la prise en compte de ce résultat est optionnelle dans la partie de code qui fait appel à la fonction, comme le montre l'exemple ci-dessous :

```c
#include <stdio.h>

int toto(int a, int b)
{
  printf("Dans la fonction\n");
  return a + b;
}

main()
{
  int x = 1, y = 3;
  toto(x,y);
  return 0;
}
```

resultat_optionnel.c

La fonction `toto` renvoie un résultat de type entier (`int`), qui n'est pas récupéré et utilisé dans la fonction `main`.

Quand on sort de la fonction `main`, le programme se termine. Pourquoi cette fonction `main` renvoie-t-elle un résultat alors qu'il n'est pas possible de le récupérer dans le reste du programme ?

Type du résultat d'une fonction

C étant un langage typé, nous avons déjà vu que les variables, les constantes et plus généralement les expressions sont typées. Pour réutiliser le résultat d'une fonction (par exemple le stocker dans une variable), il est donc logique que ce résultat soit également typé. Ainsi que le montre la déclaration de prototype en début de chapitre, ce type est placé avant le nom de la fonction. Cependant, il doit être restreint à la liste suivante :

- `void` : la fonction ne renvoie pas de résultat, c'est donc une procédure ;
- n'importe quel type de base de C : entier ou flottant, signé ou non-signé ;
- un pointeur (cf. chapitre 8), de n'importe quel type ;
- les types définis par l'utilisateur (avec `typedef` ou `struct`).

Un tableau (cf. Chapitre 7) n'est **jamais** un type possible de retour, même dans le cas d'une redéfinition de type avec `typedef` [1]. Pour retourner un tableau depuis une fonction, on utilisera en fait un pointeur sur le premier élément de ce tableau (cf. section 8.3.3).

Type par défaut et déclaration implicite de fonction

Quand un programme fait appel à une fonction dont le prototype n'est pas spécifié ou déclaré avant cet appel, le compilateur suppose que la fonction retourne une valeur entière de type `int`, ce qui peut engendrer des erreurs à l'exécution. Le message d'alerte du compilateur est le suivant :

1. Exemple de telle redéfinition : `typedef int my_tab_t[10];` qui définit un nouveau type `my_tab_t`, désignant un tableau de dix entiers.

```
warning: implicit declaration of function 'bidule'
```

Un tel message ne doit pas être traité avec légereté ! Prenons l'exemple suivant :

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

int *ma_fonction(void)
{
  int *ptr = malloc(16);
  printf("Adresse %p\n", ptr);
  return ptr;
}
```

fonction_annexe.c

Cette fonction alloue un peu de mémoire (cf. section 8.5) et retourne l'adresse de la mémoire allouée ; cette adresse est codée sur **64 bits**. Cette fonction est écrite dans son propre fichier, qui est compilé de façon indépendante [2]. Maintenant, considérons un programme qui utilise cette fonction dans son code :

```c
#include <stdio.h>

int main(void)
{
  int *ptr = NULL;
  ptr = ma_fonction();
  printf("Adresse     %p\n",ptr);
  return 0;
}
```

prog_principal_sans_proto.c

Ce programme utilise la fonction d'allocation, mais ne déclare nulle part son protoytype, le compilateur génère alors une alerte et va considérer que le type de retour par défaut de la fonction `ma_fonction` sera un entier de type `int`, qui lui est codé sur **32 bits**. Lors de l'exécution on se retrouve avec :

```
Adresse 0x556422a69260
Adresse     0x22a69260
```

Vous remarquerez que l'adresse affichée dans le `main` correspond aux 32 bits de poids faible de l'adresse affichée dans la fonction d'allocation. En effet, le compilateur a **tronqué** les 64 bits retournés initialement pour n'en garder que 32. Utiliser ensuite cette adresse dans le programme principal risque de se solder par une erreur. *A contrario*, si le programme déclare le prototype de la fonction `ma_fonction` avant de l'utiliser (ligne 3 du code) :

2. Avec la commande `gcc -c fonction_annexe.c` qui créé un fichier objet `fonction_annexe.o`.

```c
#include <stdio.h>

int *ma_fonction(void);

int main(void)
{
  int *ptr = NULL;
  ptr = ma_fonction();
  printf("Adresse %p\n",ptr);
  return 0;
}
```

prog_principal_avec_proto.c

alors la compilation se déroule sans problème et l'exécution est conforme à nos attentes :

```
Adresse 0x55b0cabe6260
Adresse 0x55b0cabe6260
```

6.3.3 Liste des arguments

Le dernier élément du prototype est la liste des arguments (également appelés paramètres) passés à la fonction. Cette liste se présente sous la forme suivante :

nom-de-type nom-argument$_1$, *nom-de-type nom-argument*$_2$, ...

La virgule utilisée ici n'est **pas** l'opérateur de succession.

Il est néanmoins possible pour une fonction de ne pas prendre d'arguments, auquel cas, le mot-clef `void` est utilisé [3]. Cette liste d'arguments est un élément important car il permet au compilateur d'effectuer des vérifications entre le prototype de déclaration et les appels de fonctions dans le code. Le compilateur utilise :

- le nombre d'arguments
- le type des arguments
- l'ordre des arguments

pour effectuer ces vérifications. En revanche, le *nom* des arguments n'est pas important. D'ailleurs, lors d'une déclaration indépendante de prototype de fonction, il est possible d'omettre les noms des arguments, mais des noms explicites peuvent parfois aider à mieux appréhender ce que la fonction est censée utiliser comme paramètres. En revanche, ces noms sont indispensables dans le prototype précédant le corps de la fonction.

Le nom d'une fonction et la liste de ses arguments forme un ensemble souvent appelé la *signature* de la fonction [4].

3. Ce mot-clef est utilisé à des fins différentes dans le langage C, il ne signifie pas toujours "rien" ou "vide".
4. Élement de culture générale en programmation totalement indispensable pour briller en société.

6.4 Passage d'arguments

Le passage des arguments à une fonction doit donc respecter le prototype (nombre, ordre et types du ou des arguments s'il y en a). Il n'est pas indispensable que les noms des variables passées en argument à une fonction au moment de l'appel soient identiques aux noms indiqués dans le prototype au moment de sa déclaration.

6.4.1 Ordre d'évaluation des arguments

Les arguments peuvent être des expressions qui seront évaluées et dont le résultat constituera la valeur passée à la fonction et utilisée dans le corps de cette dernière. **Le langage C ne spécifie pas l'ordre dans lequel l'évaluation des ces diverses expressions est effectuée.** En particulier, rien ne garantit que cette évaluation soit effectuée de gauche à droite.

La virgule utilisée ici n'est **pas** l'opérateur de succession et ne garantit pas un ordre d'évaluation.

Ainsi, **il est interdit d'avoir en argument d'une fonction une expression réalisant des effets de bord qui sont susceptibles d'avoir une influence sur les autres arguments**. Par exemple, imaginons trois fonctions `f`, `g` et `h` telles que :

```
void f(int a, int b);
int g(int a);
int h(int a);
```

Alors l'appel suivant est interdit :

```
int i = 0;
f(g(++i),h(i));
```

En effet, il est impossible de savoir si cet appel sera `f(g(1),h(1))` ou `f(g(1),h(0))`. L'ordre d'évaluation des arguments lors d'un appel de fonction est laissé à la discrétion du compilateur. Ce dernier n'effectue aucune vérification que les arguments obéissent à la règle d'indépendance ; il vérifie uniquement que leur type correspond à celui du prototype de la fonction.

6.4.2 Passage par valeur

En C, les arguments d'une fonction sont passés **par valeur** [5]. Mais qu'est-ce que cela veut dire exactement ? Pour le savoir, nous allons faire un petit exercice.

Écrivez un programme dont le `main` appelle une fonction `swap` qui échange la valeur de deux variables. Le prototype de cette fonction est donc :

```
void swap(int a, int b);
```

Affichez les valeurs des variables utilisées en argument avant et après l'appel à `swap`. Affichez également les valeurs des arguments *dans la fonction* `swap` en entrant et sortant de la fonction. Exécutez ce programme. Que constatez-vous ? Quelle explication pouvez-vous donner ?

En fait, ce ne sont pas les variables originales qui sont utilisées par la fonction mais des **copies**. **Les variables originales ne sont donc jamais modifiées dans le corps de la fonction lors d'un**

5. par opposition à un passage par adresse.

appel. Ainsi que nous le verrons, il est possible de modifier dans la fonction une variable passée en argument en fournissant son adresse (cf. Chapitre 8 consacré aux pointeurs). En pratique, ces copies d'arguments sont placées sur la pile d'exécution du programme quand la fonction est appelée et au retour de cet appel, cette mémoire peut être réutilisée : les copies sont donc des variables automatiques et il est alors possible de leur adjoindre la classe de stockage `register`.

6.4.3 ++ Fonctions à nombre variable d'arguments

Il est possible d'écrire des fonctions avec un nombre variable d'arguments avec l'utilisation des macros `va_start`, `va_arg` et `va_end`. Pour se faire, il faut indiquer les arguments obligatoires de la fonction puis qu'il y aura une liste avec trois points (`...`). Par exemple : `int func(int arg, ...);` Ce prototype indique que la fonction `func` prend **au moins** un argument de type entier puis une liste dont la taille n'est pas connue et qu'il va falloir traiter à l'aide des macros sus-nommées.

Ces macros possèdent les prototypes suivants :

```
void va_start(va_list ap, last);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

`va_start` doit être appelée en premier et initialise la liste des arguments `ap` qui va être utilisée par les deux autres macros. `last` est le dernier argument de la fonction avant la liste des arguments à traiter. C'est le dernier argument dont on connait *a priori* le type. Par exemple, `printf` est une fonction à nombre variable d'arguments, mais elle prend au moins un argument qui est une chaîne de caractères puis éventuellement une liste formée par le reste des arguments : `printf(char *str,...)`. Dans ce cas, le dernier argument avant la liste est la chaîne `str` dont le type est connu.

La macro `va_arg` traite la liste des arguments : chaque nouvel appel retourne une expression dont le type est la valeur qui correspond au prochain argument de la liste. `type` est donc un identificateur (nom) de type.

La macro `va_end` doit être appelée pour chaque occurence de `va_start` dans la fonction.

Je ne donne pas plus d'informations (si vous voulez en savoir plus, je vous invite à regarder le manuel correspondant), et je vous laisse méditer ce code récupéré sur les internets [6] :

```
1  #include<stdarg.h>
2  #include<stdio.h>
3
4  int sum(int num_args, ...) {
5    int val = 0;
6    va_list ap;
7
8    va_start(ap, num_args);
9    for(int i = 0; i < num_args; i++)
10     {
11       val += va_arg(ap, int);
12     }
13   va_end(ap);
14   return val;
15 }
16
```

6. Code pompé sans vergogne sur `https://www.tutorialspoint.com/c_standard_library/c_macro_va_start.htm`

```
17 int main(void) {
18   printf("Sum of 10, 20 and 30 = %d\n",
19    sum(3, 10, 20, 30) );
20   printf("Sum of 4, 20, 25 and 30 = %d\n",
21    sum(4, 4, 20, 25, 30) );
22   return 0;
23 }
```

fonction_args_variable.c

6.5 Appel de fonction

Pour utiliser une fonction dans un code, il faut effectuer un **appel de fonction**, en utilisant l'opérateur unaire dédié : `()`. Pour rappel, un appel de fonction est une expression composée. L'appel à une fonction doit être cohérent avec sa déclaration : c'est à dire que les types de retour et les listes d'arguments doivent être similaires. Si la définition de la fonction et ses appels se font dans le même fichier, alors le compilateur sera capable de détecter les incohérences et indiquera une erreur. En revanche, si le code de la fonction se trouve dans un autre fichier, compilé séparément du fichier ou l'appel est fait [7], alors la différence ne sera pas forcément détectée (cf. section 6.3.2).

Attention à ne pas confondre appel de fonction et pointeur de fonction (cf. section 8.4.3) !

6.5.1 Appels récursifs

Il est licite pour une fonction de s'appeler elle-même, c'est à dire d'être **récursive** . Attention cependant à bien précsier une condition d'arrêt dans le corps de la fonction pour éviter une récursion infinie, qui, en pratique, débouchera sur une erreur à cause d'un **débordement de la pile d'exécution du programme** (*stack overflow*). Comparez les fonctions `loop_overflow` et `loop_no_overflow` dans le code ci-dessous :

```
1  #include <stdio.h>
2
3  int loop_overflow(int arg)
4  {
5    printf("Arg val : %i\n",arg);
6    return loop_overflow(--arg);
7  }
8
9  int loop_no_overflow(int arg)
10 {
11   printf("Arg val : %i\n",arg);
12   if(arg)
13     return loop_no_overflow(--arg);
14 }
```

7. De tels fichiers sont en général appelés des **bibliothèques**.

```
15
16 int main(void)
17 {
18   int var = 10;
19   loop_no_overflow(var);
20   loop_overflow(var);
21   return 0;
22 }
```

overflow.c

La condition d'arrêt est placée en ligne 12.

Exo ÉCrivez une fonction qui calcule les N premiers termes de la suite de Fibonacci : $Fibo(0) = 0, Fibo(1) = 1, Fibo(n) = Fibo(n-1) + Fibo(n-2)$, N étant une constante du programme.

6.5.2 ++ *inlining* de fonctions

Il est possible de préfixer le prototype d'un fonction avec le mot-clef `inline` ce qui donne au compilateur des indications d'optimisation pour cette fonction. Ce dernier va *essayer* de remplacer dans le code un *appel* de fonction par le corps de la fonction, ce qui évite l'appel justement (et est donc plus rapide). Le mot-clef `inline` a pour les fonctions un peu le même effet que le mot-clef `register` pour les variables. De plus, il **faut** préciser une classe de stockage `static` ou `extern` afin de pouvoir faire de l'*inlining*. Le comportement des fonctions `static inline` est le même quel que soit le dialecte de C utilisé, ce qui n'est pas le cas des fonctions `extern inline`. Il est cependant peu probable que vous ayiez à utiliser des fonctions *inline* dans vos codes et si cela devait arriver, elles seraient très probablement de classe `static`.

7. Tableaux

7.1 Gestion des données vectorielles avec les tableaux

Les variables utilisées jusqu'à présent permettent une gestion de données scalaires mais sont peu adaptées pour manipuler des données vectorielles qui doivent être accédées ensemble conceptuellement parlant. De plus, il est intéressant d'avoir des garanties de contigüité sur les emplacements mémoire utilisés (en vue d'améliorer les performances).

Les accès aux données suivent un principe de localité spatio-temporelle : quand des données sont accédées à un instant t, alors d'autres données proches en mémoire seront sans doute accédées également dans un temps assez court.

Le langage C offre une construction qui permet de regrouper des données en vecteurs : les tableaux.

À la différence des variables scalaires, les tableaux ne sont pas des types de retour licites pour les fonctions. On utilisera des pointeurs à la place (cf. section 8.3.3). En revanche, un tableau peut être passé en argument à une fonction.

7.1.1 Déclaration de tableau

Les tableaux sont des collections de variables possédant **des types identiques**. Ils sont construits à l'aide de l'opérateur de réplication : `[]`, soit :

identificateur-de-type identificateur-du-tableau`[`*taille*`]`

Par exemple :

```
int toto;   /* toto est une variable de type entier */
int tab[4]; /* tab est une variable de type tableau de 4 entiers */
```

C'est parce que `[]` est un opérateur de réplication que tous les éléments du tableau sont de même type. Pour construire des collections d'élements de types différents, le C propose une construction syntaxique qui s'appelle une structure (cf. Chapitre 9).

Il n'est pas possible de déclarer des tableaux de type `void` car ce n'est pas un véritable type de données. Cela n'aurait pas vraiment de sens par ailleurs.

7.1.2 Taille d'un tableau

Initialisation et taille

En C, la définition d'un type ou la déclaration d'une variable n'est possible que si le compilateur est capable d'en calculer la taille (occupation mémoire). Cette taille **doit** être spécifiée au moment de la déclaration, et dans le cas des tableaux cela signifie qu'il faut mettre une valeur entre les crochets `[]`. Par exemple :

```
char bidule[16]; /* un tableau de 16 caracteres */
```

S'il est théoriquement possible de mettre une expression *quelconque* dans l'opérateur de réplication, je ne vous l'autorise pas en pratique [1]. Vous ne pourrez utiliser que des expressions **constantes** pour les tailles des tableaux déclarés avec l'opérateur `[]`. Par exemple :

```
#define TAB_SIZE 32

int  val = ....         /* valeur quelconque              */
int  bidule[TAB_SIZE]; /* declaration OK                  */
char truc[10];         /* declaration OK                  */
int  machin[val];      /* declaration interdite en PG109 */
```

Il est possible de ne **pas** indiquer de taille au moment de la déclaration à condition que le tableau soit initialisé à cet instant : c'est le compilateur qui va calculer la bonne taille tout seul pour déterminer la quantité de mémoire à utiliser pour stocker le tableau. Cette initialisation à la déclaration ne peut se faire qu'avec des expressions constantes. En effet, cette initialisation est dite *statique* car elle est effectuée au moment de la compilation [2] et les expressions ne sont donc pas évaluées car le programme ne s'exécute pas encore. L'initialisation d'un tableau est effectuée avec une liste de valeurs formée de cette façon : { $valeur_1$, $valeur_2$, ..., $valeur_n$ }
Par exemple :

```
int  tab[] = {1,2,3};            /* tab est un tableau de 3 entiers    */
char str[] = {'c','o','o','l'}; /* str est un tableau de 4 caracteres */
```

Opérateur de taille

Alors autant l'écrire le plus clairement possible :

EN C, IL N'EXISTE PAS D'OPERATEUR QUI RENVOIE LA TAILLE D'UN TABLEAU !

Voyons voir : cette phrase est écrite en gras, en majuscules, dans une police de caractère assez grosse, elle est soulignée et encadrée. À mon avis, ça doit être important. Mais reprenons :

1. C'est pour votre bien, faites-moi confiance.
2. Par opposition à une affectation qui est qualifiée de **dynamique** car elle est effectuée à l'exécution du programme.

pas d'opérateur, donc (ni de fonction non plus). En particulier, l'opérateur `sizeof` ne sert pas à effectuer ce genre de calcul de taille (cf règles p. 135). Cependant, le programme suivant est correct :

```c
#include <stdio.h>
int main(void)
{
  int tab[10];
  printf("Le tableau possede %2lu elements\n",
   sizeof(tab)/sizeof(tab[0]));
  return 0;
}
```

calcul_taille_tableau.c

Ce programme affiche :

```
Le tableau possede 10 elements
```

Il s'agit d'un cas particulier et vous remarquerez que la taille du tableau est **déjà** connue car il suffit de consulter sa déclaration. L'utilisation de `sizeof` n'est clairement pas justifiée ici. Or, vous vous servez de ce cas **très particulier** pour le généraliser et l'appliquer dans des situations où cela ne fonctionnne pas. Le cas le plus courant est le calcul de la taille d'un tableau dans le corps d'une fonction où il est pris en argument. Examinez le code suivant :

```c
#include <stdio.h>

long int array_size(int array[])
{
  return (sizeof(array)/sizeof(array[0]));
}

int main(void)
{
  int tab[10];
  printf("Le tableau possede %2lu elements\n",
   sizeof(tab)/sizeof(tab[0]));
  printf("Le tableau possede %2lu elements\n",
   array_size(tab));
  return 0;
}
```

calcul_foireux_taille_tableau.c

Ce programme affichera :

```
Le tableau possede 10 elements
Le tableau possede  2 elements
```

Exo Expliquez la différence d'affichage entre le `main` et la fonction qui calcule la taille du tableau.

En fait, je ne vois qu'un seul cas où l'utilisation de l'opérateur `sizeof` pourrait [3] se justifier :

```c
#include <stdio.h>

int main(void)
{
  int array[] =
    {12,0,23,56,76,89,90,-22,23,1,13,53,17,89,56,-1};
  printf("taille du tableau :%lu\n",
   sizeof(array)/sizeof(array[0]));
  return 0;
}
```

sizeof_ok.c

Mais cela reste tout de même très anecdotique et donc :

Je serai impitoyable avec celui ou celle qui utilisera `sizeof` pour calculer la taille d'un tableau (cf. les règles énoncées en page 135) [4].

Je ne serai pas aussi impitoyable que lui (quoique).

Il est impossible de déterminer la taille d'un tableau (en octets ou en nombre d'éléments, peu importe) à partir de son seul identificateur. Quand vous passez un tableau en argument à une fonction, vous **devez** passer également sa taille en argument, sauf si ladite taille est disponible autrement (par le truchement d'une variable globale, par exemple).

7.1.3 Accès à un tableau

L'accès aux éléments du tableau se fait également à l'aide de l'opérateur `[]`. Par exemple, si la déclaration suivante est faite (`TAILLE_TABLEAU` est une constante quelconque) :

```c
int array[TAILLE_TABLEAU];
```

3. Notez l'usage du conditionnel.
4. Je vous tirerai dans les rotules avec des balles rouillées jusqu'à ce que vous attrapiez le tétanos.

alors `array[`i`]` est une variable de type entier `int`, avec $i \in \{0, 1, \ldots, $`TAILLE_TABLEAU-1`$\}$.

Ainsi, il est possible d'accéder directement à n'importe quel élément du tableau directement en connaissant sa position (son indice) dans le tableau (*random access*).


La numérotation des éléments d'un tableau commence à 0.

Par ailleurs, les bornes d'un tableau ne font l'objet d'aucune vérification (à la différence d'autres langages), mais des outils de développement externes comme Valgrind peuvent vous aider dans ces vérifications [5].

Enfin, dans le cas d'un accès à un élément, les valeurs acceptables pour l'opérateur `[]` sont signées, c'est-à-dire qu'il est possible de mettre une valeur négative.


L'indice d'un élément dans un tableau indique **le décalage de cet élément par rapport au premier élément du tableau**, ce qui explique que des indices négatifs soient possibles.


Écrivez une autre version du programme calculant et **stockant** les N premiers termes de la suite de Fibonacci. L'affichage de ces termes devra être effectué **après** qu'ils auront été **tous** calculés (et non pas au fur et à mesure).

++ Type des indices de parcours des tableaux

Il est très fréquent d'utiliser des variables de type `int` pour parcourir des tableaux, comme dans l'exemple ci-dessous :

```
char tableau[TAILLE];

for(int index = 0 ; index < TAILLE ; index++)
{
  tableau[index]  = 0;
}
```

La variable `index` est du type `int`. Or, cela peut poser un problème car la taille d'un tableau peut être supérieure à la taille maximale d'un `int`, même en version non signée. Un entier n'est potentiellement pas assez grand pour stocker l'information de taille (en nombre d'éléments) d'un tableau. Une meilleure solution est d'utiliser une variable de type `size_t` qui permet de stocker la taille maximale d'un tableau de n'importe quel type. Il existe par ailleurs un type qui sert à stocker explicitement un nombre d'éléments : `ptrdiff_t` [6]. Si vous souhaitez comprendre mieux pourquoi ce type permet de stocker des éléments, vous pouvez sauter directement en section 8.3.5.

Le risque que vous rencontriez des problèmes est assez faible et vous pouvez conserver des indices de type `int` pour vos parcours de tableaux. Sachez cependant que des problèmes peuvent en découler. Pour plus d'informations à ce sujet, je vous invite à consulter la page suivante : `https://www.viva64.com/en/a/0050/` (attention, c'est un peu technique).

7.1.4 Les tableaux en tant qu'arguments de fonction

7.2 Cas des chaînes de caractères

Le type chaîne de caractères n'existe pas en C, seul le type caractère `char` est disponible. Les chaînes de caractères pouvant être considérées comme des ensembles de caractères, elles seront simplement stockées dans des tableaux.

5. Nous verrons un certain nombre d'outils, dont ce fameux Valgrind au prochain semestre, en cours de PG110.
6. Il faut faire un `#include` de `<stddef.h>` pour avoir ce type.

7.2.1 Tableau de caractères vs chaînes de caractères

Une chaîne de caractères est un tableau de caractères avec une particularité : un caractère spécifique, dit de fin de chaîne `'\0'` est forcément le dernier élément du tableau.

Donc, un tableau de caractères qui ne contient pas celui de fin de chaîne `'\0'` n'est **pas** une chaîne de caractères. Les fonctions de gestion de chaîne de caractères comme `strlen`, `strcpy` et `strcmp` sont alors inutilisables dans ce contexte (cf. règles p. 135). Également, il faut prévoir dans un tableau de caractères une place supplémentaire pour y stocker ce caractère de terminaison de chaîne. Dans le cas d'une initialisation de tableau de caractères avec une chaîne constante (entre " et "), le caractère de fin de chaîne est automatiquement ajouté. Par exemple :

```
char chaine[] = "toto";
```

Comme la taille n'est pas précisée dans l'opérateur de réplication, c'est le compilateur qui réserve l'espace nécessaire, à savoir 5 octets : 1 pour chaque lettre et 1 pour le caractère de terminaison de chaîne.

Quelle sont les différences entre ces déclarations et initialisations ?

```
char tab1[] = {'t','o','t','o'} ;
char tab2[] = {'t','o','t','o','\0'} ;
char tab3[] = "toto";
```

Correction :
Si le caractère de fin de chaîne n'est pas présent, alors le tableau n'est pas une chaîne de caractères, mais juste un tableau de caractères. Donc les déclarations de `tab2` et `tab3` sont identiques : ce sont des chaînes de caractères. En revanche, `tab1` est juste un **tableau** de caractères.

Implémenter les fonctions `strlen`, `strcpy` et `strcmp`.

7.2.2 Interprétation de la valeur d'un caractère et table ASCII

Un caractère peut être interprété deux façons : il est possible d'utiliser soit sa valeur numérique (%hhi avec `printf`) ou bien le véritable caractère (%c avec `printf`). Ce caractère est celui qui correspond à la valeur dans la table ASCII. Le programme suivant affiche le contenu affichable de la table ASCII.

```
1  #include <stdio.h>
2
3  int main()
4  {
5    for(char c = 32; c < 127 ; c++)
6     {
7       printf("%3hhi - %c ",c,c);
8       if(!(c%6))
9         printf("\n");
10    }
11
12   return 0;
```

```
13 }
```

ascii_table.c

Le caractère de terminaison de chaîne `'\0'` a une valeur de 0.

Les chiffres se trouvent à des emplacements contigus dans la table ASCII et le caractère `'1'` n'a pas pour valeur ASCII 1, mais 49. La valeur numérique d'un caractère chiffre peut être obtenue en retirant la valeur ASCII du caractère `'0'` à la valeur du chiffre (i.e *chiffre* - `'0'`).

Écrivez un programme qui convertit une chaîne de caractères ne contenant que des chiffres en sa valeur numérique. Par exemple, la chaîne `"1234"` devra être convertie en un entier valant 1234.

7.3 Retour sur le prototype de la fonction `main`

Jusqu'à présent le prototype de la fonction `main` que nous avons utilisé est le suivant :

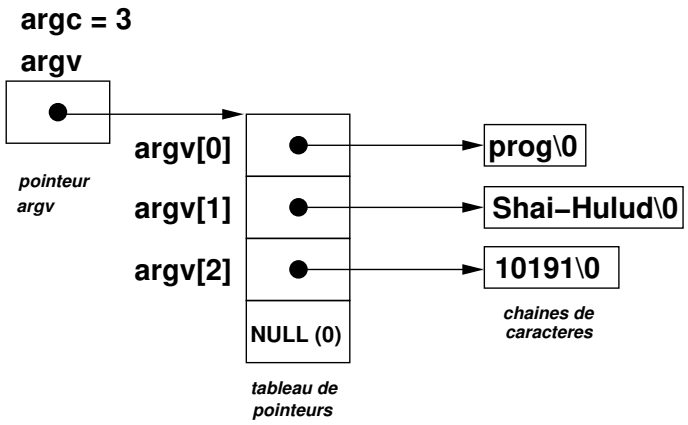
```
int main(void);
```

En réalité, le véritable prototype de cette fonction admet deux arguments : `argc` et `argv`.

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

`argc` est un entier dont la valeur est le nombre d'arguments fournis au programme.

`argv` est pointeur vers un tableau de pointeurs de caractères, que nous assimilerons à des **chaînes de caractères** pour le moment (en attendant de voir le Chapitre 8). Par exemple, quand le programme `prog` est lancé avec les arguments `Shai-Hulud` et `10191`, on obtient la situation suivante [7] :



Ces deux arguments (`argc` et `argv`) sont utilisés pour gérer ce que l'on appelle la **ligne de commande** du programme. L'utilisation de ces arguments permet au programmeur ou à la programmeuse de fournir des données au programme au moment de son exécution.

7. Ce schéma s'inspire violemment du livre consacré à C de Kernighan et Ritchie.

7.3.1 Gestion des arguments de la ligne de commande

Nombre et types des arguments

Un programme possède `argc` arguments qui sont stockés en tant que *chaînes de caractères*. `argv[0]`, `argv[1]`, ..., `argv[argc-1]` sont ces chaînes de caractères. Par ailleurs, et par convention :

— `argv[0]` est une chaîne de caractères contenant le nom du programme qui a été lancé ;
— `argv[argc]` est un pointeur qui a pour valeur `NULL` (cf. chapitre 8). `NULL` a pour valeur 0, ce qui correspond également à la valeur ASCII du caractère de terminaison de chaîne `'\0'`. Tout se passe comme si `argv[argc]` était en quelque sorte une "chaîne vide".

Un programme possède donc toujours *au moins* un argument (i.e `argc` $\geqslant$ **1**), à savoir son nom.

Écrivez un programme qui affiche tous les arguments qui lui sont passés en ligne de commande.

Conversion des arguments numériques

Étant donné que les arguments sont passés au programme sous forme de chaînes de caractères, il n'est pas possible d'utiliser directement ces arguments quand ils représentent des **nombres**. En effet la **chaîne de caractères** `"123"` n'a pas pour valeur numérique le **nombre** 123. Il faut donc effectuer une conversion, à l'aide de l'une des fonctions suivantes :

— `atoi` (*alpha to integer*) : permet de convertir une chaîne en entier (`integer`)
— `atol` (*alpha to long integer*) : permet de convertir une chaîne en entier long (`long integer`)
— `atoll` (*alpha to long long integer*) : permet de convertir une chaîne en entier très long (`long long integer`)
— `atof` (*alpha to float*) : permet de convertir une chaîne en nombre flottant (`double`)

Écrivez un programme qui affiche la somme des entiers qui lui sont passés en ligne de commande.

7.3.2 Transmission d'informations entre un programme et son utilisateur/utilisatrice

Un point important concerne les interactions entre un programme et son utilisateur/utilisatrice. Il existe trois moyens pour mettre en place une telle interaction :

— par l'utilisation d'un mode interactif avec des fonctions comme `scanf` et `printf` par exemple (ce ne sont pas les seules) ;
— par l'utilisation de variables d'environnement via les fonctions `setenv` et `getenv` ;
— par l'utilisation de la *ligne de commande* du programme.

Vous avez utilisé jusqu'à présent le premier moyen. Le second moyen n'est pas celui le plus utilisé spontanément mais il ne doit pas être négligé pour autant. Mais pratiquement, c'est la dernière méthode qui est employée le plus souvent.

7.4 ++ Tableaux multidimensionnels

L'opérateur de réplication peut être utilisé pour répliquer des tableaux. Dans ce cas, on obtiendra en quelque sorte des **tableaux de tableaux** [8] ou plus généralement des tableaux multidimensionnels.

8. C'est-à-dire des matrices si l'on se restreint à deux dimensions.

7.4.1 Disposition en mémoire des tableaux multidimensionnels

Soit la déclaration suivante :

```
int matrice[5][10];
```

Cette déclaration n'est pas aussi intuitive qu'il n'y paraît au premier abord. En effet, ici, on ne réplique pas 10 fois un tableau de 5 entiers, mais on réplique 5 fois un tableau de 10 entiers (cette déclaration n'est donc pas équivalente à `(int matrice[5])[10]` ;). On dit que la disposition des tableaux en mémoire est de type **Row-Major**[9].

Avec un modèle mémoire de type *Row-Major*, c'est toujours la dimension **la plus à gauche qui varie le moins vite** dans le cas d'un parcours du tableau et celle **la plus à droite qui varie le plus vite**. Si on se réprésente le tableau sous forme de matrice avec `X` colonnes et `Y` lignes, ces lignes et colonnes sont en réalité disposées de façon contigües en mémoire avec `X` tableaux de `Y` élements "collés" les uns aux autres, ce qui correspondrait à la déclaration suivante :

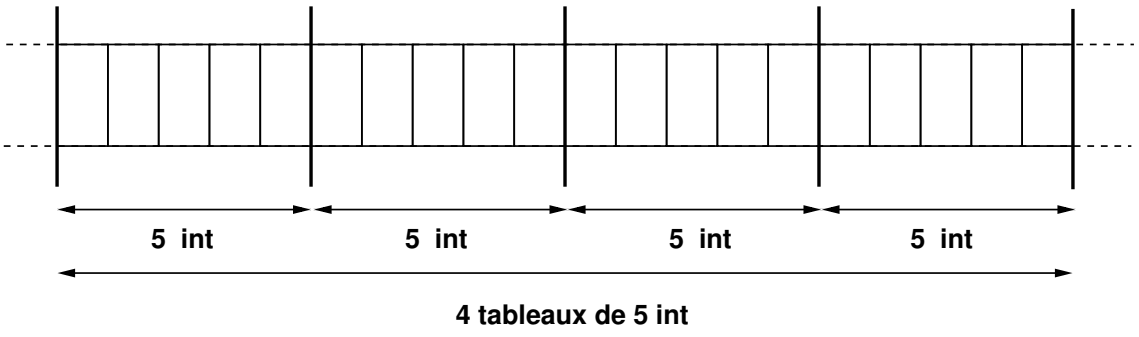
```
mon_type tableau[X][Y];
```

7.4.2 Optimisation du parcours en mémoire

Le fait que le langage C possède un modèle mémoire de type *Row-Major* a une influence sur la façon de concevoir des programmes, notamment lorque des parcours de tableaux multidimensionnels sont effectués. Afin de fixer les idées, nous allons considérer un tableau bidimensionnel d'entiers :

```
int tableau[4][5];
```

La disposition en mémoire est la suivante (une case représente un entier, soit 4 octets) :



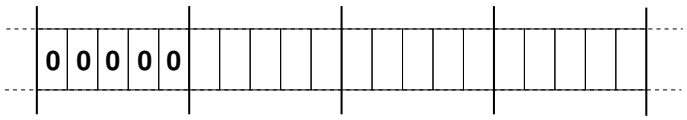
Le parcours le plus efficace du point de vue des accès mémoire est celui où la boucle sur la dimension `Y` est imbriquée dans la boucle sur la dimension `X` :

```
for(int i = 0 ; i < X ; i++)
  for(int j = 0 ; j < Y ; j++)
     tableau[i][j] = 0;
```

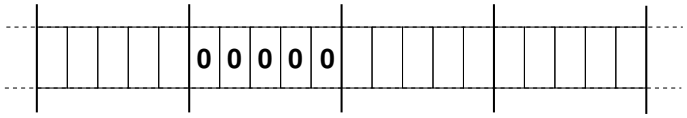
La dimension qui varie le plus rapidement correspond bien à celle dans la boucle la plus interne (boucle sur `Y`). Mais pourquoi ce parcours est-il le plus efficace ? En effet, il est possible d'intervertir les boucles sur `X` et `Y` sans changer le résultat du calcul. Regardons ce qui se passe pour le cas `i = 0` et qu'on parcourt toute la boucle `for` sur `j` :

9. Par opposition à une répartition **Column-Major** ; cela dépend du modèle mémoire du langage. C est *Row-Major*, un langage comme Fortran est *Column-Major*.

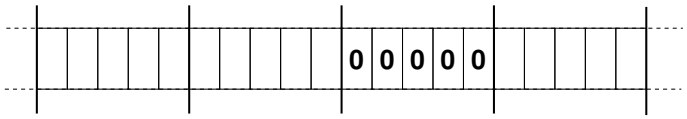
```
i = 0
0 <= j < 5
```



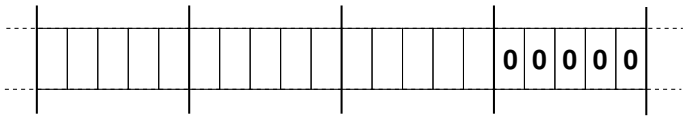

```
i = 1
0 <= j < 5
```




```
i = 2
0 <= j < 5
```




```
i = 3
0 <= j < 5
```

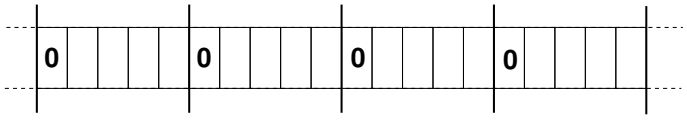


À chaque itération sur la dimension `X`, les accès mémoire se font sur des cases contigües. Si on inverse l'ordre des deux boucles `for` :

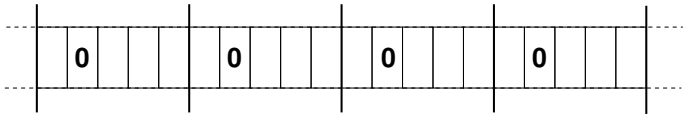
```
for(int j = 0 ; j < Y ; j++)
  for(int i = 0 ; i < X ; i++)
     tableau[i][j] = 0;
```

alors les accès mémoire du programme deviennent :

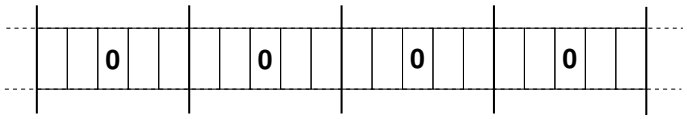
```
j = 0
0 <= i < 4
```



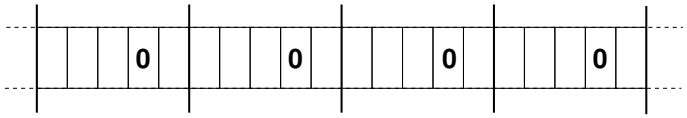

```
j = 1
0 <= i < 4
```



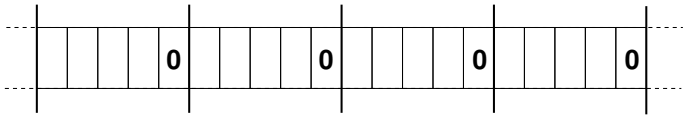
```
j = 2
0 <= i < 4
```




```
j = 3
0 <= i < 4
```




```
j = 4
0 <= i < 4
```



Les accès mémoire sont effectués sur des emplacements non-contigus dans la mémoire. Si la taille des lignes et des colonnes de la matrice est suffisamment grande pour qu'elles ne tiennent pas dans un cache du processeur, un tel parcours va entraîner des défauts de cache à chaque itération tandis que dans le cas précédent, le cache sera mieux utilisé.

7.4.3 Équivalence notationnelle

Nous venons de voir qu'un tableau bidimensionnel est stocké en mémoire dans des cases qui sont toutes contigües, ce qui qui signifie que fondamentalement, il n'y a pas de différence avec un tableau unidimensionnel. En fait, la déclaration de tableaux bidimensionnels est juste une façon de simplifier les calculs d'index pour les parcours des dimensions, car si :

```
int tableau[X][Y];
```

alors :

$$\boxed{\texttt{tableau[i][j]} \iff \texttt{tableau[i*Y + j]}}$$

On constate que seule la première dimension (`X`) n'est pas utilisée pour le calcul d'indice. En conséquence, la déclaration suivante est donc possible :

```
int tableau[ ][Y];
```

Dans le cas particulier où `Y = 1`, alors `j = 0` (car `0 <= j < 1`) et on obtient :

$$\texttt{tableau[i][j]} \iff \texttt{tableau[i*1 + 0]} \iff \texttt{tableau[i]}$$

et donc

$$\texttt{int tableau[X][1]} \iff \texttt{int tableau[X]}$$

On retombe bien sur le cas unidimensionnel.

7.4.4 Généralisation aux dimensions supérieures

L'équivalence notationnelle pour le cas bidimensionnel est généralisable à des dimensions supérieures à 2.

Cas des tableaux à trois dimensions

Par exemple, pour un tableau tridimensionnel, si :

```
int tableau[X][Y][Z];
```

alors :

$$\boxed{\texttt{tableau[i][j][k]} \iff \texttt{tableau[i*Y*Z + j*Z + k]}}$$

Encore une fois, la première dimension (`X`) n'est pas utilisée pour les calculs d'indices. Également, si `Z = 1`, alors `k = 0` (car `0 <= k < 1`) et donc

$$\texttt{tableau[i][j][k]} \iff \texttt{tableau[i*Y*1 + j*1 + 0]} \iff \texttt{tableau[i*Y + j]}$$

et donc

$$\texttt{int tableau[X][Y][1]} \iff \texttt{int tableau[X][Y]}$$

On retombe bien sur le cas bidimensionnel.

Cas général avec n dimensions

On considère un tableau `tableau` à n dimensions dont les valeurs (de ces dimensions) sont $\{D_0, D_1, \ldots, D_{n-1}\}$. On introduit une dimension supplémentaire D_n telle que $D_n = 1$. Ce tableau est donc déclaré de la façon suivante :

$$\texttt{int tableau[}D_0\texttt{][}D_1\texttt{]}\ldots\texttt{[}D_{n-1}\texttt{];}$$

Conformément à ce que nous avons vu dans les cas unidimensionnel et bidimensionnel, si $D_n = 1$, alors cette déclaration est identique à :

$$\texttt{int tableau[}D_0\texttt{][}D_1\texttt{]}\ldots\texttt{[}D_{n-1}\texttt{][}D_n\texttt{];}$$

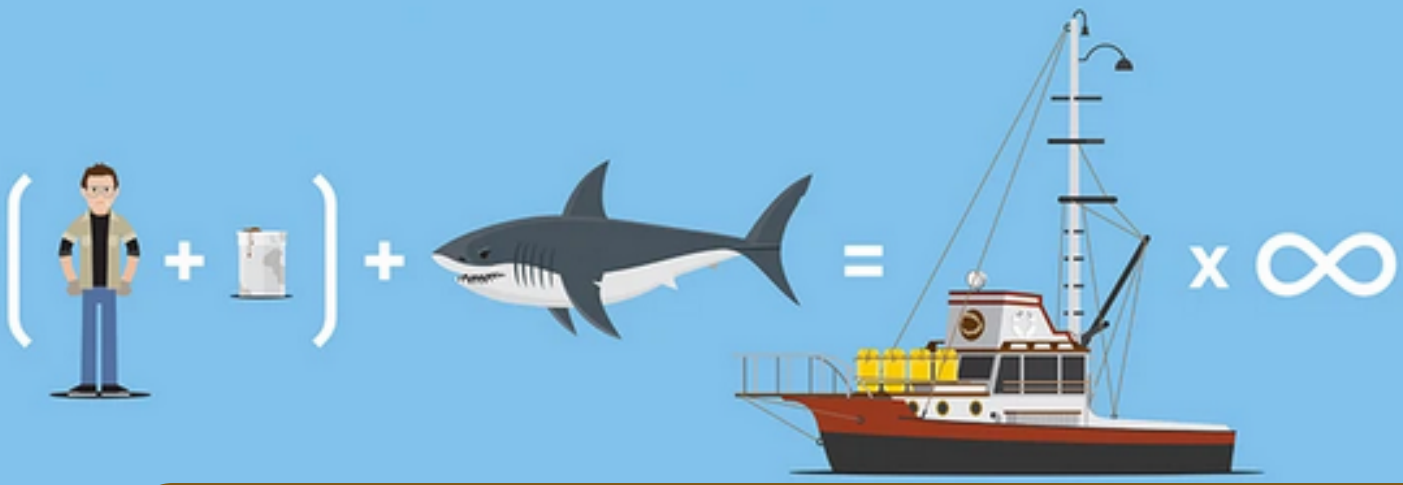
Alors on a les équivalences suivantes :

$$\texttt{tableau[}x_0\texttt{][}x_1\texttt{]}\ldots\texttt{[}x_{n-1}\texttt{][1]} \iff \texttt{tableau[}x_0\texttt{][}x_1\texttt{]}\ldots\texttt{[}x_{n-1}\texttt{]}$$

$$\boxed{\texttt{tableau[}x_0\texttt{][}x_1\texttt{]}\ldots\texttt{[}x_{n-1}\texttt{]} \iff \texttt{tableau[}\textstyle\sum_{i=0}^{n-1} x_i * \prod_{j=i+1}^{n} D_i\texttt{]}}$$

On voit clairement que la dimension D_0 n'est jamais utilisée, comme nous l'avons vu dans les cas à deux et trois dimensions (cf. ci-dessus).

IV La quintessence de C



8. Pointeurs

Le C est un langage avec peu de mots-clefsn modérément sophistiqué au niveau des concepts et constructions syntaxiques qu'il propose. Il est très proche du système (notamment du couple processeur/mémoire) et en ce sens, pourrait être qualifié de "macro-assembleur portable". C permet un contrôle poussé de la mémoire et à ce titre, les pointeurs constituent l'outil principal permettant ce contrôle. Statistiquement, la grosse majorité des erreurs que vous allez commettre dans vos programmes seront des erreurs impliquant des pointeurs. Il est donc essentiel de bien comprendre leur fonctionnement et notamment les quelques subtilités les concernant.

8.1 Notions élémentaires sur les pointeurs

Un pointeur est une **variable** qui stocke une information d'un genre particulier puisqu'il s'agit d'une **adresse-mémoire**. Mais de quoi s'agit-il exactement ?

8.1.1 Quelques éléments d'architecture

Les programmes sont divisés en unités élémentaires, les instructions, qui sont exécutées par le **processeur**. Ce processeur stocke dans de la mémoire qui lui est propre, les**registres**, les instructions machine et les données qu'il doit manipuler. Ces instructions et données sont stockées dans la **mémoire (physique)** de la machine (la RAM) et sont transférées dans les registres du processeur à l'aide d'un élément matériel appelé **bus** (ou bus mémoire). Ces différents élements (processeur et ses registres, bus, mémoire) travaillent sur des objets ayant une taille minimale que l'on appelle des **mots-mémoire** (*memory word*). Par exemple, quand vous entendez parler de processeurs ou de systèmes 64 bits, cela signifie que les registres du processeur sont capables de contenir 64 bits d'information, soit 8 **octets** (*byte*) ou que le bus mémoire est capable en un seul "coup" (on parle de cycle d'horloge) de récupérer 64 bits d'information en mémoire physique pour les transférer dans les registres du processeur. **En résumé, le mot-mémoire est la plus petite quantité d'information pouvant être traitées par la machine.**

La mémoire physique est quant à elle organisée en éléments contigus que l'on appelle des **cases** ou encore des **cellules** (*memory cells*). Afin d'identifier ces différentes cellules pour y stocker les données et instructions machine, on leur attribue un numéro unique, que l'on appelle l'**adresse**. **Ces**

cellules mémoires constituent la plus petite unité de mémoire identifiable par le processeur : ce dernier, à l'aide de leur adresse, est capable de les accéder individuellement.

Cependant, ces cases n'ont pas forcément la même taille que les mots-mémoire. Ainsi sur la plupart des machines actuelles [1], les cases mémoire ont une taille d'un octet (i.e 8 bits), tandis que la taille d'un mot-mémoire est passée progressivement de 8 à 64 bits [2]. **Cela signifie qu'en un seul cycle, le bus est capable de rapatrier vers le processeur plusieurs cases mémoire contigües.**

Merci de ne pas confondre un *bit* avec un *byte* (un octet). Il y a "juste" un facteur 8 de différence, ce qui peut changer des choses quand on calcule des tailles ou des adresses ...

8.1.2 Déclaration de pointeur

Les pointeurs vont donc servir à stocker une information précieuse sur l'emplacement en mémoire des données du programme. Il sera ainsi facile d'indiquer que l'on cherche à manipuler les données qui se trouvent "à tel endroit en mémoire" plutôt que de les désigner explicitement par leurs noms [3]. La déclaration d'une variable de type [4] pointeur se fait avec l'**opérateur unaire** `*` :

identificateur-de-type `*` *qualificateur-de-type*$_{optionnel}$ *identificateur-de-variable*

On dit qu'un pointeur "pointe" sur une variable dont le type correspond au type de déclaration du pointeur, c'est-à-dire que la valeur contenue par le pointeur est (ou sera) l'adresse d'une variable dont le type est identique à celui du pointeur. Par exemple :

`char *ptr` $\Longrightarrow$ `ptr` est un pointeur sur une variable de type caractère `char`
`int *var` $\Longrightarrow$ `var` est un pointeur sur une variable de type entier `int`

Bien entendu, il s'agit là d'expressions. Pour les transformer en instructions effectives, il faut leur accoler le caractère de terminaison d'instruction : `;` Par ailleurs, vous noterez que lors de ces déclarations, les variables pointées ne sont pas encore connues. En fait, la déclaration d'un pointeur d'un certain type signifie que ce pointeur va **potentiellement** nous permettre d'accéder à des variables du même type que celui du pointeur. Cela va dépendre de son utilisation dans le programme. Le type du pointeur peut être précisé davantage par un certain qualificateur de type, qui est optionnel. Ce qualificateur peut être choisi parmi les mots-clefs suivants : `restrict`, `const` ou `volatile` (cf. section 4.1.4 pour revoir leur sens).

8.1.3 Importance du typage des pointeurs

Typer un pointeur n'est pas une chose anodine. En effet, ce typage influence le comportement du pointeur vis-à-vis des opérateurs dont il sera une opérande. De plus, le type (du pointeur) permet de modifier la "vision" que le pointeur possède de l'organisation de la mémoire, lui permettant d'accéder, ou non, à certaines adresses.

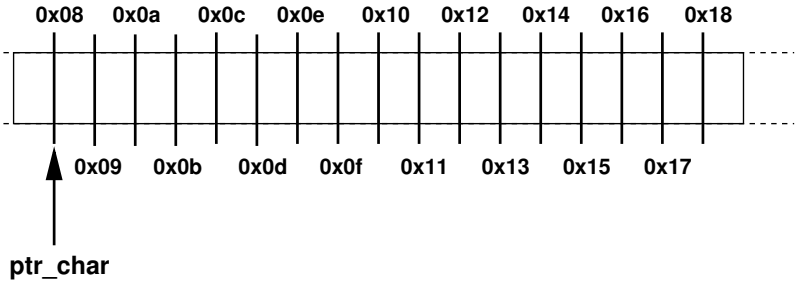
1. depuis le milieu des années 70, à l'exception du Cray C90
2. Sur la plupart des systèmes actuels, un mot-mémoire fait 64 bits, et plus rarement 32.
3. Que l'on ne possède pas toujours d'ailleurs.
4. à prendre comme un synonyme du mot "genre", pas comme un type au sens du langage C.

Exemple des pointeurs de caractères

Soit la définition suivante :

```
char *ptr_char = 0x08;
```

Le pointeur `ptr_char` est de type `char`, ce qui signifie qu'il peut être utilisé pour référencer des octets individuels en mémoire (un caractère est codé sur 8 bits, soit un octet : `sizeof(char)`= 1). Ce pointeur "voit" la mémoire ainsi :



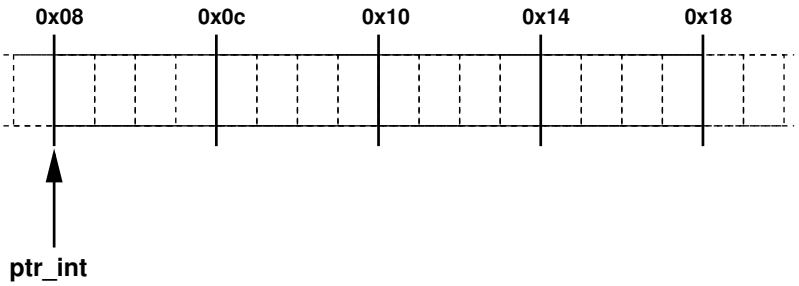
Chaque case correspond à un octet. Comme l'octet est la plus petite unité adressable, chaque octet possède donc une adresse qui lui est propre.

Exemple des pointeurs d'entiers

Soit un pointeur d'entiers, de type `int`, initialisé de la même façon que précédemment :

```
int *ptr_int = 0x08;
```

Le pointeur `ptr_int` peut être utilisé pour référencer des entiers, qui sont codés sur 32 bits, soit 4 octets (i.e `sizeof(int)`= 4). Dans ce cas, tout se passe comme si le pointeur "voyait" la mémoire organisée de la façon suivante :



Pour ce pointeur, les seules unités accessibles sont les entiers, donc des "cases" de 4 octets (= `sizeof(int)`). Il ne (re)connait pas les véritables cases mémoire individuelles (i.e les octets, qui existent toujours physiquement et sont indiqués en pointillés sur le schéma). Le compilateur calcule les adresses des donnés de type `int` de façon à ce qu'elles soient toujours des multiples de le taille de ce type (i.e 4, soit `sizeof(int)`). C'est ce qu'on appelle l'**alignement mémoire**. Cet alignement est mis en place pour des questions de performance ; certains systèmes acceptent que les données puissent ne pas être alignées [5], d'autres le refusent.

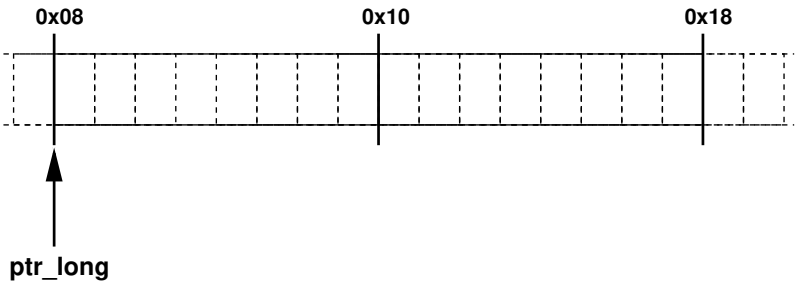
5. Par exemple en déréférençant un pointeur de type *type_t* dont la valeur d'adresse n'est pas un multiple de `sizeof(`*type_t*`)`.

Exemple des pointeurs d'entiers longs

La situation est similaire avec un pointeur de type `long int` :

```
long int *ptr_long = 0x08;
```

Comme les entiers longs sont codés sur 64 bits, c'est-à-dire 8 octets(8 = `sizeof(long int)`), le pointeur `ptr_long` ne reconnait que des "cases" de 8 octets et l'alignement est effectif sur un taille de 8 octets (i.e les adresses des entiers longs sont forcément des multiples de 8) :



Là encore, les octets individuels (cases de la mémoire physique) existent toujours physiquement mais ne sont pas naturellement accessibles en tant que tel par le pointeur `ptr_long`, sauf à malmener l'alignement calculé par le compilateur.

++ Conséquence de l'alignement mémoire sur les adresses des variables

Ainsi, il devient clair que les variables ne sont pas disposées n'importe comment en mémoire : elles sont alignées sur des adresses particulières, c'est-à-dire que ces adresses possèdent une valeur spécifique qui dépend du type de la variable.

De façon générale, une variable *variable* de type *type_t* est alignée si l'expression suivante est vraie :

```
(((size_t) &variable) % sizeof(variable)) == 0
```

Pointeurs particuliers

Il existe deux sortes de pointeurs un peu particuliers :

— les pointeurs de type `char` : un caractère étant codé sur un octet, il faut comprendre que des pointeurs sur des variables caractères correspondent souvent à des pointeurs sur des octets quelconques. Les données sur lesquelles ils pointent ne seront pas forcément interprétées comme étant des caractères. Les pointeurs de ce type permettent de référencer n'importe quelle partie de la mémoire car l'octet est la plus petite unité de mémoire adressable par le système.

— les pointeurs de type `void` : ils sont appelés **pointeurs génériques** . Une variable de ce type indique juste que l'on va utiliser un pointeur sans en connaître le type exact au moment de sa déclaration. Ce typage interviendra plus tard dans le programme à l'aide de l'opérateur de conversion (ou *cast*, `()` unaire). Un abus de langage consiste à dire que ces pointeurs ne "pointent sur rien" [6].

Un pointeur générique ne reconnait aucune sorte de cases puisqu'il ne possède pas de type déterminé.

6. Historiquement les pointeurs génériques étaient de type `char`, mais il était facile de confondre pointeur générique et pointeur d'octet. C89 a introduit le mot-clef `void` afin de mettre fin à cette confusion. Et dans ce cours, qui dit confusion, dit contusion.

8.1.4 Informations véhiculées par les pointeurs

Un point essentiel à comprendre est qu'en réalité **un pointeur ne véhicule pas une, mais deux informations** afin de manipuler la mémoire :

1. **une information d'adresse** : cette information est véhiculée par la **valeur** du pointeur ;
2. **une information de taille** des données accessibles : cette information est véhiculée par le **type** du pointeur.

Cette information de taille est indispensable pour pouvoir effectuer des opérations comme le **déréférencement** (opérateur unaire `*`, cf. ci-dessous) ou encore des **calculs d'adresses** avec un pointeur (cf. section 8.2). **En particulier, un pointeur générique (de type `void`) ne possède pas cette information de taille et ne peut être ni utilisé pour des calculs d'adresses, ni déréférencé.**

Il est courant de faire le raccourci *pointeur = adresse* mais en faisant un tel raccourci, on perd un aspect essentiel de ce qu'est un pointeur. Ainsi, il est fondamental de bien réfléchir au typage des pointeurs que l'on va utiliser dans les programmes. Un typage utilisant la technique du doigt mouillé ou du pifomètre total a peu de chance de déboucher sur un programme correct. Cette information de taille ne doit pas être confondue avec la taille de la variable pointeur elle-même. **Un pointeur, quel que soit son type, possède toujours la même taille sur un système donné**. Par exemple, sur un système 64 bits, un pointeur sera stocké sur 8 octets (`sizeof(`*type_quelconque* `*)` $= 8$), i.e 64 bits, qu'il pointe sur un entier (`sizeof(int)` $= 4$) ou caractère (`sizeof(char)` $= 1$).

Il existe une constante spécifique aux pointeurs : `NULL`. Il s'agit d'une constante qui à pour valeur 0 (adresse `0x000000000000`) et qui sert notamment à l'initialisation des pointeurs. Tout accès à cette adresse se solde automatiquement par un échec de l'exécution du programme [7]. En C, les variables n'étant pas initialisées par défaut à la déclaration, un pointeur peut donc contenir une adresse quelconque. Il est conseillé de systématiquement initialiser un pointeur avec la constante `NULL` (sauf s'il est possible de lui assigner une autre valeur à ce moment-là).

Attention de ne pas confondre la constante `NULL` qui est une valeur, avec `void` qui est un type (plus ou moins).

8.1.5 Le pointeur, une variable commme les autres

En dépit de ce que l'on pourrait croire, les pointeurs sont des variables classiques. Elle ne possèdent pas de propriétés spécifiques. En particulier, tout ce que nous avons déjà vu concernant les variables s'applique également aux pointeurs, à savoir :

— il est possible de créer des tableaux de pointeurs ;
— les pointeurs peuvent être passés en tant qu'arguments à des fonctions ;
— les pointeurs sont un type de retour licite dans les fonctions ;
— les pointeurs peuvent être déclarés en tant que champ de structure (cf. Chapitre 9) ;
— un pointeur peut pointer sur un pointeur [8] !

En revanche, les pointeurs obéissent à des règles d'arithmétique spéciales et tous les opérateurs ne sont pas utilisables (cf. section 8.2 ci-dessous). Le compilateur générera une erreur en cas d'utilisation d'une opération non autorisée sur un pointeur.

7. C'est comme une division par 0 en mathématiques : ça ne marche pas trop bien.
8. Si, si. Je pense que vous sentez venir le problème, là. Et les ennuis sont loin d'être finis ...

8.1.6 Opérations spécifiques aux pointeurs

Nous allons maintenant examiner les opérations spécfifiques aux pointeurs, qui donnent tout leur sel à la chose. Ces opérations sont effectuées via les opérateurs suivants :

— l'opérateur de récupération de pointeur `&`
— l'opérateur de déréférencement (ou d'indirection)`*`
— l'opérateur de conversion `()`

Ces opérateurs ont une associativité de **droite à gauche** et sont **unaires**, ce qui permet de ne pas les confondre avec les opérateurs du ET bit-à-bit et de la multiplication pour les deux premiers, respectivement. Ces opérateurs unaires de pointeurs sont par ailleurs beaucoup plus prioritaires que leurs versions binaires. Quant à la conversion que nous avons déjà rencontrée en section 4.1.3, elle n'est pas l'apanage des pointeurs mais est très souvent utilisée avec eux.

Récupération d'adresse

L'opérateur `&` permet de récupérer un pointeur (et non pas *juste* une adresse) sur une variable quelconque. Le pointeur ainsi récupéré est du type de la variable sur laquelle l'opérateur a été appliqué. Ainsi, si la déclaration suivante est faite :

type_t `variable;`

alors l'initialisation suivante est licite :

type_t `*pointeur = &variable;`

car `&variable` est un pointeur de type *type_t*, ce qui est cohérent avec la déclaration du type du pointeur `pointeur`.

Il n'est pas possible de récupérer un pointeur sur une variable dont la classe de stockage est `register`, car cette variable n'est pas forcément en mémoire donc potentiellement sans adresse.

Par abus de langage, on dit souvent que l'opérateur `&` permet de récupérer l'adresse d'une variable, ce qui ne recouvre qu'une partie de la réalité. En effet, le fait de récupérer un pointeur entraîne que l'information de taille est également obtenue, et pas uniquement l'adresse de la variable.

Déréférencement

Un pointeur est un moyen de manipuler une donnée via son emplacement en mémoire (emplacement accédé par le truchement de l'adresse). **Déréférencer un pointeur consiste à récupérer le contenu de la "case mémoire" sur laquelle il pointe** [9]**.** On obtient alors une variable du même type que le pointeur ayant servi au déréférencement. Formellement, c'est l'opérateur unaire `*` qui permet ce déréférencement. Donc, si :

type_t `* pointeur;` *où l'on suppose que la valeur du pointeur est l'adresse* $@_{xyz}$

alors il est possible de récupérer les données qui sont à l'adresse $@_{xyz}$ de cette façon :

type_t `resultat = *pointeur;`

`*pointeur` est alors une variable de type *type_t* (car c'est aussi celui de `pointeur`). Il est donc possible de l'affecter à une autre variable de ce type, comme `resultat`. Toutes les opérations sur les variables "classiques" sont donc possibles avec la variable `*pointeur` : lecture bien entendu mais également écriture. Le type de `*pointeur` ne dépend pas de celui de la donnée originale stockée à l'adresse $@_{xyz}$, mais uniquement de celui du pointeur opérande de l'opérateur `*`.

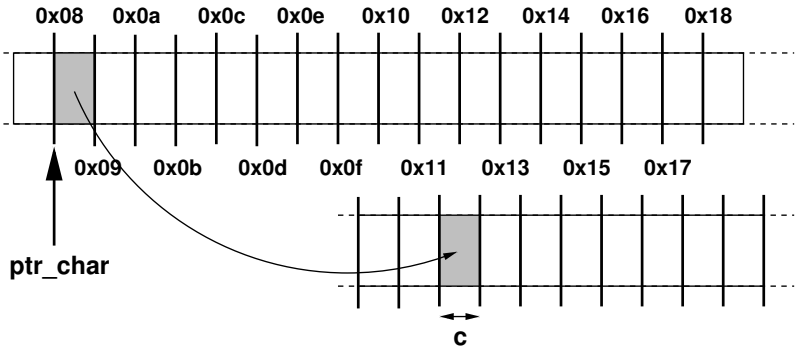
9. Je mets des guillemets car cette "case mémoire" correspond à celle vue par le pointeur (cf ci-dessus) et non pas à celles de la mémoire physique, qui sont les octets dans tous les cas.

En effet, c'est ce pointeur qui possède l'information sur la taille de la "case mémoire" qu'il faut récupérer (c'est-à-dire le nombre d'octets que l'opérateur `*` doit aller chercher) et cette information, comme nous l'avons déjà vu, est véhiculée par son type. Sans cette information de taille, il est impossible de récupérer des données, ce qui précise un peu plus ce que nous avions écrit en section 8.1.4. Dès lors, vous comprenez mieux les limitations d'un pointeur générique : comme il ne possède pas de type précis, alors il ne possède pas d'information concrète de taille et il n'est donc pas déréférençable.

Reprenons les exemples précédents ; nous supposons que la mémoire pointée est accessible et donc que le déréférencement ne produira pas d'erreur à la lecture. Nous allons d'abord déréférencer le pointeur de caractères `ptr_char` et stocker la donnée récupérée dans une autre variable `c`, de type caractère (`char`) :

```
char *ptr_char = 0x08;
char  c        = *ptr_char;
```

Il se passe alors la chose suivante :

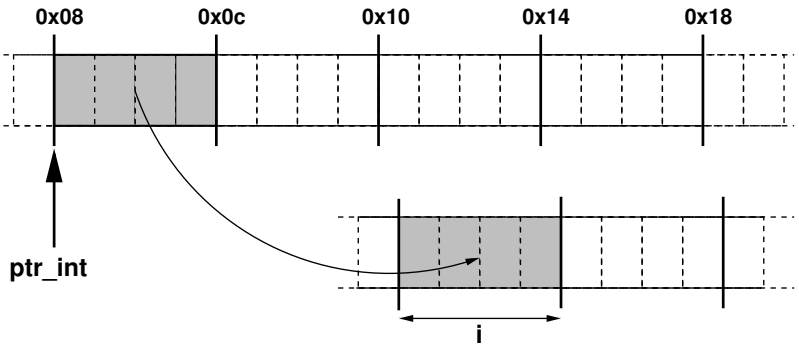


Pour un pointeur de type `char`, une "case" correspond exactement à une case mémoire de la mémoire physique, soit un octet. L'octet récupéré lors du déréférencement est alors **copié** dans la variable `c` (peu importe son emplacement).

Prenons maintenant le cas du pointeur d'entier `ptr_int` :

```
int  *ptr_int  = 0x08;
int   i        = *ptr_int;
```

L'exécution de ce code peut être représentée par le schéma suivant :



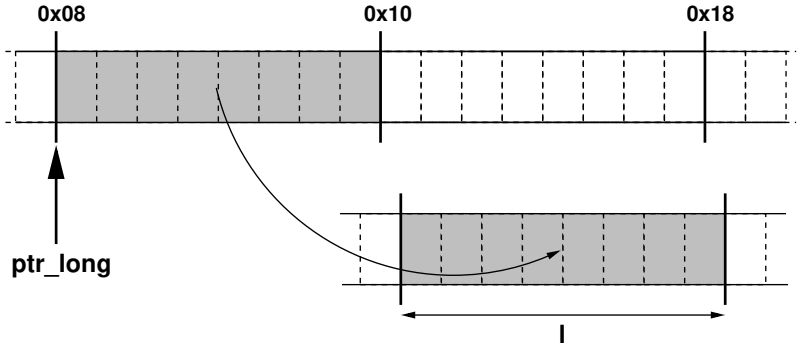
En effet, pour un pointeur de type `int`, une "case" correspond à un entier, soit 4 octets (car `sizeof(int)` = 4), ce qui est plus grand que la taille d'une case mémoire physique (toujours un

octet). Le contenu de cette “case” sera copié dans la variable `i` (peu importe son emplacement).

Enfin dans le cas du pointeur d'entier long `ptr_long`, le code est identique, seul le typage est modifié :

```
long    *ptr_long = 0x08;
long int l        = *ptr_long;
```

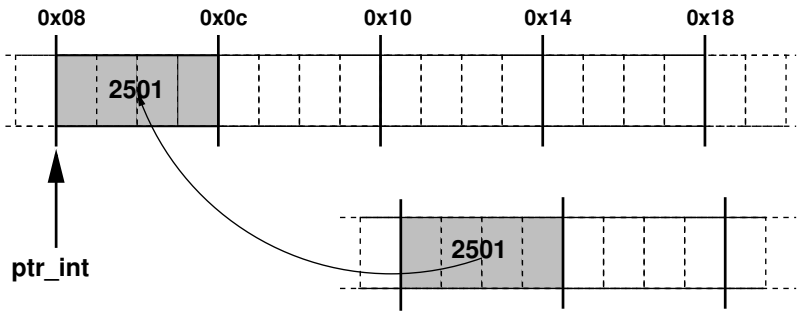
Et cela donne :



Pour un pointeur de type `long int`, une “case” correspond à un entier long soit 8 octets (car `sizeof(long int)`$= 8$). Le contenu de la “case” pointée sera copié dans la variable `l` (peu importe son emplacement). Dans les trois cas, la taille des données étant inférieure ou égale à celle d'un mot-mémoire (64 bits, donc 8 octets), le transfert sera effectué en un seul cycle.

Tous ces exemples illustrent comment l'opérateur de déréférencement peut être utilisé pour lire des données en mémoire. Effectuer une écriture est évidemment possible comme nous l'avons déjà indiqué en début de section et se fait de la façon suivante :

```
int *ptr_int = 0x08;
*ptr_int = 2501;
```



La valeur du pointeur (ie l'adresse sur laquelle il pointe) n'est pas modifiée par cette opération.

Cette opération de déréférencement agit comme l'inverse de la récupération de pointeur. Formellement :

$$*\&\ variable \equiv variable$$

et ce, quel que soit le type de la variable considérée.

L'équivalence suivante est également valable pour les pointeurs de type quelconque (sauf les génériques) :

$$\boxed{\&* \textit{pointeur} \equiv \textit{pointeur}}$$

Enfin, précisons le vocabulaire : j'utilise le terme "déréférencement" ici mais les concepts de pointeur et de référence, bien que très proches, sont sensiblement différents. Certains langages possèdent des pointeurs (C), d'autres des références (Caml) et d'autres encore disposent des deux (C++). Une différence importante est, par exemple, l'impossibilité de faire des opérations arithmétiques (cf. section 8.2) avec des références.

Conversion

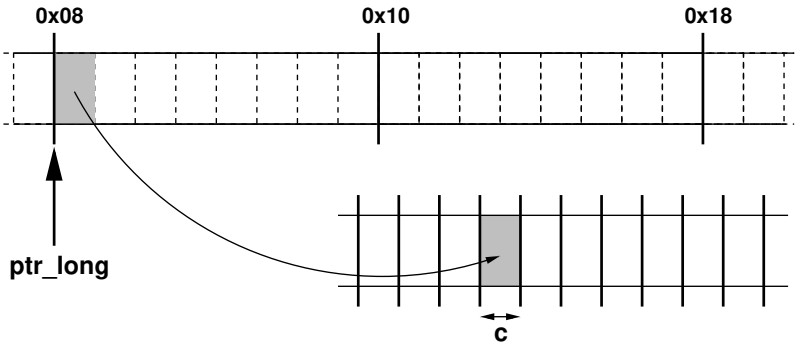
Dans les exemples de la section 8.1.3, j'ai utilisé des pointeurs de type différents, mais possédant la même valeur (i.e la même adresse). Plutôt que d'utiliser trois fois la même valeur explicitement, j'aurais pu réutiliser le premier pointeur `ptr_char` initialisé avec cette adresse et l'affecter aux deux autres pointeurs `ptr_int` et `ptr_long`. Cependant, ces pointeurs étant de types différents, il faudrait procéder à des conversions. À la différence des variables classiques, cette conversion est obligatoirement explicite et s'appuie sur l'opérateur `()`. Voici ce que cela donnerait en pratique :

```
char     *ptr_char = 0x08;
int      *ptr_int  = (int *)ptr_char;
long int *ptr_long = (long int *)ptr_char;
```

Examinons un exemple un peu plus compliqué, qui mélange conversion et déréférencement : Nous avons un pointeur d'entier long `ptr_long` que l'on veut déréférencer, mais le résultat est plus grand que la variable qui doit le stocker (1 octet à la place de 8). La solution est donc de changer le type du pointeur afin que l'opération de déréférencement retourne des données de la bonne taille :

```
long int *ptr_long = 0x08;
char      c        = *((char *)ptr_long);
```

Il se passe alors la chose suivante :



Seul le premier octet est récupéré lors du déréférencement.

Plutôt que de convertir explicitement le pointeur, nous aurions pu compter sur une conversion implicte de la variable obtenue au moment du déréférencement pour aboutir à un résultat identique :

```
long int *ptr_long = 0x08;
char      c        = *ptr_long;
```

Lors du déréférencement la variable `*ptr_long`, de type `long int` est convertie implicitement en un caractère (type `char`), ce qui évite un écrasement de la mémoire qui suit le caractère `c`. Notez

toutefois que si le résultat est identique, il ne se passe pas la même chose que dans le cas précédent. Ici, 8 octets sont effectivement récupérés lors du déréférencement mais seul le premier est conservé lors de la conversion implicite.

Une opération de conversion de pointeur (*cast*) ne **change pas la valeur** du pointeur, mais uniquement son type. Elle ne modifie pas non plus la variable pointée.

Attention à ne pas vous tromper de type lors de l'opération de conversion (ce que l'on pourrait appeler une *erreur de casting*).

Une autre erreur *manifeste* de casting.

8.2 Arithmétique pointeur

Les pointeurs sont fréquemment utilisés pour effectuer des calculs d'adresse, ce qui nous allons appeler de l'arithmétique pointeur. Une fois qu'une adresse est déterminée, il est alors possible de lire/écrire des données depuis/vers l'adresse calculée.

Le C permet d'accéder en théorie à l'intégralité de la mémoire. En pratique, des zones sont inaccessibles en lecture et/ou écriture et nécessitent d'être **allouées** (cf. section 8.5 ci-dessous). Un accès à une zone encore non allouée se soldera par un échec à l'exécution [10]. Les erreurs les plus fréquentes que vous allez être amenés à commettre sont des accès illicites à des zones mémoires non autorisées suite à un calcul d'adresse erroné.

8.2.1 Addition d'un entier et d'un pointeur

La première (et la plus importante) des opérations est celle qui consiste à additionner une valeur entière à un pointeur. La somme d'un pointeur de type *type_t* et d'un entier (dont le type importe peu : `char`, `short`, `int`, `long int` ou `long long int`, signé ou non) est un nouveau pointeur, dont le type est toujours *type_t* mais dont **la valeur (c'est-à-dire l'adresse) dépend non seulement de la valeur de l'entier additionné mais aussi du type *type_t*.** Regardons ce phénomène plus en détail en considérant un pointeur `ptr` de type *type_t* :

type_t `* ptr;`

Par définition, `ptr` pointe sur un élément en mémoire de type *type_t* dont l'adresse est un multiple de la taille de *type_t* (cf. section 8.1.3). La notation formelle $(\texttt{ptr})_{type\_t}$ désigne un tel pointeur dans la suite de cette section.

10. Le fameux `segmentation fault` ...

Il n'y a que le premier pas qui coûte...

Définissons `(ptr+1)` comme un nouveau pointeur, toujours de type *type_t* et qui contient l'adresse de l'élément **suivant immédiatement** celui pointé par `ptr`. Une telle définition est assez naturelle étant donné que les seules adresses contenues par des pointeurs de type *type_t* sont celles des éléments de ce type, justement. Ce pointeur est défini ainsi :

$$(\texttt{ptr + 1})_{type\_t} \equiv (type\_t\texttt{*})\texttt{ptr + (1)}_{type\_t}$$

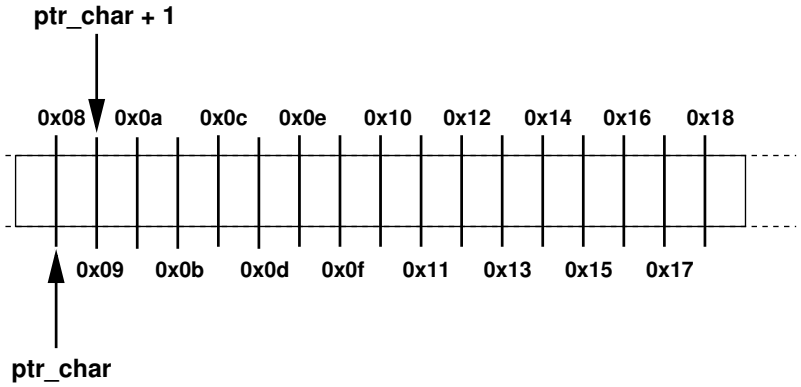
Calculons maintenant la valeur **en octets** de $(\texttt{ptr + 1})_{type\_t}$. En effet, les octets (de type `char`) étant les plus petites unités adressables par le processeur, ce sont leurs adresses qui sont utilisées en pratique par le compilateur pour les calculs de valeurs de pointeurs et d'arithmétique. Par définition, un décalage d'un élément de type `type_t` est égal à un décalage de `sizeof(`*type_t*`)` octets (i.e $(\texttt{1})_{type\_t} \equiv (\texttt{sizeof(}type\_t\texttt{)})$). De plus, la valeur d'un pointeur (i.e l'adresse qu'il contient) est la même quel que soit son type. D'où :

$$\boxed{(\texttt{ptr + 1})_{type\_t} \equiv \texttt{(char *)ptr + sizeof(}type\_t\texttt{)}}$$

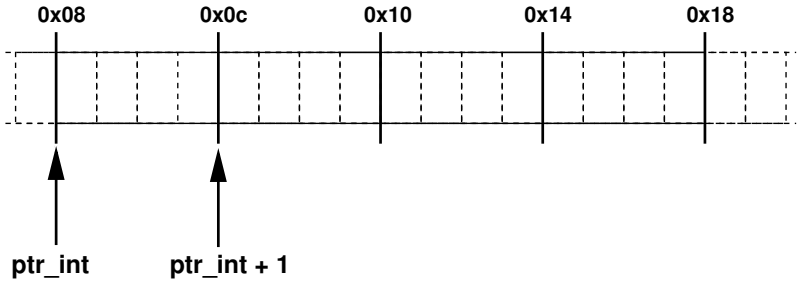
Donc pour une même opération (`+1`), le résultat est différent selon le (type du) pointeur de départ. **C'est donc le type du pointeur qui détermine le comportement de l'opération et son résultat.** Reprenons les pointeurs des exemples précédents :

```
char     *ptr_char = 0x08;
int      *ptr_int  = (int *)ptr;
long int *ptr_long = (long int *)ptr;
```

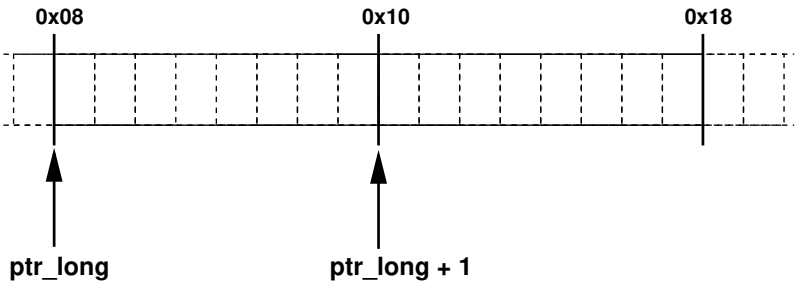
alors les schémas suivants représentent les situations pour `(ptr_char+1)`, `(ptr_int+1)` et `(ptr_long+1)` :



L'adresse en octets `(ptr_char+1)` est bien égale à ... `(ptr_char+1)` (elle-même).



L'adresse en octets `(ptr_int+1)` est bien égale à `(ptr_int+4)` car `sizeof(int)`$= 4$.



L'adresse en octets `(ptr_long+1)` est bien égale à `(ptr_long+8)` car `sizeof(long int)`$= 8$.

Généralisation de l'opération d'addition

Maintenant que nous avons compris le fonctionnement de l'incrémentation de pointeur, il devient trivial de généraliser le principe à n'importe quelle valeur d'entier (signé ou non) :

$$\texttt{(ptr + i)}_{type\_t} \equiv \texttt{(char *)ptr + i*sizeof(}type\_t\texttt{)}$$

Étant donné que `sizeof(void)` vaut 0, on comprend mieux pourquoi il est impossible de faire des opérations d'addition avec des pointeurs génériques (type `(void *)`).

Vous n'avez **jamais** à préciser le `sizeof` quand vous effectuez des calculs d'adresses. Ce facteur multiplicateur est automatiquement appliqué par le compilateur dès que vous précisez le type du pointeur (à la déclaration, ou avec une conversion explicite par exemple). D'où l'importance d'un typage correct de vos pointeurs. Vous comprenez également pourquoi le compilateur va se plaindre quand vous *essayez* d'effectuer des calculs arithmétiques avec des pointeurs de types différents [11] : cela n'a aucun sens !

8.2.2 ++ Différence de pointeurs

Une autre opération importante en arithmétique pointeur est la différence de deux pointeurs **du même type**. Soit :

```
type_t *ptr;
type_t *ptr_offset;
```

tels que :

```
ptr_offset = (ptr + i); /* affectation en C !*/
```

Si nous faisons la différence de `ptr_offset` et `ptr` :

$$\texttt{(ptr\_offset)}_{type\_t} \texttt{ - (ptr)}_{type\_t} \equiv \texttt{(ptr + i)}_{type\_t} \texttt{ - (ptr)}_{type\_t}$$
$$\texttt{(ptr\_offset)}_{type\_t} \texttt{ - (ptr)}_{type\_t} \equiv \texttt{(ptr)}_{type\_t} \texttt{ + (i)}_{type\_t} \texttt{ - (ptr)}_{type\_t}$$

Soit

$$\texttt{(ptr\_offset)}_{type\_t} \texttt{ - (ptr)}_{type\_t} \equiv \texttt{(i)}_{type\_t}$$

`i` est donc bien un **décalage** entre l'élément pointé par `ptr_offset` et l'élément de départ pointé par `ptr`. Donc, `i` représente **un nombre d'éléments de type *type_t* et non pas un nombre d'octets (sauf si *type_t* est `char`).** À nouveau le compilateur n'effectue ses calculs qu'avec des

11. À l'école, on a dû vous apprendre qu'il n'était pas possible d'additionner des pommes et des carottes. C'est rigoureusement la *même chose* ici.

valeurs et des adresses en octets. Ainsi le calcul *effectif* du décalage `i` est effectué avec les conversions (cast) suivantes [12] :

`(char *)ptr_offset-(char *)ptr` $\equiv$ `(char *)ptr + i*sizeof(`*type_t*`)-(char *)ptr`

Soit :

$$\texttt{i} \equiv \frac{\texttt{(char *)ptr\_offset - (char *)ptr}}{\texttt{sizeof(}\textit{type\_t}\texttt{)}}$$

Il est donc impossible d'effectuer une différence de pointeurs génériques : comme `sizeof(void)` vaut 0, cette opération est mathématiquement problématique.

La valeur obtenue lors d'une soustraction de pointeurs possède même un type particulier en C : il s'agit du type `ptrdiff_t` [13]. La section 7.1.3 concernant les types pour les indices de parcours de tableaux devrait être plus claire pour vous désormais. En effet, comme son nom ne l'indique pas, `ptrdiff_t` est bien un type employé pour stocker un **nombre d'éléments**.

8.2.3 Autres opérations

Outre l'addition d'un entier (signé) et la différence de deux pointeurs de même type entre eux, les autres opérations possibles avec les pointeurs sont relativement limitées :

— l'assignation de pointeurs du même type ;
— l'assignation et la comparaison d'un pointeur de n'importe quel type avec la valeur entière `NULL` (0) ;
— la comparaison de pointeurs sur deux membres d'un même tableau.

En particulier les opérations suivantes ne sont **pas** permises :

— assigner un pointeur d'un type sans le convertir (*cast*) à un pointeur d'un autre type (exception : les pointeurs génériques `(void *)`) ;
— additionner deux pointeurs y compris de même type ;
— additionner un nombre flottant à un pointeur ;
— utiliser la multiplication avec un pointeur ;
— utiliser la division avec un pointeur ;
— utiliser les opérateurs de décalage >> et << sur un pointeur.

8.3 Pointeurs et tableaux

Les pointeurs et les tableaux (cf. Chapitre 7) sont en réalité liés et nous allons voir que des données stockées dans un tableau peuvent être accédées par le biais de pointeurs. Avec le temps, vous allez de plus en plus utiliser les pointeurs plutôt que les tableaux, même si ces derniers peuvent encore se révéler utiles pour simplifier parfois des notations trop complexes dans vos codes. L'emploi de la notation pointeur à la place de la notation tableaux reste *in fine* une affaire de goût.

8.3.1 Tableaux, pointeurs : même combat ?

Dans le Chapitre 7, nous avons que lorsqu'un tableau est déclaré tel que le suivant :

type_t `tab[SIZE_TAB];`

12. Convertir un pointeur revient en fait à changer l'unité (ou la base, au sens mathématique du terme) dans laquelle les calculs avec lui sont effectués. Ensuite, il serait incohérent d'utiliser dans un calcul des termes exprimés dans des unités (ou des bases) différentes, d'ou la conversion ici de **tous** les termes dans la seule unité que le système utilise, à savoir les octets (`char`).

13. Il faut inclure le fichier `<stddef.h>` pour avoir cette définition de type utilisable dans les programmes.

alors `tab[i]` est une variable de type *type_t*, qui est l'élément numéro `i` de ce tableau [14]. Il s'agit donc d'un **décalage** par rapport à l'élément 0 de `tab`.

Déclarons maintenant un pointeur `ptr_tab` de type *type_t* qui pointe sur le premier élément du tableau `tab`, soit :

```
type_t *ptr_tab = &tab[0];
```

Additionnons un entier `i` au pointeur `ptr_tab` (`(ptr_tab +i)`) : d'après la section 8.2.1, cet entier `i` est également un décalage de `i` éléments de type *type_t* par rapport à l'élément pointé par `ptr_tab`. qui est le premier du tableau `tab`. D'où l'équivalence :

$$\texttt{*(ptr\_tab + i)} \equiv \texttt{tab[i]}$$

Quand on utilise l'opérateur de récupération de pointeur sur les deux parties de l'équivalence, on obtient :

$$\texttt{(ptr\_tab + i)} \equiv \texttt{\&tab[i]}$$

Dans le cas particulier où `i`$= 0$, on retrouve bien l'initialisation du pointeur `ptr_tab`.
Par convention, l'adresse du premier élément d'un tableau est le nom de ce tableau, ce qui se traduit par :

$$\boxed{\texttt{\&tab[0]} \equiv \texttt{tab}}$$

Ce qui nous donne :

$$\texttt{ptr\_tab} \equiv \texttt{tab}$$

et donc :

$$\texttt{*(ptr\_tab + i)} \equiv \texttt{*(tab + i)}$$

Équivalence notationnelle

Un tableau étant une sorte de pointeur, un accès à un élément du tableau peut être effectué par le biais de la "notation tableau", c'est-à-dire à l'aide de l'opérateur `[]` ou via la "notation pointeur". En effet, comme :

$$\texttt{*(ptr\_tab + i)} \equiv \texttt{tab[i]} \quad \text{et} \quad \texttt{ptr\_tab} \equiv \texttt{tab}$$

Alors :

$$\boxed{\texttt{*(tab + i)} \equiv \texttt{tab[i]}}$$

Il y a donc une équivalence notationnelle entre tableaux et pointeurs.

14. Et non pas le i-ème élément puisque la numérotation commence à zéro.

Accès à un tableau avec un pointeur

Cette équivalence est intéressante et très utilisée en pratique. Il n'est cependant pas nécessaire de *systématiquement* remplacer la notation tableau par la notation pointeur car cela peut parfois alourdir le code et nuire à sa lisibilité. Les circonstances dictent souvent quelle notation est la plus pertinente. Cette équivalence fonctionne dans le sens réciproque : quand vous avez un pointeur sur des données, ces dernières peuvent à leur tour être accédées via une notation tableau.

En revanche, un pointeur sur un tableau ne pointe **que sur le premier élément** de ce tableau. En particulier, ce pointeur ne possède pas d'information sur le nombre d'éléments que ce tableau contient. Donc, si vous souhaitez parcourir le tableau vous aurez besoin de sa taille, qu'il vous faut obtenir par ailleurs.

8.3.2 Les tableaux, des pointeurs comme les autres.

Pas tout à fait. Le nom du tableau est certes un pointeur comme nous venons de le déterminer, mais ce **pointeur est une constante**, qui n'est donc pas susceptible d'être modifiée [15]. Dans l'exemple de la section précédente, si `tab` devait être déclaré en tant que pointeur, cela donnerait :

```
type_t * const tab;
```

Notez bien la présence du mot-clef `const` entre l'opérateur `*` et l'identificateur du tableau. Les conséquences sont multiples :

— un tel pointeur ne peut pas être affecté ;
— l'arithmétique avec un tel pointeur est limitée.

L'impossibilité de l'affectation répond donc à la question de l'impossibilité de copier un tableau directement dans un autre avec l'égalité. Si A et B sont deux tableaux, par exemple :

```
int A[12];
int B[12];
```

Alors `A = B` n'est pas possible car `A`, qui est une constante, ne peut pas voir sa valeur modifiée [16].

Cependant, si les pointeurs et les tableaux sont liés et qu'il y a un équivalence en termes de notation (`*(tab +i)` ≡ `tab[i]`), il n'y pas une telle équivalence pour la déclaration et/ou l'initialisation. Regardons les exemples suivants :

```
int tab[]      = {1,2,3,4}; /* possible   */
int *tab_ptr   = {1,2,3,4}; /* impossible */
```

`tab` est un tableau et quand cette variable est déclarée, un certain nombre de cases mémoires sont réservées. Il est possible de copier dans ces cases des données. Le tableau agit bien comme un vecteur de données. En revanche, pour `tab_ptr`, la situation est différente car non seulement l'espace mémoire alloué est insuffisant (seule la mémoire pour stocker le pointeur lui-même est allouée, cf section 8.5.2), mais de plus un pointeur est une variable scalaire qui ne peut pas contenir de vecteur de données.

15. Merci, Captain Obvious !

16. On remarquera que de toute façon, même si `A` était affectable, l'opération `A = B` ne copierait pas les données du tableau `B` dans le tableau `A`, mais ferait que `A` pointerait sur les données de `B` (aliasing de pointeurs). Toute modification sur les données de `B` serait répercutée sur `A`, ce qui n'arriverait pas en cas de copie, car les données des tableaux seraient alors mutuellement indépendantes.

8.3.3 Les tableaux comme type de retour

Comme nous l'avons vu dans la section 6.3.2 et au debut du chapitre 7, les tableaux ne sont pas des types de retour licites dans les fonctions. En revanche, les pointeurs, sont eux, totalement acceptables en tant que type de retour. Donc, plutôt que de retourner un tableau complet, une fonction pourra retourner un **pointeur** sur le premier élément de ce tableau à la place. l'accès aux divers éléments de ce tableau pourra se faire avec la notation pointeur ou la notation tableau de façon indifférenciée.

Quand une fonction retourne un pointeur, et que ce dernier est interprété comme pointant sur un tableau [17], n'oubliez pas qu'il pointe uniquement sur le **premier** élément de ce tableau.

8.3.4 Retour sur les chaînes de caractères

Nous avons vu en section 7.2.1 que les chaînes de caractères étaient des tableaux de caractères avec une particularité : la présence **obligatoire** dans le tableau du caractère de terminaison de chaîne `'\0'`. Étant donné que les pointeurs peuvent être utilisés pour accéder aux éléments d'un tableau, il est donc possible de manipuler des chaînes de caractères à l'aide de pointeurs, mais quelques précautions s'imposent.

Déclaration d'une chaîne de caractères avec un pointeur

Considérons la déclaration suivante :

```
char *mystere;
```

Que représente la variable `mystere` ?

1. un pointeur sur des octets en mémoire ?
2. un tableau de caractères ?
3. une chaîne de caractères ?

La bonne réponse est la première. Mais il convient de compléter cette réponse car cela dépend également de ce sur quoi pointe le pointeur `mystere`. En l'absence de cette information, on peut juste se contenter de la réponse la plus générique (i.e la première). Mais une chaîne de caractères est bien évidemment un tableau de caractères, et un tableau de caractères constitue un ensemble de d'octets en mémoire, car un caractère est stocké dans un octet.

De façon générale, vous avez tendance à répondre spontanément que c'est la dernière réponse qui est la bonne : ce qui est faux dans le cas général. Un effet de bord malheureux et que par conséquent, vous allez utiliser des fonctions utilisables uniquement sur des chaînes (en particulier `strlen` ...) même quand ce n'est pas possible, ce qui va entraîner des erreurs : merci de relire les règles page 135.

Initialisation d'un pointeur avec une chaîne de caractères

Quelle sera l'exécution du programme suivant :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
```

17. Il peut être interprété de bien des manières et pas nécessairement celle-ci.

```c
int main(int argc, char *argv[])
{
  char toto[] = {'s','a','l','u','t','\0'};
  char tata[] = "salut";
  char *titi  = "salut";

  toto[0] = 'r';
  tata[0] = 'r';
  titi[0] = 'r';

  printf("%s %s %s\n",toto,tata,titi);

  return EXIT_SUCCESS;
}
```

string.c

8.3.5 ++ Parcours et position dans un tableau avec des pointeurs

Un pointeur constitue un moyen puissant d'effectuer des parcours de tableaux parce qu'ils permettent, **avec une seule variable**, d'avoir en même temps accès aux éléments du tableau et un indice permettant son parcours. Par exemple :

```c
#include <stdio.h>

#define SIZE_TAB 8

int main(int argc, char *argv[])
{
  int tab[SIZE_TAB] = {2,4,6,8,10,12,14,20};
  int *ptr_tab = NULL;

  for(ptr_tab=tab ; ptr_tab-tab < SIZE_TAB ; ptr_tab++)
    {
      *ptr_tab += 3;
      printf(" %i ",*ptr_tab);
    }
  printf("\n");

  return 0;
}
```

parcours_tableau_pointeur.c

Vous remarquerez que la condition d'arrêt de la boucle `for` est basée sur une différence de pointeurs, comme vue en section 8.2.2. Cette différence étant un nombre d'éléments, il est possible de faire la comparaison avec la taille du tableau.

Quelle que soit la méthode de parcours utilisée, la connaissance de la taille (le nombre d'éléments) du tableau est indispensable. Je rappelle qu'un pointeur sur un tableau ne vous donne pas cette information de taille.

Plus précisément, quand vous possédez un pointeur sur un élément d'un tableau, mais pas son indice, vous pouvez facilement obtenir ce dernier en faisant la différence entre l'adresse de cet élément et l'adresse de début de tableau, comme le montre cet exemple :

```c
#include <stdio.h>

#define SIZE_TAB 8

int main(int argc, char *argv[])
{
  int tab[SIZE_TAB] = {2,4,6,8,10,12,14,20};
  int *ptr_tab = NULL;

  for(ptr_tab=tab ; ptr_tab-tab < SIZE_TAB ; ptr_tab++)
    {
      int index = ptr_tab - tab;
      printf("%2i => pos %i in tab\n", *ptr_tab,index);
    }
  return 0;
}
```

indice_tableau_pointeur.c

Ce programme affiche bien ce qui est attendu :

```
 2 => pos 0 in tab
 4 => pos 1 in tab
 6 => pos 2 in tab
 8 => pos 3 in tab
10 => pos 4 in tab
12 => pos 5 in tab
14 => pos 6 in tab
20 => pos 7 in tab
```

Vous pouvez maintenant relire la section 7.1.3 concernant le type des indices de parcours de tableaux et avoir votre épiphanie.

Il s'agit là d'un moyen efficace pour retrouver l'indice d'un élément dans un tableau sans avoir à parcourir ce dernier et effectuer des comparaisons d'éléments. De plus, en cas d'éléments identiques dans le tableau, cette méthode basique de parcours et de comparaisons pose des problèmes.

8.4 Pointeurs et fonctions

Les pointeurs étant des variables classiques, ils peuvent en particulier être passés en argument à des fonctions. Ils sont aussi des types de retour valides pour les fonctions, notamment (mais pas seulement) pour pallier l'impossibilité de retourner des tableaux (cf. section 8.3.3).

8.4.1 Équivalence notationnelle des arguments de fonctions

Dans le cas de fonctions, il y a une équivalence de notation entre `*` et `[ ]` pour les arguments de fonction. Ainsi les deux prototypes suivants sont **strictement équivalents** :

```
int ma_fonction(int *arg);  /* arg est un pointeur */

int ma_fonction(int arg[]); /* arg est encore un pointeur */
```

Pourquoi une telle équivalence formelle ? Tout simplement parce que les tableaux n'existent pas vraiment. Nous avons vu que l'identificateur d'un tableau était en réalité un pointeur (constant) sur le premier élément de ce tableau. Donc, du point de vue de la fonction, ces deux types d'arguments sont identiques.

En revanche, cette équivalence n'est pas valable pour les pointeurs génériques. En effet, `void` n'étant pas un vrai type de données, déclarer un tableau d'éléments de ce type n'a pas de sens. Donc le prototype suivant n'est pas possible :

```
int ma_fonction(void arg[]); /* impossible */
```

Ceci dit, je vous rappelle que des pointeurs génériques peuvent tout à fait être passés en arguments à des fonctions :

```
int ma_fonction(void *arg); /* arg est un pointeur générique */
```

8.4.2 Modification des arguments d'une fonction

Nous avons vu que les arguments d'une fonction sont passés par valeur (cf. section 6.4), ce qui signifie qu'ils ne sont pas modifiés dans le corps de la fonction. Au retour de celle-ci, les variables passées en arguments possèdent toujours les valeurs qu'elles avaient avant l'appel de la fonction [18]. Or, il est courant de vouloir modifier des arguments passés à une fonction. Les pointeurs offrent **la** solution pour arriver à obtenir cela [19]. En effet comme la valeur d'un pointeur est une adresse, il va être possible de modifier des données situées à cette adresse, sans que la valeur de l'argument (le pointeur) ne soit modifiée. Il faut donc passer un pointeur sur une variable passée en argument à une fonction si on veut que cette dernière puisse modifier la variable.

Le pointeur lui-même n'est pas modifié par la fonction, parce qu'il est passé par valeur. Pour modifier le pointeur, c'est-à-dire changer la valeur de l'adresse qu'il contient alors il faut prendre un pointeur sur la variable pointeur elle-même (cf. section 8.1.5). Par exemple :

```
int **double_ptr = NULL;
```

est un pointeur double (aka un pointeur de pointeur).

Reprendre le code de la fonction `swap` et le modifier pour qu'il fonctionne de la façon attendue.

18. Rappel : en pratique les arguments passés à la fonction sont des **copies** des variables.
19. Ne cherchez pas, il n'y en a pas d'autre en C.

8.4.3 Pointeurs sur des fonctions

Outre les pointeurs sur des variables, il est également possible de prendre des pointeurs sur des fonctions. Il s'agit d'un mécanisme puissant pour gagner en abstraction dans vos programmes. Un pointeur de fonction se déclare de la façon suivante :

type_retour_t `(*`*nom_fonction*`)(` *liste_des_arguments* `)`

Cette déclaration signifie : *nom_fonction* est un pointeur sur une fonction qui prend comme arguments *liste_des_arguments* et qui retourne un résultat de type *type_retour_t*. Par exemple :

```
void * (* ma_super_fonction)(int arg1, int *arg2, void *arg3);
```

Comme pour les prototypes de fonctions, le nom des arguments peut être omis car c'est le nombre, l'ordre et le type des paramètres qui sont importants.


Les parenthèses autour du nom du pointeur sont indispensables.

Comme tout pointeur, les opérateurs `*` et `&` sont utilisables avec les pointeurs de fonctions. Cependant, en pratique, ils ne sont pas utilisés car **le nom d'une fonction est également un pointeur sur cette fonction**. Donc pour la récupération de pointeur :

`nom_fonction` $\equiv$ `&nom_fonction`

et pour déréférencer un pointeur de fonction (par exemple pour effectuer un appel avec des arguments) :

`nom_fonction(`*args*`...)` $\equiv$ `*nom_fonction(`*args*`...)`


Attention à ne pas confondre appel de fonction et pointeur de fonction (cf. section6.5) !


Les pointeurs de fonctions sont très utilisés et sont loin d'être anecdotiques, il est important de comprendre leur fonctionnement en vue d'une utilisation future (TP, projet, etc).

8.4.4 ++ Ambigüité des pointeurs en tant qu'arguments de fonctions

Un pointeur est une variable qui peut être interprétée de multiples façons (cf. l'exercice de la section 8.3.4). Et comme il y a équivalence formelle entre tableaux et pointeurs pour les arguments de fonction, un prototype de fonction peut vite devenir difficile à décrypter. Voici quelques petits conseils pour vous aider à y voir plus clair. Nous allons considérer une fonction `func` qui prend en argument un pointeur de type *type_t*.

— Si la fonction doit travailler sur un tableau et que le pointeur est le premier élément de ce tableau, alors il est raisonnable d'adopter le prototype suivant :

type_retour_t `func(` *type_t* `ptr[]);`

La présence des crochets montre bien que le paramètre attendu est un tableau.

— Si la fonction doit travailler sur une variable que l'on doit modifier, alors le prototype suivant pourra être envisagé :

type_retour_t `func(` *type_t* `*ptr);`

— Enfin si la fonction prend en argument une adresse sans que les données qui s'y trouvent ne soient modifiées, il est possible de le préciser de la façon suivante :

type_retour_t `func( const` *type_t* `*ptr);`

Le mot-clef `const` indique que les données pointées par `ptr` ne sont pas modifiables : on peut en déduire que ce n'est donc pas pour cette raison que le pointeur est passé en argument et que l'information dont la fonction a besoin est l'adresse elle-même.

Bien entendu, tout ceci reste de l'ordre du conseil et n'a rien d'obligatoire.

8.5 Allocation dynamique de mémoire

Jusqu'à présent, nous avons travaillé dans un contexte **statique** où la taille des données à utiliser est connue lors de la compilation du programme par le biais d'une définition de constante. Également, les variables possèdent une taille bien déterminée (c'est le critère pour que le compilateur ne génère pas d'erreurs). Mais il est courant que la taille des données à utiliser soit dynamique, c'est-à-dire que cette taille n'est connue qu'en cours d'exécution du programme. Nous avons déjà vu en section 4.4.4 l'organisation de la mémoire d'un programme et comment les variables y sont stockées en fonction de leurs classes de stockage. Cependant les zones de pile et de données ne servent qu'à stocker des données statiques et non des données dynamiques.

8.5.1 Zones mémoire d'un programme, le retour

La figure 8.1 montre l'organisation d'un programme avec une nouvelle zone, appelée le **tas** (*heap*), dans laquelle peuvent être allouées des données pendant l'exécution du programme.

Vous n'avez pas à gérer vous-même cette zone, le langage C vous propose des fonctions pour le faire (cf. section 8.5.2 ci-dessous).

Quand le tas augmente, les adresses croissent. Quand la pile augmente, les adresses **décroissent**.

Mais quel est le rapport avec les pointeurs ? La valeur d'un pointeur est une adresse et il est possible d'affecter à un pointeur l'adresse d'une partie du tas pendant l'exécution du programme. Les pointeurs vont donc nous servir dans ce cadre pour référencer des zones mémoire dimensionnées dynamiquement : c'est ce que nous appelons l'allocation dynamique de tas.

8.5.2 Allocation dynamique dans le tas

Si vous ne gérez pas vous-même le tas [20], vous allez vous reposer sur des fonctionnalités que le langage C fournit sous forme de fonctions disponibles dans une bibliothèque spéciale, appelée la libc [21]. Les fonctions d'allocation dynamique sont :

— `void *malloc(size_t size);`

20. La gestion du tas est une chose qui n'est pas *a priori* des plus faciles, alors autant ne pas en rajouter dans la difficulté.

21. La libc est un élément fondamental du système que vous utilisez. Elle est employée par défaut et de façon silencieuse quand vous compilez et exécutez vos programmes.

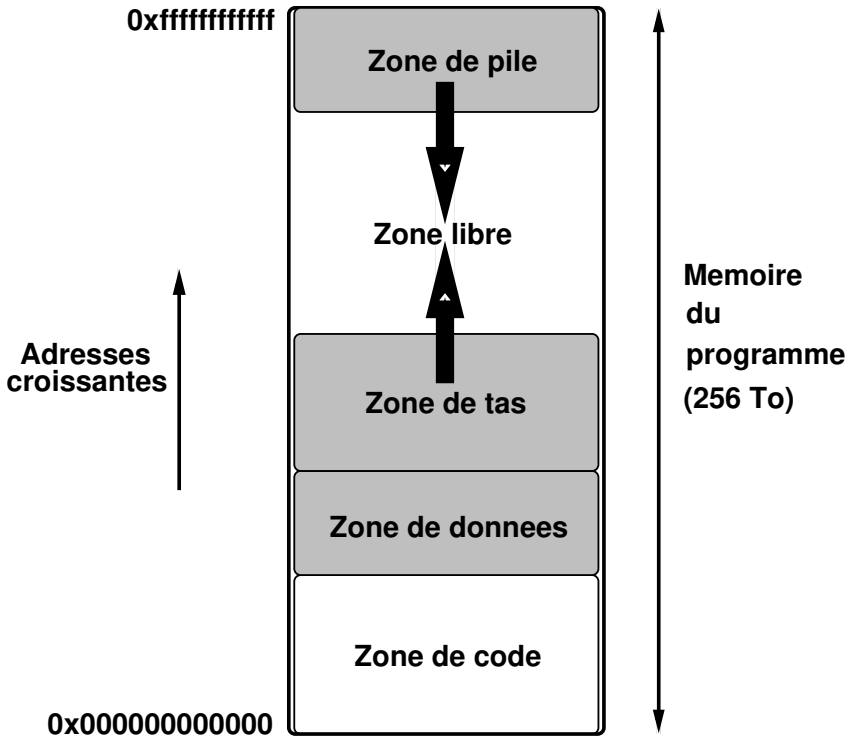


FIGURE 8.1: Organisation de la mémoire d'un programme en C (avec le tas). Chaque programme dispose (en environnement 64 bits) de 256 To de mémoire.

— `void free(void *ptr);`
— `void *calloc(size_t nmemb, size_t size);`
— `void *realloc(void *ptr, size_t size);`

Les deux fonctions que vous allez principalement utiliser sont `malloc` (M*emory* ALLOC*ation*) pour l'allocation dynamique dans le tas et `free` pour la libération d'une zone mémoire allouée avec une des fonctions d'allocation.

Quand une zone de la mémoire est libérée, elle est juste marquée comme étant réutilisable par le programme. La mémoire n'est pas effectivement effacée.

Rappel : en C, c'est **vous** qui gérez explicitement la mémoire, il n'y a pas de mécanisme de type ramasse-miettes (*garbage collector*) qui est mis en place [22]. En revanche, pas de panique : toute la mémoire allouée dans le programme est automatiquement libérée (au sens indiqué ci-dessus) quand il se termine. Ce qui ne veut pas dire qu'il ne faut pas faire preuve de rigueur quant à la gestion de cette ressource dans vos programmes !

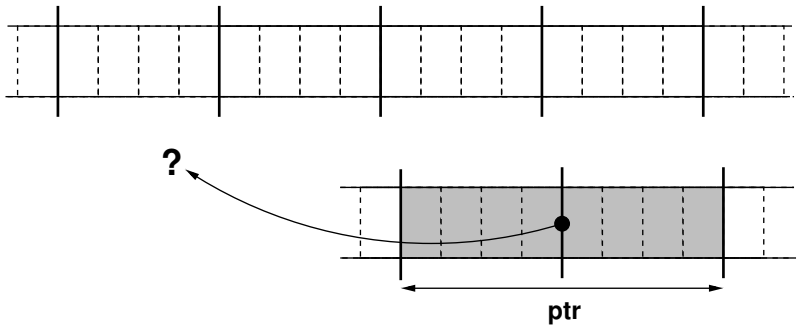
Déclaration de pointeur vs. allocation de mémoire

Une erreur courante est de faire une confusion [23] entre une déclaration de pointeur et une allocation de mémoire. Une déclaration de pointeur ne réserve de la mémoire que pour la variable pointeur elle-même mais elle n'effectue pas l'allocation de la zone sur laquelle le pointeur est

22. Ce qui est parfois le cas pour d'autres langages, surtout interprétés, car c'est l'interpréteur qui se charge alors de ce travail.
23. Attention à la contusion !

censé pointer (et d'ailleurs, où se trouve l'information de taille pour faire cette allocation ?). Ainsi, comme toute autre variable, un pointeur n'est pas automatiquement initialisé lors de sa déclaration. En particulier, la fonction `malloc` n'est pas appelée magiquement et automatiquement. Si vous ne l'appelez pas vous-même, alors rien ne se passe. Comparez la situation entre la déclaration simple d'un pointeur d'entier :

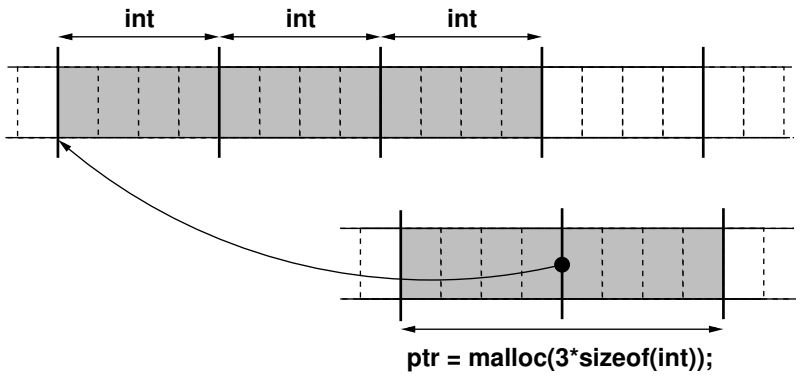
```
int *ptr; /* ptr pointe n'importe ou en memoire */
```



La valeur du pointeur n'est pas connue : il peut potentiellement pointer n'importe où en mémoire.

Voici maintenant avec une déclaration et une initalisation de pointeur : il pointe sur une zone de 3 entiers. Cette zone est allouée dynamiquement avec `malloc`.

```
int *ptr = malloc(3*sizeof(int));
```



La valeur du pointeur est l'adresse de la zone des trois entiers allouée avec `malloc`. Dans les deux cas, comme `ptr` est une variable, de la mémoire est réservée pour la stocker. Si `ptr` est une variable automatique, elle sera allouée sur la pile.

Utilisations de `malloc` et de `free`

La fonction `malloc` prend en argument une taille en **octets** et non pas en nombre d'éléments. N'oubliez donc pas le `sizeof(`*type*`)` adéquat ! De plus La mémoire allouée avec `malloc` n'est jamais récupérée tant que `free` n'est pas appelée (avec comme argument l'adresse de la zone à libérer).

Logiquement, un appel à `free` devrait apparaître dans vos programmes pour chaque appel à `malloc` effectué.

La fonction `malloc` s'utilise de deux façons :

1. Le premier (et le plus évident) cas d'utilisation de cette fonction d'allocation dynamique concerne les données dont la taille est connue en cours d'exécution. C'est dans ce cas que vous allez utiliser `malloc` le plus ;
2. le second (et moins évident) cas d'utilisation de `malloc` concerne les allocations de données **pérennes**. En effet, toute donnée stockée dans le tas y reste tant que la zone n'a pas été libérée avec `free`. Il est des fois intéressant de réserver une telle zone (y compris quand la taille est connue statiquement) parce qu'on a la garantie que les données y resteront (je vous rappelle que la pile est gérée dynamiquement et que les données qui y sont stockées sont susceptibles d'être écrasées en cours d'exécution du programme.). Par exemple, il peut être intéressant d'allouer dans une première fonction une zone avec `malloc`, de retourner un pointeur dessus et de le passer ensuite à d'autres fonctions pour qu'elles utilisent la zone.

8.5.3 ++ Allocation dynamique sur la pile

Une autre erreur classique consiste à restreindre le principe de l'allocation dynamique à la zone de tas. En effet, il est également possible d'allouer des données dynamiquement sur la pile. Ce n'est pas quelque chose que je vous recommande de faire, mais il faut savoir que cela existe et c'est même quelque chose que vous faites sans le savoir, en déclarant une taille de tableau dynamiquement. Par exemple :

```
int n;

/* choses dans le programme */

n = /* calcul de n */

/* autres choses dans le programme */

int tab[n];

/* le programme continue */
```

Cette allocation n'est **pas** une allocation sur le tas, mais sur la pile. C'est l'équivalent d'un appel à la fonction `alloca`. Ce genre d'acrobaties est possible quand nous ne respectez pas les règles de la page 135, notamment la déclaration de variable en début de bloc et la taille entre `[]` doit être une constante. Un code équivalent avec un appel à `malloc` et qui respecte les règles serait :

```
int  n;
int *tab = NULL; /* je ne connais pas encore la taille */

/* choses dans le programme */

n = /* calcul de n */

/* autres choses dans le programme */

tab = malloc(n*sizeof(int));

/* le programme continue */

free(tab);
```

Bien entendu, il faut dans ce cas libérer la mémoire avec `free`.

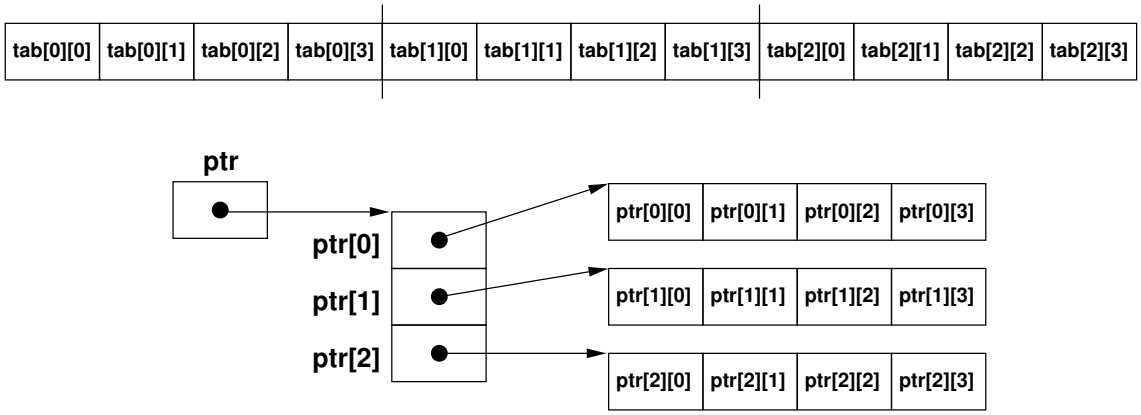
Il n'est pas nécessaire de libérer la mémoire de la pile car c'est automatique. En particulier, il ne faut pas appeler `free` sur des adresses de pile [24].

8.6 ++ Pointeurs de pointeurs vs. tableaux multidimensionnels

Nous avons déjà vu qu'il existait une analogie entre tableaux et pointeurs (cf. section 8.3). Nous savons par ailleurs qu'il existe des tableaux multidimensionnels (cf. section 7.4) et qu'il est possible de récupérer un pointeur sur un pointeur (cf. section 8.4.2). Il est donc légitime de se poser la question suivante : un pointeur multiple est-il l'équivalent d'un tableau multidimensionnel ?

La réponse est non : dans le cas d'un tableau multidimensionnel, tous les différents tableaux sont alloués de façon contigüe en mémoire. C'est d'ailleurs ce qui permet de convertir des coordonnées en plusieurs dimensions en une unique coordonnée (cf. formule en section 7.4.4). Dans le cas de pointeurs de pointeurs, les différentes zones mémoires allouées ne sont pas forcément contigües et il n'est donc pas possible d'effectuer cette conversion de coordonnées. Comparons un exemple en deux dimensions :

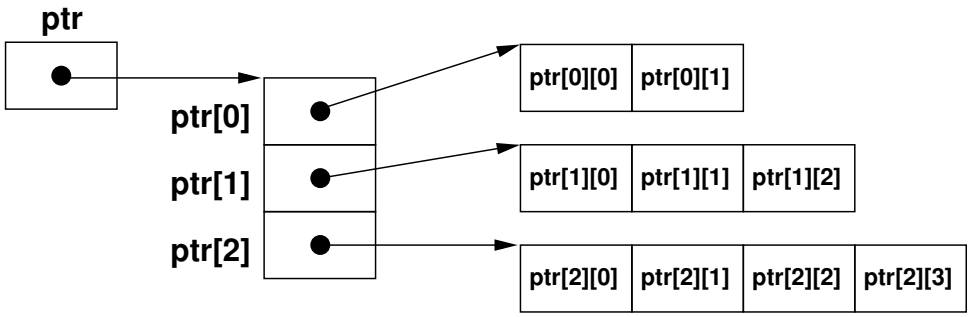
```
long int tab[3][4] ;

long int **ptr = NULL;
ptr = malloc(3*sizeof(long int *));
for (int i = 0 ; i < 3 ; i++)
  ptr[i] = malloc(4*sizeof(long int));
```



En revanche, l'approche des pointeurs de pointeurs permet d'avoir des tableaux de tailles différentes sur les dimensions suivant la première. Par exemple :

```
long int **ptr = NULL;
ptr = malloc(3*sizeof(long int *));
ptr[0] = malloc(2*sizeof(long int));
ptr[1] = malloc(3*sizeof(long int));
ptr[2] = malloc(4*sizeof(long int));
```

24. Il semblerait que sur les systèmes actuels, il ne soit plus possible de récupérer une adresse de pile de toute façon (`NULL` est renvoyé dans tous les cas) alors que c'était possible auparavant.


ptr
ptr[0]
ptr[1]
ptr[2]
ptr[0][0]
ptr[0][1]
ptr[1][0]
ptr[1][1]
ptr[1][2]
ptr[2][0]
ptr[2][1]
ptr[2][2]
ptr[2][3]

9. Structures et unions de types

9.1 Structures de données

Les tableaux (cf. Chapitre 7 offrent un moyen pratique de manipuler des variables qui devraient être conceptuellement utilisées de façon concomitante (cf. principe de localité spatio-temporelle des données). Ils permettent de pouvoir utiliser autre chose que des scalaires dans les programmes parce qu'ils se rapprochent fondamentalement de vecteurs.

Les structures, quant à elles, sont des collections de variables, possiblement de types différents, qui sont regroupées au sein d'une même entité pour une manipulation plus facile. En effet, s'il est toujours possible de déclarer les variables d'un structure de façon indépendante, cela ne permet pas de bien mettre en évidence l'organisation logique du code et ne facilite ni sa lecture, ni sa maintenance ou sa mise à jour (surtout dans un contexte de grosse application).

9.1.1 Définition de structure

La définition d'une structure revient à faire la liste exhaustive de ses composantes, que l'on appelle des **champs**. La syntaxe de définition est la suivante :

```
struct identificateur_optionnel {
    nom-de-type identificateur-de-champ_1 ;
    nom-de-type identificateur-de-champ_1 ;
    ...
    nom-de-type identificateur-de-champ_n ;
};
```

N'oubliez pas le terminateur d'instruction ; à la fin de la définition de la structure !

Une définition de structure n'est **pas** une déclaration de variable. Il s'agit en fait de la création d'un **nouveau type de données**, qui vient s'ajouter aux types de base du langage C (cf. section 4.1).

L'identificateur placé après le mot-clef `struct` est optionnel, mais c'est un synonyme de la définition de la structure (le bloc entre accolades) ce qui permet de ne pas avoir à répéter cette définition lors de la déclaration d'une variable du type de la structure définie.

Exemple

Afin de fixer les idées, nous allons définir une structure `point` contenant des coordonnées cartésiennes :

```
struct point {
  int abscisse;
  int ordonnee;
};
```

Nous pouvons maintenant déclarer des variables possédant ce nouveau type que nous venons de définir de la façon suivante :

```
struct point pt1, pt2;
```

Dans le cas où l'identificateur `point` ne serait pas utilisé, la déclaration de ces variables prendrait alors cette forme :

```
struct {
  int abscisse;
  int ordonnee;
} pt1, pt2;
```

Nous pouvons également utiliser le mot-clef `typedef` afin de créer un nouveau nom de type en même temps que le type lui-même :

```
typedef struct point { /* definition de type */
  int abscisse;
  int ordonnee;
} point_t;              /* point_t est un nouveau nom de type */

point_t truc;           /* declaration d'une variable de type point_t */
```

L'identificateur `point` est encore une fois optionnel dans ce cas. Bien entendu, il est possible qu'un champ d'une structure soit lui-même une structure (imbrication) :

```
struct point {
  int abscisse;
  int ordonnee;
};

struct cercle {
 struct point centre;
 int          rayon;
};
```

9.1.2 Accès aux champs d'une structure

Tous les champs d'une structure peuvent être accédés individuellement avec l'opérateur `.` (point). Reprenons l'exemple précédent :

```
struct point {
  int abscisse;
  int ordonnee;
};

struct point truc;
```

`truc` est donc une variable de type `struct point`, qui possède deux champs : `abscisse` et `ordonnee`. Nous pouvons utiliser ces champs comme des variables entières classiques de type `int`, par exemple :

```
truc.abscisse = 1;
truc.ordonnee = 1;
```

Il est possible de récupérer un pointeur sur un champ d'une structure de façon classique :

```
int *ptr = &truc.ordonnee ;
```

L'opérateur `.` étant plus prioritaire que l'opérateur `&`, les parenthèses ne sont pas nécessaires dans ce cas.

9.1.3 Opérations sur les structures

Outre l'accès aux champs avec l'opérateur `.`, la plupart des opérations possibles sur les variables possédant un type de base sont aussi possibles sur les structures, notamment :

- la copie d'une structure dans une autre structure **de même type** avec l'opérateur d'assignation `=`
- les structures sont assignables (et donc initialisables à la déclaration [1]) avec l'opérateur d'assignation `=`
- le passage en tant qu'argument à des fonctions
- les structures peuvent être retournées par des fonctions
- La récupération d'un pointeur sur une structure avec l 'opérateur unaire `&`
- le déréférencement d'un pointeur sur une structure avec l 'opérateur unaire `*`
- la réplication avec l'opérateur `[]` pour créer des tableaux de structures

En revanche, il n'est **pas possible de comparer** des structures directement entre-elles avec les opérateurs de comparaison (i.e. `==`, `!=`, `<`, `>`, `<=` et `>=`). Il n'est pas possible non plus de récupérer un pointeur sur un champ individuel en calculant l'adresse manuellement à partir de l'adresse de début de la structure (cf. ci-dessous).

Nous avons vu qu'il n'existe pas d'opérateur permettant de copier directement un tableau dans un autre, quelque chose comme :

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5   int  A[5]  = {0,0,0,0,0};
6   int  B[5]  = {1,2,3,4,5};
7
8   A = B;
9
```

1. mais pas à la définition

```
10    for(int i = 0; i < 5; i++)
11      printf("%i ",A[i]);
12    printf("\n");
13
14    return 0;
15  }
```

copie_tableau.c

n'est pas possible. Pourtant, le code ci-dessous est possible et fonctionne parfaitement :

```
1   #include <stdio.h>
2
3   struct array {
4     int tab[5];
5   };
6
7   int main(int argc, char *argv[])
8   {
9     struct array A = {.tab = {0,0,0,0,0}};
10    struct array B = {.tab = {1,2,3,4,5}};
11
12    A = B;
13
14    for(int i = 0; i < 5; i++)
15      printf("%i ",A.tab[i]);
16    printf("\n");
17
18    return 0;
19  }
```

copie_tableau2.c

Pourquoi ? Réponse : 1- déjà, on ne connaît pas la taille d'un tableau donc impossible de savoir combien il faut copier. Avec une structure, on connait la taille. 2- A est un pointeur constant (`int *cont A`) donc inassignable. Une structure est assignable, elle. 3- Même si A était assignable, ce ne serait pas une copie mais de l'aliasing de pointeurs.

Initialisation d'une structure

Tout comme un tableau (cf. section 7.1.2), une structure est initialisable statiquement au moment de sa déclaration avec une liste d'expressions constantes (i.e { $valeur_1$, $valeur_2$, ..., $valeur_n$ }) Par exemple :

```
struct point {     /* definition de la structure */
  int abscisse;
  int ordonnee;
};

struct point bidule = {1, 4};  /* premiere version */
struct point truc   = {.abscisse = 2, .ordonnee = 12}; /* seconde version */
```

Dans la première version, les valeurs fournies sont affectées aux champs selon l'ordre de leur déclaration (i.e `bidule.abscisse` vaut `1` et `bidule.ordonnee` vaut `4`).


Attention de ne pas oublier les points devant les noms des champs dans la seconde version !

Pointeurs sur des structures

La récupération d'un pointeur sur une variable dont le type est une structure se fait classiquement avec l'opérateur unaire `&`. Similairement, il est possiblde de déclarer un pointeur sur une structure :

```
struct point {    /* definition de la structure */
  int abscisse;
  int ordonnee;
};

/* declaration et initialisation des variables */
struct point pt = {.abscisse = 2, .ordonnee = 34 };
struct point *ptr = NULL;

/* assignation du pointeur */
ptr = &pt;
```

Ainsi si `ptr` est un pointeur sur une structure de type `struct point`, alors `*ptr` est une structure de ce type également : l'opérateur de déférencement fonctionne là encore de façon tout à fait classique. Il est dès lors possible d'accéder aux champs de la structure avec l'opérateur `.` : selon la définition de `struct point`, `(*ptr).abscisse` et `(*ptr).ordonnee` sont donc des variables de type entier `int`.


Notez dans l'exemple précédent l'emploi de parenthèses, qui ne sont pas là pour faire joli. Si nous les enlevons, alors l'expression `*ptr.abscisse` pose un problème. En effet, l'opérateur `.` étant plus prioritaire que l'opérateur `*`, sans les parenthèses, le compilateur interprète `*ptr.abscisse` comme `*(ptr.abscisse)`. Or, `ptr.abscisse` ne correspond à rien et une erreur sera générée lors de la compilation. Les parenthèses servent ici à indiquer que l'on déréférence **d'abord** le pointeur **puis** que l'on accède au champ de la structure.

Il est très fréquent d'accéder aux champs d'une structure obtenue par déréférencement, mais l'emploi des opérateurs `*` et `.` n'est pas très pratique. Il existe donc un opérateur spécifique pour cela : `->`. Ainsi, en lieu et place de `(*ptr).abscisse`, on écrira plutôt `ptr->abscisse`.


(**pointeur-de-structure*) .*champ* $\Longrightarrow$ *pointeur-de-structure*->*champ*

Vous utiliserez **systématiquement** l'opérateur `->` dans vos programmes. Personne, mais vraiment personne, n'utilise `(*).` pour coder.

Cette notation se révèle pratique en cas de pointeurs de structures imbriqués. Imaginons la situation suivante :

```
/* definitions de structures */
struct A {
  int x;
  int y;
```

```
};

struct B {
  char array[10];
  int  z;
  struct A *a_ptr;
};

/* fonction renvoyant un pointeur de structure */
struct B *func(void)
{
 /* ...... */
  return ptr;
}

int main(int argc, char *argv[])
{
  struct B *ptr = func();

  /*acces au champ z de B */
  ptr->z = 23;

  /*acces au champ x de A */
  ptr->a_ptr->x = 45;

  /* ............. */
  return 0;
}
```

On suppose que dans la fonction, le pointeur de structure a été correctement alloué, avec par exemple un appel à `malloc(sizeof(struct A))`. L'associativité de l'opérateur `->` fait que les parenthèses sont inutiles pour l'expression `ptr->a_ptr->x`. Sans l'opérateur `->`, il aurait fallu écrire `(*(*ptr).a_ptr).x = 45;`, ce qui est à la fois plus compliqué et bien moins lisible [2].

9.1.4 Taille d'une structure

Le point le plus délicat pour les structures concerne leur taille. Tout comme pour un type de base, il faut **toujours** utiliser l'opérateur `sizeof` pour calculer la taille d'une structure.

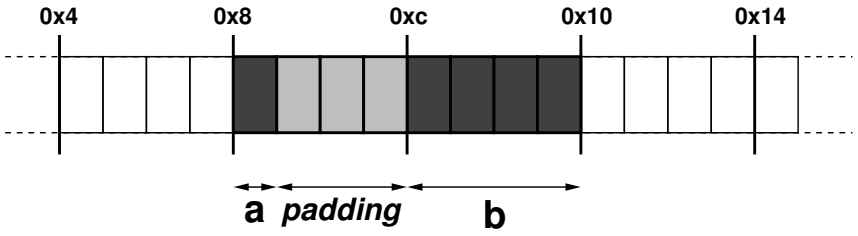
La taille d'une structure est toujours **supérieure ou égale** à la somme des tailles des champs qui la composent.

Pourquoi cela ? Rappelez vous : les données ne sont pas disposées aléatoirement en mémoire, elles sont **alignées** sur des adresses particulières, et cela pour des raisons de performance (cf. section 8.1.3). Concrètement, à cause de cet **alignement mémoire**, les champs d'une structure ne sont pas placés consécutivement en mémoire et du bourrage (*padding*) est possible. C'est notamment pour cela qu'il ne faut pas calculer les adresses des champs individuels à partir de la somme de l'adresse de base de la structure et des tailles des champs intermédiaires. Prenons par exemple la structure suivante :

2. Vous reprendrez bien un petit peu de sucre syntaxique, hmmm ?

```
struct bidule {
  char a;
  int  b;
};
```

En supposant que les `int` sont codés sur 32 bits (4 octets), la taille de cette structure sera de 8 octets, alors que la taille du champ `a` est de 1 octet et celle du champ `b` est de 4 octets, soit 5 octets au total. Cependant, le champ `b` doit être aligné sur une frontière d'entier et donc nous aurons l'occupation mémoire suivante (comme d'habitude, une case représente un octet) :



Comme vous pouvez le constater, **3 octets de padding sont intercalés** entre `a` et `b`, ce qui porte la taille effective à 8. L'alignement est variable selon les systèmes et la définition de la structure : il sera calculé en fonction du champ de plus grande taille.

Enfin, la taille d'un structure dépend de l'ordre dans lequel ses champs sont déclarés. Encore une fois, c'est l'alignement qui est responsable de cela. Regardons le programme suivant :

```
1  #include <stdio.h>
2
3  struct machin {
4    char a;
5    char b;
6    int  c;
7  };
8
9  struct bidule {
10   char a;
11   int  c;
12   char b;
13 };
14
15 int main(int argc, char *argv[])
16 {
17   struct machin x = {.a = 'z', .b = 'e', .c = 33};
18   struct bidule y = {.a = 'r', .c = 666, .b = 'p'};
19
20   printf("taille de x : %lu\n",sizeof(x));
21   printf("taille de y : %lu\n",sizeof(y));
22
23   return 0;
24 }
```

structures.c

Quel sera l'affichage de ce programme ? Faites un dessin de l'occupation mémoire des variables `x` et `y`.

Depuis C11, langage C offre une fonction `alignof` qui permet de calculer cet alignement, comme le montre le code ci-dessous :

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

struct bidule {
  char a;
  int  b;
};

int main(int argc, char *argv[])
{
  printf("Size is %li, Alignment is %li\n",
   sizeof(struct bidule), alignof(struct bidule));
  return 0;
}
```

alignement.c

Ce programme affiche : `Size is 8, Alignment is 4`

La taille d'une structure de données ou d'un type est toujours un multiple de son alignement. Pour un type de base, sa taille et son alignement sont égaux.

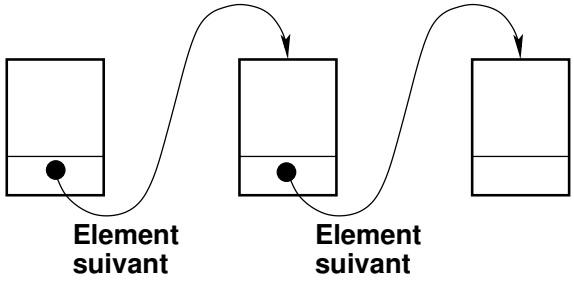
Vous n'avez **pas** besoin de connaître les règles d'alignement mémoire, mais juste de savoir que le phénomène existe. Utilisez systématiquement l'opérateur `sizeof` pour calculer la taille d'une structure et vous ne rencontrerez aucun problème dans vos programmes [3]. vous n'aurez jamais à utiliser la fonction `alignof` de C11 non plus.

9.1.5 Structures auto-référentes

Généralités sur les listes chaînées

Les structures (au sens du langage C) sont des constructions qui permettent également d'implémenter des structures de données (au sens théorique ou algorithmique) très courantes : les **listes chaînées** . Une liste chaînée est une structure possédant un chaînage, c'est-à-dire que dans la structure se trouve une variable qui permet d'accéder au reste de la liste.

3. Si, si c'est promis. Juré, craché.



À la différence d'un tableau, il n'est pas possible d'accéder à n'importe quel élément de la liste en temps constant. L'accès à un élément dépend de l'état de la liste. L'avantage des listes chaînées est leur aspect dynamique : lorsque d'un tableau est rempli, il faut en allouer un nouveau plus grand et copier dans le nouveau l'ancien puis le libérer. Cela implique souvent de devoir réserver plus de place que nécessaire afin de prévoir le stockage de nouveaux éléments dans le tableau. Ceci n'est pas utile dans le cas des listes chaînées qui sont tout le temps idéalement dimensionnées. En cas de nouvel élément, il est créé/alloué puis rajouté dans la liste.

Définition de chaînage à l'aide de structures

En C, une définition de type ou une déclaration de variable n'est valide que si le compilateur est capable d'en calculer la taille pour en déduire l'occupation mémoire. Donc, la définition suivante est problématique :

```
struct bidule {
  int champ1;
  int champ2;
  /* ..... */

  struct bidule suivant;
}
```

Il n'est pas licite pour une structure de contenir un champ de son propre type car la définition de la structure ne permet pas d'en calculer la taille. En revanche, il est possible d'avoir un **pointeur sur une structure du type en train d'être défini**. En effet, un pointeur, quel que soit son type, possède une taille bien définie, ce qui permet de calculer la taille du type :

```
struct bidule {
  int champ1;
  int champ2;
  /* ..... */

  struct bidule *suivant;
}
```

est donc une définition licite. Le pointeur n'a pas vocation à être obligatoirement le dernier champ de la structure. Également, il est possible d'avoir des chaînages multiples.

Exo Définissez une structure d'arbre binaire, comportant une étiquette entière et un sous-arbre gauche et un sous-arbre droit.

! Les structures auto-référentes sont des constructions **très** utilisées dans le langage C. Cette partie du cours doit donc être consciencieusement étudiée.

9.1.6 De l'utilité des structures ...

Les structures sont des constructions du langage C qui sont très utilisées parce qu'elle permettent une meilleure organisation du code. Ce dernier est également plus facilement maintenable. Pur fixer les idées, supposons que nous codions une fonction destinée à se retrouver dans une bibliothèque et donc utilisée dans de nombreux codes applicatifs. Imaginons que le prototype de cette fonction soit le suivant :

```
int ma_super_fonction(int arg1, char arg2);
```

Si, dans le futur, cette fonction est appelée à évoluer et à prendre des arguments supplémentaires, alors son prototype devra être modifié en conséquence :

```
int ma_super_fonction(int arg1, char arg2, int arg3, void *arg4);
```

Et bien entendu, **tous** les programmes qui utilisent cette fonction devront être modifiés également en remplaçant les appels de la fonction. Maintenant, si je définis la structure suivante :

```
struct mon_type {
 int   arg1;
 char  arg2;
 int   arg3;
 void *arg4;
} ;
```

et que ma fonction a pour prototype :

```
int ma_super_fonction(struct mon_type arg);
```

Alors il me suffit à l'avenir de modifier ma structure sans avoir à changer le prototype de la fonction. En revanche, il est évident que le corps de la fonction devra, lui être modifié pour prendre en compte les nouveaux champs rajoutés à la structure. Quant aux codes applicatifs utilisant la fonction, les appels ne seront pas changés, mais il est à peu près certain qu'il faudra apporter des modifcations mineures.

L'exemple ci-dessus est destiné à nourrir votre réflexion ; je ne vous demande **pas** de systématiquement créer des structures pour passer des arguments à une fonction. Il faudra justement réflechir au cas par cas à la pertinence de la mise en place de ce type de mécanismes quand vous codez.

9.1.7 ++ Champs de bits

La notion de champ de bits existe en C avec les structures, il faut indiquer un longueur de champ en bit (: *length*). Les champs de bit ne sont pas portables donc je n'en parlerai pas plus.

9.2 Unions de types

Une variable déclarée comme union de types est une sorte de variable qui qui possède un type (et donc une taille) changeant durant l'exécution du programme (le compilateur se charge de la gestion des problèmes de taille et d'alignement). Les unions permettent de manipuler des données possédant des types différents mais occupant une unique zone de stockage.

9.2.1 Déclaration d'union

La déclaration ressemble à celle d'une structure, avec le mot-clef `struct` remplacé par le mot-clef `union` :

`union` *identificateur*$_{optionnel}$ `{`
nom-de-type$_1$ *identificateur-de-champ*$_1$;
nom-de-type$_2$ *identificateur-de-champ*$_1$;
...
nom-de-type$_n$ *identificateur-de-champ*$_n$;
`};`

À l'opposé d'une structure, les types des divers champs sont tous différents. En effet, une variable déclarée comme une union de types ne possède qu'un type à un instant donné et ce type sera soit *nom-de-type*$_1$, soit *nom-de-type*$_2$, ... soit *nom-de-type*$_n$. Donc il ne sert à rien d'avoir des champs de même type dans une union.

9.2.2 Typage effectif d'une union

Pour typer effectivement une union, on utilise l'opérateur `.` comme pour les structures, et la sélection d'un champ d'un type donné détermine le type de l'union. Par exemple si on déclare l'union suivante :

```
union bidule {
  int entier;
  char carac;
} ;

union bidule toto ;
```

Alors la variable `toto` est soit de type entier `int`, soit de type caractère `char`. Pour choisir le type effectif de `toto`, on choisit le champ correspondant à ce type :

`toto.entier` $\Longrightarrow$ `toto` est un entier
`toto.carac` $\Longrightarrow$ `toto` est un caractère

Dans le cas d'un pointeur d'union, c'est l'opérateur `->` qui va remplacer `(*).` comme pour les structures.

9.2.3 Occupation mémoire d'une union

La taille d'une union correspond à la taille du **champ de plus grande taille**. En effet, une union étant un choix d'un type parmi plusieurs, il n'est pas utile de réserver de la place pour chacun des champs.

9.2.4 ++ Forçage de l'alignement avec une union

Une union peut être utilisée pour aligner une structure de données sur une frontière particulière : il suffit de déclarer dans l'union un champ du type de la fontière voulue. Ce champ ne sera jamais utilisé en pratique mais il force l'alignement. Par exemple

```
union a_aligner {
  struct donnees {
    char a;
    char b;
  } data ;
  long int alignement;
};

union a_aligner bidule;
```

Dans ce cas, la variable `bidule` qui est en fait une structure contenant deux caractères, sera alignée sur une frontière d'entier long `long int`, soit tout les 8 octets, comme le confirme le programme suivant :

```
1  #include <stdio.h>
2  #include <stdalign.h>
3
4  struct donnees{
5      char a;
6      char b;
7  };
8
9  union a_aligner {
10   struct donnees data;
11   long int alignement;
12 };
13
14 int main(int argc, char *argv[])
15 {
16   union a_aligner bidule;
17
18   printf("alignement data : %lu \n",
19    alignof(struct donnees));
20   printf("alignement union: %lu \n",
21    alignof(bidule));
22
23   return 0;
24 }
```

forcage_alignement.c

Ce programme affiche :

```
alignement data : 1
alignement union: 8
```

10. Le Préprocesseur C

Le préprocesseur C permet quelques facilités d'écriture pour les programmes mais il faut être parfois vigilant(e) au code produit. Pour rappel, lors de la première phase de compilation, le préprocesseur transforme votre texte (code-source original) en un nouveau texte et traite toutes les directives, c'est-à-dire modifie le texte selon les indications de ces dernières. Ces directives se reconnaissent car le premier caractère de la ligne est #.

Vous avez déjà utilisé le préprocesseur pour :

— inclure des fichiers d'entête `.h` afin de pouvoir utiliser certaines fonctions déjà implémentées

— définir des constantes (cf. section 4.2.2)

Cependant, le préprocesseur permet plus de choses.

10.1 Compilation conditionnelle

10.1.1 Code optionnel

Une utilisation intéressante du préprocesseur consiste à compiler une version du code ou une autre selon qu'une constante est définie (il suffit de la déclarer comme définie, peu importe sa valeur). Cela permet de faire cohabiter plusieurs versions et de ne pas avoir à modifier le code chaque fois que l'on souhaite en changer. Par exemple, il est ainsi possible d'écrire une version "debug" de votre code avec des `printf` un peu partout que vous n'allez activer que lorsque la constante `DEBUG` est définie. Pour cela, il faut utiliser la directive `#ifdef` :

```
#ifdef DEBUG
  printf("Debug : un message de controle\n");
#endif
```

La portion de code activée se trouve entre la directive `#ifdef` jusqu'au `#endif` correspondant. Il est possible d'imbriquer ces directives :

```
#ifdef DEBUG
  printf("Debug : un message de controle\n");
#ifdef DEBUG2
```

```
  printf("Encore plus de debug!\n");
#endif
#endif
```

Dans ce cas, le second `printf` ne s'affichera que si les constantes `DEBUG` **et** `DEBUG2` sont définies.

10.1.2 Code alternatif

Il est possible également d'avoir des parties de codes alternatives et pas uniquement supplémentaires avec la directive `#else` :

```
#ifdef DEBUG
  printf("Debug : un message de controle\n");
#else
  printf("Message normal\n");
#endif
```

10.1.3 Définition dynamique de constante

S'il est possible de définir les constantes dans les fichiers sources à l'aide de la directive `#define`, cela n'est pas très pratique puisque cela impose des modifications de fichiers. Il existe un moyen dynamique de faire cette définition : utiliser l'option `-D` du compilateur qui permet de définir une variable à la compilation, sans avoir à le faire dans le code. Par exemple :

```
gcc -DDEBUG -o toto toto.c
```

Vous remarquerez l'absence d'espace entre le `-D` et le nom de la constante. Également, la constante est uniquement définie mais sans valeur particulière. Il est possible de définir une valeur à la compilation :

```
gcc -DDEBUG=3 -o toto toto.c
```

Dans ce cas, il est possible de tester la valeur de cette constante avec la directive `#if` :

```
#if DEBUG == 1
  printf("Debug1 : un message de controle\n");
#elif DEBUG == 2
  printf("Debug2 : un autre message de controle\n");
#elif DEBUG == 3
  printf("Debug2 : encore un autre message de controle\n");
#endif
```

La directive `#elif` remplace `#else if`.

10.2 Macros

10.2.1 Définition de macros

Le préprocesseur C autorise la définition de **macros**. Une macro se déclare ainsi :

`#define` *nom-de-macro* *texte-de-remplacement*

Le fonctionnement est simple : le préprocesseur remplace dans le code le texte *nom-de-macro* par le *texte-de-remplacement* (sauf si *nom-de-macro* se trouve dans une chaîne de caractères).

Il est possible de déclarer des macros avec des arguments, par exemple :

```
#define TRUC(x)  if( x ) printf("OK\n")
#define MAX(a, b) a  >  b ? a : b
```

Il ne s'agit pas d'un appel de fonction car c'est le code qui est copié directement [1]

Il faut cependant faire attention à l'écriture des macros, comme le montre l'exemple ci-dessous :

```
#define SUM(A,B) A+B

int main(int argc, char *argv[])
{
   int a = 2;
   int b = 3;

   printf("%i\n", SUM(a,b)*SUM(a,b));

   return 0;
}
```

Que va afficher ce programme ?

Il faut toujous mettre des parenthèses dans les macros pour éviter les problèmes. Par exemple :

```
#define SUM(A,B) ((A) + (B))
```

10.2.2 ++ Remplacer un paramètre par une chaîne de caractères

Un nom de paramètre de macro peut être remplacé par une chaîne de caractères correspondante si ce paramètre est précédé du caractère # :

```
#define MY_MACRO(x)   printf(#x " = %i\n", x)
```

Si le code du programme "appelle" la macro avec pour paramètre l'expression `y++`, alors cette macro sera remplacée dans le code par :

```
printf("y++" " = %i\n", y++)
```

La fonction `printf` concatène automatiquement des chaînes de caractères et donc :

```
printf("y++ = %i\n", y++)
```

10.2.3 ++ La directive

Cette directive permet de concaténer des paramètres de macros.

Soit la définition suivante :

```
#define COLLAGE(x, y)  x ## y
```

Si le code appelle la macro comme ceci :

```
int truc = COLLAGE(toto, 3);
```

alors le préprocesseur remplace le tout par :

```
int truc = toto3;
```

En supposant qu'une variable `toto3` a été déclarée et est visible ici. Il est possible de générer des noms de fonctions suivant certains types de données par exemple.

1. Les fonctions déclarées en `static inline` (cf. section 6.5.2) fonctionnent donc à mi-chemin entre des macros et des fonctions classiques : si le compilateur réussi à faire l'*inlining* alors le corps de la fonction est copié et il n'y a pas d'appel. Sinon, c'est un appel classique de fonction.

V

Annexes

Quelques fonctions (utiles ?) de la libc

- double atof(const char *nptr);
- int atoi(const char *nptr);
- long atol(const char *nptr);
- long long atoll(const char *nptr);
- void *calloc(size_t nmemb, size_t size);
- int fgetc(FILE *stream);
- char *fgets(char *s, int size, FILE *stream);
- int fprintf(FILE *stream, const char *format, ...);
- void free(void *ptr);
- int getchar(void);
- int isalnum(int c);
- int isalpha(int c);
- int iscntrl(int c);
- int isdigit(int c);
- int isgraph(int c);
- int islower(int c);
- int isprint(int c);
- int ispunct(int c);
- int isspace(int c);
- void *malloc(size_t size);
- void *memcpy(void *restrict s1, const void *restrict s2, size_t n);
- void *memmove(void *dest, const void *src, size_t n);
- int printf(const char *restrict format, ...);
- void *realloc(void *ptr, size_t size);
- int scanf(const char *restrict format, ...);
- int strcmp(const char *s1, const char *s2);
- char *strcpy(char *restrict s1, const char *restrict s2);
- size_t strlen(const char *s);

Règles d'hygiène de programmation

Première règle :

En début de bloc tes variables toujours du déclareras.

Deuxième règle :

Pour déclarer la taille d'un tableau, une constante toujours tu utiliseras.

Troisième règle :

Pour calculer la taille d'un tableau, jamais `sizeof` *tu n'utiliseras.*

Quatrième règle :

Pour calculer la taille d'un tableau de caractères, jamais `strlen` *tu n'utiliseras.*

Références bibliographiques

1. Brian W. Kernighan et Dennis Ritchie, *The C programming language Second Edition*, Prentice Hall
2. Jean-pierre Braquelaire, *Méthodologie de la programmation en langage C*

Notes de lecture

Index

V