Programmation impérative avec C++

Algorithmique et programmation niveau 2

Frédéric Goualard

2021-02-08, v. 0.1

The biggest advantage of learning the ENIAC from the diagrams was that we began to understand what it could and could not do. As a result we could diagnose troubles almost down to the individual vacuum tube. Since we knew both the application and the machine, we learned to diagnose troubles as well as, if not better than, the engineer. — Betty J. Jennings.
In *The Women of ENIAC*, W. Barkley Fritz, 1996[1].

Figure 1 : Betty J. Jennings, 1924–2011. Crédit photo : Wikipedia.

Écrire du code correct et efficace requiert une connaissance minimale de l'environnement d'exécution d'un programme. Cette connaissance est apportée par la section 1. Je décrirai ensuite dans la section 2 certains éléments importants de la syntaxe et de l'utilisation du langage C++. Sauf exception dûment notée, je considérerai le standard ISO/IEC 14882:2017[2], aussi appelé C++17. La notion de *programmation orientée objet* ne sera abordée que dans les limites nécessaires à l'utilisation de la bibliothèque standard de C++ (STL, *Standard Template Library*). En particulier, je ne montrerai pas dans ce fascicule comment écrire ses propres classes et je restreindrai mon exposé du langage C++ à un sous-ensemble sans classes. L'organisation du matériel présenté fait l'hypothèse d'une connaissance minimale du langage C ou C++ et ce fascicule n'a pas vocation à servir de base pour l'apprentissage initial du C++.

1 De l'écriture à l'exécution

Considérons le programme C++ ci-dessous :

```
#include <iostream>
#include <cmath>

using namespace std;

int x;
int y = 100;

int main(void)
{
   const char *str = "Hello";
   double z = cos(42.0); // z <- cosinus(42.0)
   int *t = new int[10];
   cout << str << endl;
   cout << z << endl;
   cin >> t[0] >> t[1];
   cout << t[0]+t[1] << endl;
}
```

Ce code, lisible par toute personne connaissant le langage C++, ne peut pas être directement exécuté par un processeur car il n'est pas complet (par exemple, le code

[1] W.B. Fritz. "The women of ENIAC". In : *IEEE Annals of the History of Computing* 18.3 (1996), p. 13-28

[2] *ISO/IEC 14882 :2017*. URL : https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/06/85/68564.html

de la fonction `cos()` calculant le cosinus d'un angle n'est pas donné) ; de plus, pour des raisons historiques, les composants d'un ordinateur sont uniquement capables de manipuler de l'information sous forme *binaire*[3] et cela vaut aussi bien pour les données d'un programme que pour le programme lui-même.

[3] Pour ce qui concerne la notion de représentation binaire, on se reportera au fascicule « Représentation des nombres » et à la section 2.1.

1.1 Création d'un fichier exécutable

Un outil comme g++ est responsable de la création du fichier binaire contenant le programme exécutable par le processeur à partir du fichier de code source C++. Appelé par abus de langage « *compilateur* », il s'agit en fait d'un programme servant de frontal à d'autres programmes spécialisés, dont le compilateur proprement dit. La création d'un programme exécutable passe par les étapes suivantes (figure 2) :

- *Pré-processing* : le langage C et le langage C++ offrent des facilités hors du langage lui-même grâce aux *directives de compilation* (reconnaissables par leur nom commençant par un « # » : `#define`, `#include`, `#ifdef`, ...) pour définir des macros et inclure le contenu de fichiers[4]. Le remplacement d'une macro par sa définition dans tous les endroits du code source où elle apparait, ainsi que le remplacement d'une directive `#include` par le contenu du fichier en paramètre, est fait par le *pré-processeur*. Ce programme transforme le code source C++ en un nouveau code source C++ où toutes les directives ont été remplacées (voir l'exemple de la table 1) ;
- *Compilation* : le fichier C++ issu de la phase de pré-processing est d'abord traduit par le compilateur en *langage d'assemblage*, qui est un langage de programmation très simple composé d'instructions ayant une traduction directement compréhensible par le processeur sur lequel sera exécuté le code final. La table 2 montre le code en langage d'assemblage généré pour un processeur de type MIPS32. Chaque type de processeur comprenant un jeu d'instructions différent, le code en langage d'assemblage généré est lui aussi différent ;
- Le programme en langage d'assemblage est contenu dans un fichier texte (main.s dans l'exemple de la figure 2) lisible — et même modifiable directement — par l'utilisateur. Il est ensuite transformé en du code binaire (une longue suite de nombres directement interprétable comme une chaîne de « 0 » et de « 1 ») par l'*assembleur* (figure 3). Le fichier résultant est appelé *fichier objet* (main.o, dans la figure 2). Il n'est pas encore exécutable car il ne contient par le code des fonctions appelées par l'utilisateur mais non définies par lui (la fonction `cos()`, par exemple) ;
- Dans la dernière étape, un programme appelé *éditeur de liens* recherche et agrège au fichier objet le code manquant pour avoir un programme complet. Pour cela, il a besoin de savoir où le chercher. Dans le cas de la fonction `cos()`, son code se trouve dans la bibliothèque mathématique libm.so. On indique cela à l'éditeur de liens en ajoutant l'option « -lm »[5] sur la ligne de compilation. L'éditeur de liens a aussi pour responsabilité de créer un fichier exécutable dans le format attendu par le système d'exploitation utilisé car celui-ci aura la responsabilité de charger en mémoire le programme suivant un protocole précis avant son exécution. Avec le système d'exploitation Linux, le standard pour organiser un fichier exécutable ou un fichier objet s'appelle ELF (*Executable and Linkable Format*).

[4] Si les macros et les directives d'inclusion ont toujours leur importance dans les dernières versions du standard ISO C, elles ont beaucoup perdu de leur intérêt dans les versions récentes du standard C++. L'utilisation de macros est désormais déconseillé en C++ et je m'efforcerai de restreindre leur utilisation dans ce fascicule ; avec la version C++20, les directives d'inclusion sont appelées à être remplacées à terme par la notion de *module*. Comme il s'agit cependant d'un ajout récent au standard qui n'est pas encore bien supporté par tous les compilateurs, je continuerai à utiliser les directives d'inclusion dans ce fascicule.

[5] L'option « -l*xxxxx* » sur la ligne de compilation de g++ demande d'utiliser la bibliothèque lib*xxxxx*.so ou lib*xxxxx*.a pour trouver le code manquant au programme compilé. Le chemin d'accès à ces fichiers est déterminé par le contenu des variables d'environnement `LD_LIBRARY_PATH` et `LIBRARY_PATH` respectivement.

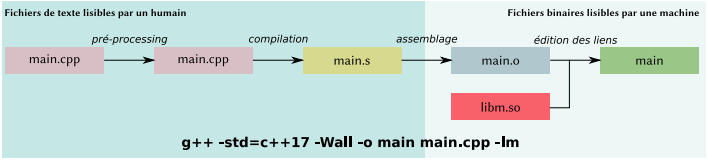


FIGURE 2 : Les différentes étapes dans la création d'un programme exécutable.

Table 1 : Exemple de pré-processing d'un fichier C++. L'option « -E » de g++ arrête le processus de « compilation » après la phase de pré-processing et permet de visualiser à droite le contenu du fichier C++ original de gauche après expansion de toutes les macros.

```cpp
#define USE_MUL true
#define f(a,b) a+b

int main(void)
{
   double x = 1;
#if USE_MUL
   double y = 0.5*x - f(x,7);
#else
   double y = x/2.0 - f(x,7);
#endif
}
```

```cpp
int main(void)
{
 double x = 1;

 double y = 0.5*x - x+7;



}
```

Figure 3 : Code machine MIPS32 pour Hello.s exprimé en hexadécimal.

```
0x0c100003
0x2402000a
0x0000000c
0x24020004
0x3c011001
0x34240000
0x0000000c
0x03e00008
```

Table 2 : Compilation du fichier C++ à gauche en en langage d'assemblage MIPS32 à droite.

```cpp
// Hello.cpp
#include <iostream>

using namespace std;

int main(void)
{
   cout << "Hello World!";
   return 0;
}
```

```
# Hello.s
   .data
msg:   .asciiz   "Hello World!"
   .text
   .globl __start
__start:
   jal main
   li $v0, 10
   syscall
main:
   li $v0, 4
   la $a0, msg
   syscall
   jr $ra
```

1.2 Chargement en mémoire et exécution

La figure 5 présente une vue de haut niveau d'un ordinateur à « *architecture de von Neumann*[6] », composé de trois grandes parties qui interagissent :

- Un processeur, qui exécute le programme se trouvant en mémoire ;
- De la mémoire, qui contient le programme *et* les données sous forme binaire ;
- Des périphériques de masse pour stocker le code et les données sur le long terme.

À ces parties, on peut rajouter des *périphériques* pour que l'ordinateur puisse communiquer avec son environnement — et inversement : clavier, souris, écran, ...

Pour que les différents éléments puissent échanger le plus efficacement possible, deux processeurs spécialisés, les *chipsets* Northbridge et Southbridge ont la tâche exclusive d'interconnecter les différentes parties. Le Northbridge gère les éléments les

Figure 4 : John von Neumann, 1903–1957. Crédit photo : Wikipedia.

[6] L'architecture de von Neumann correspond à l'organisation interne la plus communément utilisée aujourd'hui pour les ordinateurs. L'alternative la plus répandue, appelée *architecture de Harvard*, où les données et le code se trouvent dans des mémoires séparées, est essentiellement rencontrée dans les microcontrôleurs.

plus importants (mémoire, processeur et carte graphique, essentiellement), alors que le Southbridge s'occupe des éléments secondaires (autres périphériques d'entrées/sorties, BIOS, ...).

La figure 6 présente l'intérieur d'un processeur, ici un *bi-cœur*, composé de :

— Deux cœurs ;
— Une *mémoire cache* de niveau 2 (*Level 2*, ou L2).

Un cœur peut, en première approximation, être considéré récursivement comme un petit processeur contenant le circuit chargé d'exécuter les instructions d'un programme et deux *mémoires cache* de niveau 1 (*Level 1*, ou L1) pour le code et les données. Un processeur bi-cœur se comporte donc comme deux processeurs mono-cœurs indépendants.

On constate que l'image simple de la figure 5 (un processeur/une mémoire) se complexifie lorsque l'on regarde à l'intérieur d'un processeur, où l'on trouve deux autres circuits de mémorisation : les caches L1 et L2. Ces trois niveaux de mémoire (L1, L2, RAM) correspondent à la solution adoptée pour résoudre le problème du « *processor-memory performance gap* » illustré dans la figure 7 : au cours du temps, les performances des processeurs ont augmenté beaucoup plus vite que celles de la mémoire RAM disponible. En conséquence, les processeurs passaient de plus en plus de temps à attendre les données nécessaires à un calcul. La mémoire cache de niveau 2 est une mémoire beaucoup plus rapide que la RAM, dans laquelle sont stockées les données et le code utilisés intensément à un moment donné de l'exécution. Un mécanisme complexe d'anticipation permet au processeur de trouver dans le cache les données nécessaires au moment où il en a besoin, sans avoir à les demander à la RAM. L'introduction de la mémoire cache de niveau 1 est une tentative d'obtenir des performances encore meilleures en utilisant une mémoire encore plus rapide que la mémoire cache de niveau 2 et encore plus proche du processeur. On notera dans la figure 6 que le volume de mémoire installé décroît avec sa vitesse pour des raisons de coût : plus une mémoire est rapide et plus elle est chère.

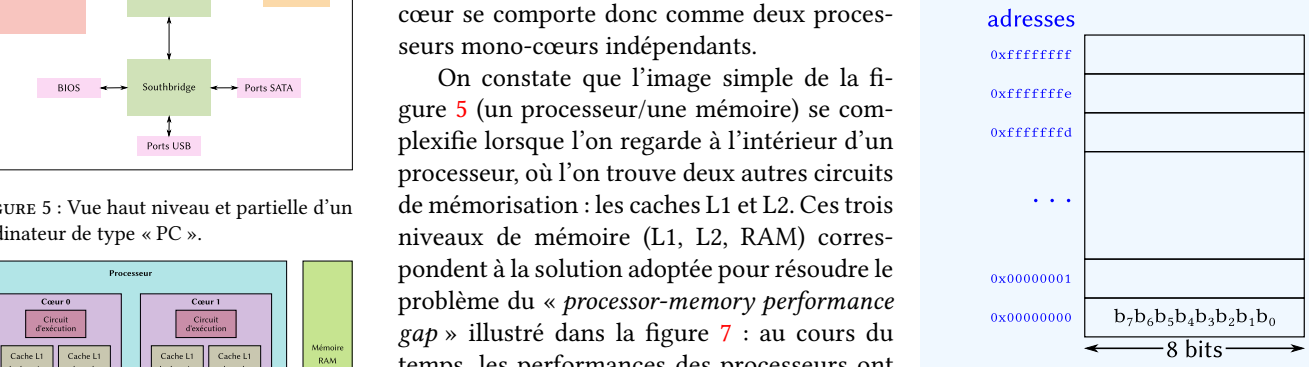
Aujourd'hui, une utilisation optimale des différents caches par les programmes est une condition cruciale pour obtenir de bonnes performances. On verra dans la suite de ce fascicule comment écrire du code pour profiter des caches disponibles.

Pour pouvoir être exécuté par le processeur, un programme doit d'abord être chargé en mémoire à partir du fichier exécutable. Cette tâche est dévolue à un composant spécialisé du système d'exploitation, le *loader*.

Chaque machine possède plus ou moins de mémoire RAM physiquement installée (figure 8). La quantité maximale que peut gérer un système d'exploitation peut être supérieure ou inférieure à la mémoire réellement présente. Par exemple, un *système d'exploitation 32 bits* est capable d'associer une adresse unique à 2^{32} cases d'un octet, soit approximativement 4 GiO. Avec un tel système, il est inutile de posséder plus de 4 GiO de mémoire RAM car la mémoire au-delà de cette limite n'est pas accessible. En théorie, un *système d'exploitation 64 bits* repousse la limite à 2^{64} cases d'un octet, soit une mémoire maximum adressable physiquement de 16 EiO (16 exbioctets). En pratique, les systèmes actuels sont limités à 2^{46} octets pour des raisons d'efficacité, ce qui représente

La mémoire.

Il existe différentes implémentations pour la mémoire RAM offrant des performances différentes. La modélisation reste cependant toujours la même : on considère la RAM comme un tableau dont chaque case peut contenir 8 bits (un *octet*). Chaque case possède une *adresse* correspondant simplement à sa position dans le tableau à partir de l'indice 0 :



Dans la mémoire ci-dessus, chaque case possède une adresse sur 32 bits (ou huit chiffres hexadécimaux), ce qui fait 2^{32} adresses possibles, de 00000000_{16} à $ffffffff_{16}$.

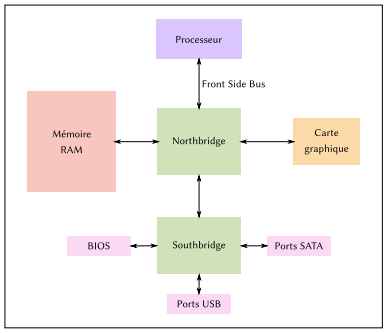


FIGURE 5 : Vue haut niveau et partielle d'un ordinateur de type « PC ».

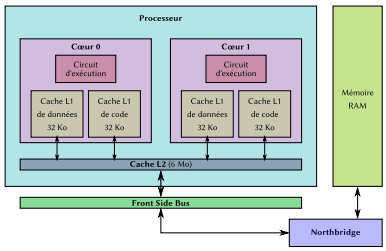


FIGURE 6 : Le processeur et la mémoire.

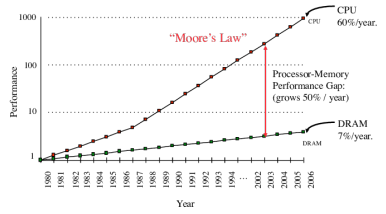


FIGURE 7 : Performances des processeurs *vs.* performances de la mémoire RAM. Crédit photo : [7].

[7] M. BAHI et C. EISENBEIS. "High Performance by Exploiting Information Locality through Reverse Computing". In : *2011 23rd International Symposium on Computer Architecture and High Performance Computing*. Oct. 2011, p. 25-32

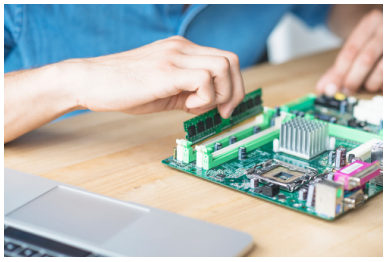


FIGURE 8 : Installation d'un module de mémoire RAM sur une carte-mère. Crédit photo : freepik.

déjà 16384 fois plus de mémoire qu'un système 32 bits. Même si les systèmes 64 bits sont aujourd'hui la norme, les exemples qui suivent considèrent par simplicité un système 32 bits. Les concepts décrits restent les mêmes dans les deux cas, nonobstant quelques variations et complications sans intérêt dans le cadre de ce fascicule.

S'il est possible pour un ordinateur de ne pas pouvoir accéder à de la mémoire physiquement présente, il est, à l'inverse, courant pour un programme s'exécutant sur un système 32 bits de faire référence à une case mémoire dans la limite des 4 GiO, même si cette case n'existe pas physiquement (exemple : un système 32 bits avec seulement 2 GiO installés). Le système d'exploitation offre une représentation virtuelle de la mémoire aux applications, la même pour toutes, où l'intégralité des 4 GiO est mise à leur disposition exclusive, même dans le cas d'un système multi-programmé où plusieurs programmes peuvent s'exécuter en parallèle ou concurremment (figure 9).

Chaque application dispose d'un *espace d'adressage virtuel* de 4 GiO ; cet espace est décomposé en blocs de 4 KiO[8], les *pages*. La mémoire physique est elle-même décomposée en blocs de la même taille, les *cadres de pages*. À tout moment, le système d'exploitation choisit les pages virtuelles des différentes applications exécutées à mettre dans les cadres de pages de la mémoire physique. Les données ou le code d'un programme en mémoire virtuelle qui ne peuvent trouver place en mémoire physique sont stockées en attente sur un périphérique de masse (par exemple, un disque dur).

Le processeur ne peut exécuter que du code se trouvant réellement en mémoire RAM ; de même, il ne peut manipuler que des données se trouvant physiquement en mémoire. Le système d'exploitation doit donc régulièrement mettre à jour les pages se trouvant dans la RAM en fonction des besoins. Si une page ne se trouve pas en mémoire au moment où c'est nécessaire, on a un *défaut de page*. Chaque défaut de page influe négativement sur les performances des programmes.

[8] La taille des pages peut varier en fonction du système utilisé. Sous linux, vous pouvez obtenir la taille de page utilisée avec la commande « `getconf PAGE_SIZE` ».

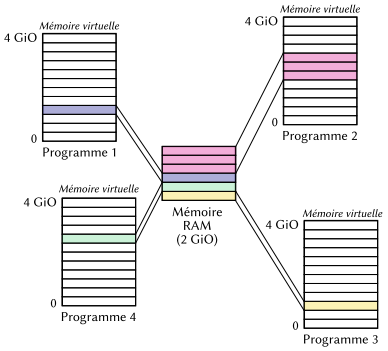


FIGURE 9 : Pagination des mémoires virtuelle et réelle.

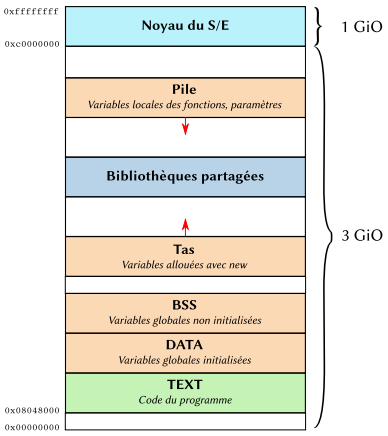


FIGURE 10 : Segmentation de la mémoire virtuelle d'un processus.

La mémoire virtuelle allouée à chaque programme exécuté (*processus*) est découpée en segments (figure 10) ; dans l'énumération qui suit, tous les identifiants font référence à l'exemple de la table 3, page 6 :

- Le segment **TEXT** contient le code du programme ainsi que toutes les chaînes de caractères constantes. Il est situé dans la partie basse de la mémoire. Sa taille est connue lors de l'assemblage et de la génération du code objet. Le code de la fonction `main()` ainsi que la chaîne `"Hello"` seront stockés dans **TEXT** ;
- Le segment **DATA** contient les variables globales initialisées. La variable « y » sera stockée dans **DATA** ;
- Le segment **BSS** contient toutes les variables globales non initialisées. La variable « x » sera stockée dans **BSS** ;
- Le segment de **tas** stocke toutes les données créées dynamiquement avec `new`. Sa taille ne peut être connue à la compilation. Le segment commence avec une taille nulle et croît avec les besoins vers les adresses hautes de la mémoire. Le tableau de dix entiers créé avec la commande « `new int[10]` » sera stocké dans le tas ;
- Lors de l'édition des liens, on peut coller dans le fichier objet le code des fonctions définies dans les bibliothèques pour faire un programme exécutable. Ce type de bibliothèque est appelé *bibliothèque statique* et le code de la bibliothèque rejoint alors le reste du code dans le segment **TEXT**. Une alternative, qui est le choix par défaut sous Linux, est de ne coller à l'édition des liens que les informations indiquant où trouver les bibliothèques contenant le code extérieur et de charger ce code à la demande. Cette solution a un coût en temps à l'exécution puisqu'il faut arrêter le programme pour charger du code supplémentaire en mémoire la première fois que l'on rencontre une fonction externe ; l'avantage est que ce code peut être partagé par tous les programmes s'exécutant en parallèle ou concurremment. On voit par exemple qu'il est moins coûteux en mémoire de partager le code de la fonction `cos()` de la bibliothèque mathématique avec tous les programmes qui l'utilisent que de laisser chaque programme charger sa propre copie du même code. On parle ici de *bibliothèque dynamique*. Grâce la virtualisation de

la mémoire, chaque programme a l'impression d'avoir sa propre copie du code des bibliothèques alors que ce code ne se trouve qu'une seule fois en mémoire physique et qu'il est ensuite mappé dans l'espace virtuel de chaque processus. Le code des bibliothèques dynamiques apparait dans le segment **DLL** de chaque processus. Ici, le code de la fonction `cos()` se trouverait dans ce segment ;
— Le segment de **pile** contient les variables locales à une fonction ainsi que les paramètres qui lui sont passés. On reverra le mécanisme précis dans la section 2.5. La taille de la pile dépend de chaque exécution et ne peut donc être connue à l'avance. Sa taille est nulle au démarrage du processus et elle croît vers les adresses basses de la mémoire. Les variables `str`, `z` et `t` seront stockées dans la pile lors de l'exécution de la fonction `main()` ;
— Le segment **Noyau** contient la partie la plus importante du système d'exploitation : son *noyau*. Ce segment ne peut être ni lu ni écrit par l'utilisateur. Seuls certains appels de fonctions privilégiées permettent d'y accéder.

TABLE 3 : Localisation des variables en mémoire.

```
int x;
int y = 100;

int main(void)
{
   char *str = "Hello";
   double z = cos(42.0);
   int *t = new int[10];
   // [...]
}
```

2 Le langage C++

Toutes les données manipulées au sein d'un programme C++ doivent avoir un type, indiquant à la fois l'ensemble des valeurs possibles et la taille de leur représentation en mémoire. Cette information permet au compilateur de calculer à la compilation et non à l'exécution la taille des segments contenant le code, les données globales initialisées et non initialisées (figure 10).

2.1 Les types fondamentaux

Le langage C++ définit des types fondamentaux et des constructeurs de types pour générer de nouveaux types arbitrairement compliqués à partir de ceux-là. Le mécanisme de *classe* permet aussi de créer de nouveaux types mais je n'en parlerai pas dans ce fascicule.

En C++, il n'existe réellement que quatre types fondamentaux : les *caractères*, les *entiers*, les *réels* et les *booléens*. Pour des raisons d'efficacité, le langage définit cependant plusieurs variantes de tailles et de propriétés différentes. Le tableau 4 présente les différents type avec leur nom et leur taille en nombre d'octets[9].

Notez la différence entre les *entiers non-signés*, qui sont toujours positifs ou nuls et les *entiers signés*, qui peuvent être positifs, négatifs ou nuls. Pour une taille de type entier donné, on peut représenter deux fois plus de nombres positifs avec un type non-signé qu'avec un type signé ; le choix entre, par exemple, `int` et `unsigned int` doit donc se faire en fonction de l'utilisation prévue. Lorsqu'une donnée ne peut être que positive (par exemple, l'âge en années de personnes), utiliser un type non-signé donne

[9] Chaque case de la mémoire faisant un seul octet, tout objet qui nécessite plus d'un octet est stocké dans plusieurs cases adjacentes. L'adresse de l'objet est alors celle de la case de plus petite adresse qui sert à le représenter.

Table 4 : Types fondamentaux du langage C++.

Nom	Taille†	Description
`bool`	1	Booléen (`true` ou `false`)
`char`	1	Caractère
`unsigned char`	1	Entier dans l'intervalle $[0, 255]$
`short int`	2	Entier dans l'intervalle $[-32768, 32767]$
`unsigned short int`	2	Entier non-signé dans l'intervalle $[-32768, 32767]$
`int`	4	Entier dans l'intervalle $[-2^{31}, 2^{31} - 1]$
`unsigned int`	4	Entier non-signé dans l'intervalle $[0, 2^{32} - 1]$
`long int`	8	Entier dans l'intervalle $[-2^{63}, 2^{63} - 1]$
`unsigned long int`	8	Entier non-signé dans l'intervalle $[0, 2^{64} - 1]$
`size_t`	8	Entier non-signé dans l'intervalle $[0, 2^{64} - 1]$
`float`	4	Nombre « réel » simple précision
`double`	8	Nombre « réel » double précision
`long double`	16	Nombre « réel » en précision étendue

† Taille exprimée en nombre d'octets.

une information supplémentaire au compilateur, qui peut ainsi s'assurer que le code que vous écrivez respecte bien cette spécification. De la même façon, si une variable a pour vocation de contenir une taille de donnée, il est plus approprié de la définir de type `size_t` que de type `unsigned long int`, même si ces deux types sont rigoureusement équivalents. On donne ainsi aux personnes qui reliront le code une information sur l'usage de la variable qui serait perdue avec le type `unsigned long int`.

Une constante caractère doit être entourée d'apostrophes simples pour la différencier d'un nom de variable :

```
char a = 'Z';
char c = 'a'; // c <- 'a'
char d = a;   // d <- 'Z'
```

La taille exacte des types n'est pas précisée par le standard du langage. Les valeurs indiquées dans le tableau 4 correspondent à des tailles classiques avec le compilateur GNU C++ sur un système Linux 64 bits. L'opérateur `sizeof` peut être utilisé pour déterminer le nombre d'octets utilisés pour représenter un type ou une expression :

```
size_t x = sizeof(double); // x <- 8
size_t y = sizeof x; // y <- 8
```

L'opérateur `sizeof` retourne un `size_t`.

Le fichier d'en-tête cstdint offre de nouveaux noms de types et en particulier des noms de types pour les entiers représentés sur un certain nombre de bits :

Nom	Taille (en octets)	Description
`int8_t`	1	Entier signé sur 8 bits
`int16_t`	2	Entier signé sur 16 bits
`int32_t`	4	Entier signé sur 32 bits
`int64_t`	8	Entier signé sur 64 bits
`uint8_t`	1	Entier non signé sur 8 bits
`uint16_t`	2	Entier non signé sur 16 bits
`uint32_t`	4	Entier non signé sur 32 bits
`uint64_t`	8	Entier non signé sur 64 bits

Ces types sont utiles lorsque l'on a besoin d'avoir une garantie sur le domaine représentable.

Aux types déjà présentés, on peut rajouter deux types particuliers :

— `void` : le type représentant un ensemble vide de valeurs. Il s'agit d'un type incomplet et l'on ne peut pas définir de valeur de type `void`. On peut l'utiliser pour indiquer qu'une fonction ne prend pas d'argument ou ne retourne rien :

```
// Une fonction sans paramètre retournant un entier
int main(void);
// Une fonction prenant un entier en paramètre et ne retournant rien
void fun(int);
```

On peut aussi utiliser le type `void*` (*pointeur sur* `void`) pour définir un pointeur sur un objet dont le type est inconnu ;

— `nullptr_t` : le type de l'objet `nullptr`[10], correspondant à un pointeur nul.[11]

Le mot-clé `auto`. Le standard C++11 a introduit la possibilité de laisser le compilateur déterminer le type d'une variable en fonction de ce qui lui est affecté. Il suffit pour cela de remplacer le type par « `auto` » :

```
auto x = 1; // Définition d'une variable x de type `int`
auto y = 3.5f; // Définition d'une variable y de type `float`
```

Ce mot-clé permet souvent de raccourcir et simplifier le code et doit être utilisé dès que cela est possible. Comparez les deux codes équivalents suivants :

```
for (std::vector<int>::const_iterator x = T.begin(); x != T.end(), ++x) {
  // [...]
```

et :

```
for (auto x = T.begin(); x != T.end(), ++x) {
  // [...]
```

Les nombres « réels. » Les ensembles de nombres entiers naturels ou relatifs sont infinis mais il est possible de travailler sans erreur dans le domaine de définition de chaque type entier. À l'inverse, l'ensemble des nombres réels étant indénombrable, aucun intervalle réel, aussi petit soit-il, ne peut être représenté complètement dans un ordinateur dont les ressources sont en nombre fini. C'est pourquoi l'ensemble des réels est approché par un ensemble fini de rationnels représentés par une partie fractionnaire binaire dans l'intervalle $[0, 2)$ et un facteur d'échelle. Le facteur d'échelle induisant un déplacement (« *flottement* ») de la virgule dans la partie fractionnaire, on utilisera le terme « nombre flottant » pour les nommer. La représentation précise en mémoire des nombres flottants ainsi que les propriétés de leurs opérateurs sont définies par le standard IEEE 754[12]. On en trouvera sur Wikipedia une présentation assez bien faite.

Lorsqu'un nombre réel ne peut être représenté, il est remplacé par le nombre le plus proche. De calcul en calcul, on peut très facilement calculer des résultats très éloignés de la vraie valeur réelle.

L'exemple de la figure 11 montre le type de problème qui peut arriver lorsque l'on utilise les nombres flottants, quelque soit leur précision : la valeur 0.1 ne peut pas être représentée exactement par un nombre flottant binaire. Elle est donc remplacée par une approximation représentable. Lors de l'ajout à chaque tour de boucle de la valeur 0.1, l'erreur est répétée et l'on n'a aucune garantie que la boucle sera bien répétée mille fois. D'ailleurs, il vous suffit de tester ce programme sur votre machine pour constater qu'il ne s'arrête pas.

Les nombres flottants sont proposés en trois précisions : simple, double et étendue. Plus la précision est grande et plus le type peut représenter des nombres petits et des nombres grands. Pour un usage général, le type `double` est conseillé. La précision du

[10] Herb Sutter et Bjarne Stroustrup. *A name for the null pointer : nullptr*. Rapp. tech. SC22/WG21/N2431. Oct. 2007

[11] Avant le standard C++11, le langage C++ utilisait la macro `NULL` déjà présente dans le langage C (mais avec une définition différente). Le langage C++ imposant des contraintes de typage plus fortes que le C et autorisant la *surcharge* des fonctions, la nécessité d'avoir un vrai pointeur nul a entraîné la création de `nullptr`. Il est désormais fortement recommandé de ne pas utiliser `NULL` pour cet usage.

[12] *IEEE Standard for Floating-Point Arithmetic*. Rapp. tech. IEEE Std 754-2019 (Revision of IEEE 754-2008). Juil. 2019

Figure 11 : Erreurs d'arrondis en calcul flottant.

```
#include <iostream>
using namespace std;

int main(void)
{
   double accu = 0.0;
   unsigned int ntours = 0;

   while (accu != 100.0) {
      accu += 0.1;
      ntours += 1;
      // Affichage de `ntours`
      // tous les 100 tours.
      if (ntours % 100) {
         cout << ntours << endl;
      }
   }
   cout << ntours << endl;
}
```

type `long double` dépend de la plateforme utilisée et peut éventuellement ne pas être supérieure à celle du type `double`. Sauf pour des applications où l'espace mémoire serait le point de contention, il est inutile d'utiliser le type `float`.

2.2 Pointeurs et références

Pour chaque type T, fondamental (section 2.1) ou composé (section 2.3), on peut créer un type « *pointeur sur le type* T » ou « *référence sur le type* T » :

```
int *x; // Pointeur sur entier
```

Ici, la variable `x` peut contenir une adresse en mémoire. À cette adresse, on trouvera un entier de type `int`. On pourrait écrire, par exemple :

```
int a = 5;
int *x = &a;
```

L'esperluette « & » se lit « *adresse de* ». En écrivant `&a`, on prend donc l'adresse en mémoire de la variable `a`. Comme `a` est de type `int`, `&a` est de type `int*`. La figure 12 présente l'organisation en mémoire pour cet exemple si l'on suppose que la variable `a` se trouve à l'adresse 23456789_{16} en mémoire. On est ici avec un système 32 bits, donc toutes les adresses sont représentées avec 8 chiffres hexadécimaux (voir fascicule « représentation des nombres »).

Notez qu'un pointeur est une variable comme les autres (c'est même simplement un entier, qui s'avère correspondre à une adresse en mémoire). Il est donc possible de pointer dessus. On aurait alors :

```
int a = 5;
int *x = &a;
int **y = &x;
```

Notez qu'il vaut mieux coller l'étoile à la variable plutôt qu'au type :

```
int *x = &a; // OUI!
int* x = &a; // NON!
```

Les deux notations sont autorisées mais la première évite de faire l'erreur suivante :

```
int* t, z;
```

Ici, on a déclaré un pointeur sur entier `t` et un *entier* `z`. Cela devient beaucoup plus visible si l'on écrit :

```
int *t, z;
```

La variable `y` est de type « *pointeur sur pointeur sur entier* ». Si l'on veut afficher la valeur de `a` en passant par `y`, il faut doublement déréférencer le pointeur :

```
cout << y << endl;   // Affiche 0x2345678d
cout << *y << endl;  // Affiche 0x23456789
cout << **y << endl; // Affiche 5
```

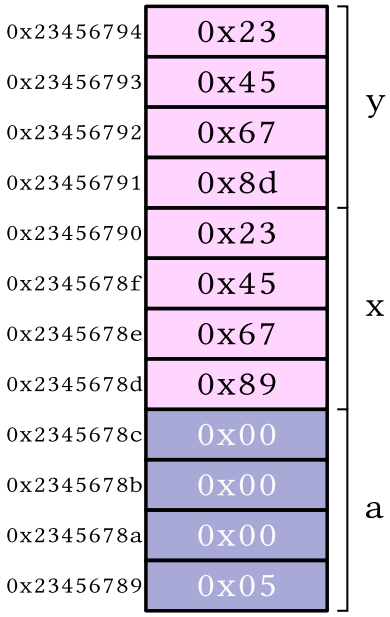
Une *référence* est un alias d'une autre variable. Par conséquent, on ne peut définir une référence sans lui donner de valeur, contrairement à un pointeur :

```
int& r; // NON!
int s;
int& t=s;
```

La variable `t` est une référence sur `s`. Elle agit comme un autre nom de cette variable :

```
int s = 4;
int& t = s;
++s;
cout << t << endl; // Affiche 5
```

FIGURE 12 : Pointeur et objet pointé (système 32 bits).



2.3 Les types composés

Il est possible de construire des objets compliqués à partir des types de bases vus dans la section 2.1. Le C++ offre pour cela les mécanismes suivants :

- Les *tableaux*, pour agréger des données de même type ;
- Les *énumérations*, pour définir un ensemble de constantes ;
- Les *structures*, pour agréger des données de types différents ;
- Les *unions*, pour avoir des données dont le type peut être choisi à l'exécution parmi plusieurs alternatives prédéfinies.

2.3.1 Les tableaux

Un tableau correspond à la juxtaposition d'éléments de même type. Chaque élément est accessible par son indice, le premier élément se trouvant à l'indice 0.

Il existe trois manières, décrites ci-après, de créer un objet de type tableau.

Les tableaux à la C/C++. On peut créer un tableau à partir de n'importe quel type en spécifiant la taille du tableau entre crochet. On peut initialiser le tableau en fournissant les différents éléments entourés d'une paire d'accolades et séparés par des virgules. Si le tableau est initialisé, il n'est pas nécessaire de fournir sa taille, qui sera calculée à partir de l'initialiseur.

```
// Création d'un tableau de 12 entiers
int T[12]; // T[0] ... T[11]
// Création et initialisation d'un tableau de 4 doubles
double V[4] = { 1.0, 3.5, 4.5, 8.0 };
// Création et initialisation d'un tableau de 5 entiers
size_t M[] { 2, 6, 7, 1, 5 }; // M de taille 5
```

Ce type de tableau, partagé avec le langage C, a l'avantage de la simplicité. Il a l'inconvénient de n'offrir aucune sécurité (il est possible d'accéder en lecture ou en écriture à l'extérieur du tableau) et de ne pas être auto-descriptif. En particulier, on ne peut connaitre la taille d'un tableau sans l'accompagner d'une variable entière pour la sauvegarder.

Un tableau est une structure de données efficace car tous ses éléments sont contigus en mémoire, ce qui autorise une bonne utilisation de la mémoire cache. De plus, l'accès à n'importe quel élément se fait en temps constant[13]

Une chaîne de caractères est un tableau de caractères, avec la convention que le dernier caractère de la chaîne doit être le caractère spécial `'\0'`. Une chaîne de caractères entre guillemets est interprétée comme un tableau de caractères avec le dernier caractère — invisible — égal à `'\0'` :

```
char chaine1[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char chaine2[] = "Hello";
```

Dans cet exemple, `chaine1` et `chaine2` sont deux chaînes de caractères de taille 6 qui sont rigoureusement équivalentes.

Le nom d'un tableau est aussi un pointeur sur sa première case. On verra donc dans la section 2.5.1 qu'il n'est pas possible de passer un tableau C/C++/ par copie à une fonction. De même, la copie d'un tableau dans un autre ne peut se faire par une simple affectation des noms :

```
int T1[] {1, 2, 3};
int T2[] {4, 5, 6};
T1 = T2; // NON!
```

Une telle tentative d'affectation sera détectée à la compilation et génèrera une erreur.

[13] En théorie, l'accès à un élément d'un tableau C/C++ se fait en temps constant quelle que soit la position de l'élément dans le tableau. En pratique, le temps d'accès peut être sensiblement différent, en particulier pour de très gros tableaux qui ne tiendraient pas entièrement en mémoire et où on pourrait rencontrer des *défauts de pages* lors de l'accès à certains éléments.

Le type paramétré `array<>`. Le type paramétré `array<>`, propre à C++, sert à définir des tableaux dont la taille est connue. Pour utiliser ce type, il faut inclure le fichier d'en-tête `array` :

```
#include <array>
int main(void)
{
  std::array<double,3> T {1.0, 2.5, -7.5};
}
```

Les paramètres du type `array` sont :
— Le type de chaque élément du tableau ;
— Le nombre d'éléments du tableau.

Contrairement aux tableaux C/C++, il est facile de connaitre la taille d'un `array` avec la *méthode* `.size()` :

```
std::array<int,6> x;
cout << x.size() << endl; // Affiche 6
```

La perte de performance d'un `array` par rapport à un tableau C/C++ est minimale. Il est plus cependant facile de connaitre sa taille. Par ailleurs, un `array` peut être passé par valeur en paramètre d'une fonction.

Le type paramétré `vector<>`. Le type `vector<>` offre la plus grande souplesse pour créer des tableaux et doit être préféré lorsque ses performances légèrement inférieures aux deux premières solutions ne posent pas de problème.

Le type `vector<>` est paramétré par le type des éléments. Il est possible de créer un tableau d'une certaine taille (éventuellement nulle) et de rajouter des éléments par la suite. Une telle extension de tableau a un coup, amorti cependant en partie par des astuces de son implémentation.

Je vous renvoie à la documentation pour connaître toutes les méthodes associées au type `vector` et je me contenterai ici de présenter rapidement son usage et certains de ses dangers.

Pour utiliser le type `vector`, il faut inclure le fichier d'en-tête `vector`. On peut ensuite créer des tableaux sans éléments ou déjà initialisés :

```
// Un tableau vide d'entiers
std::vector<int> T1;
// Un tableau de doubles initialisé par un tableau C/C++
std::vector<double> T2 { 2.5, 3.5, -7.5};
// Un tableau de doubles de 6 éléments tous initialisés à -1.0
std::vector<double> T3(6,-1.0);
```

L'accès à une case d'un tableau peut se faire de deux façons :

```
std::vector<double> T2 { 2.5, 3.5, -7.5};
// Comme un tableau C/C++
cout << T2[2] << endl;
// Avec la méthode at()
cout << T2.at(3) << endl;
```

La première méthode est la plus rapide mais n'offre aucun contrôle sur la légitimité de l'accès. La deuxième méthode est plus lente mais lève une exception si l'on essaye d'accéder à un élément qui n'existe pas.

Une erreur fréquente est d'essayer d'accéder à une case d'un tableau qui n'existe pas encore :

```
std::vector<double> T2 { 2.5, 3.5, -7.5};
T2[3] = 4.5; // NON!
```

Le tableau `T2` ne possède que trois cases. On ne peut pas essayer d'accéder à la case 3 qui n'existe pas. Selon les cas, cela génèrera une erreur immédiate ou pas (l'utilisation de la méthode `at()` générerait systématiquement une erreur). Si l'on n'a pas déjà créé les cases à la construction, il faut utiliser la méthode `.push_back()` pour ajouter à la fin du tableau :

```
std::vector<double> T2 { 2.5, 3.5, -7.5};
T2.push_back(4.5);
cout << T2[3] << endl; // Affiche 4.5
```

Si l'on ne connait pas la taille du tableau à sa définition mais que l'on a plus tard une information sur sa taille, il est possible de créer toutes les cases nécessaires d'un seul coup avec la méthode `.reserve()`, ce qui est beaucoup efficace que de laisser `.push_back()` ajouter chaque nouvelle case l'une après l'autre :

```
vector<int> T;
// [...]
T.push_back(5);
// On sait maintenant que le tableau aura 1000 cases
T.reserve(1000);
T[1] = 4;
T[2] = -1;
// [...]
```

2.3.2 Les énumérations

Les énumérations permettent de définir des ensembles symboliques en associant des noms à des entiers. En C++, il existe deux types de définitions avec des caractéristiques différentes.

La première méthode, rend visible la dualité « constante symbolique/entier » :

```
enum ville_t {
  londres,        // 0
  paris,          // 1
  bruxelles = 5,  // 5
  amsterdam       // 6
};
```

Dans cet exemple, on définit le type `ville_t` ; une variable de ce type peut prendre comme valeur l'un des quatres noms de villes indiqués. Chaque constante à l'intérieur de l'`enum` a une valeur entière commençant à 0 par défaut et s'incrémentant d'une constante à l'autre. On peut imposer une valeur à l'une des constantes ; dans ce cas, les constantes suivantes dans la liste ont des valeurs qui repartent de cette valeur imposée (exemple : `amsterdam` a la valeur 6 car l'on a imposé la valeur 5 à `bruxelles`, la constante précédente).

On peut utiliser de façon complètement interchangeable le nom symbolique ou la constante entière :

```
ville_t v { bruxelles };
cout << v << endl; // Affiche 5
v = amsterdam - bruxelles;
cout << (v == paris) << endl; // Affiche 1 pour `true`
```

Une variable de type `ville_t` se comportant comme un entier, l'affichage sera aussi celui de l'entier correspondant et non pas le nom symbolique. Si l'on souhaite afficher la chaîne de caractères, il faut surcharger l'opérateur d'affichage « << » pour le type `ville_t` :

```
ostream& operator<<(ostream& os, const ville_t &v)
{
   const char *villestr[] {
     "londres","paris","","","","bruxelles","amsterdam"
   };
   os << villestr[v];
   return os;
}

int main(void)
{
   ville_t v {bruxelles};
   cout << v << endl; // Affiche "bruxelles"
}
```

La deuxième méthode pour définir une énumération est spécifique à C++ :

```
enum class jour_t {
  lundi,     // 0
  mardi,     // 1
  mercredi,  // 2
  jeudi,     // 3
  vendredi,  // 4
  samedi,    // 5
  dimanche   // 6
};
```

La dualité « nom symbolique/entier » existe toujours mais elle est désormais cachée et l'on ne peut plus utiliser une constante comme entier sans faire de *cast* explicite. De même, la définition d'un type avec « `enum class` » crée un espace de nom dans lequel sont insérées toutes les constantes symboliques. On ne peut donc y accéder qu'en les préfixant par le nom du type. Ce petit inconvénient a le grand avantage d'éviter de polluer l'espace de nom global avec pleins de noms de constantes comme le fait une déclaration « `enum` » :

```
jour_t d {jour_t::mardi};
cout << d << endl; // Erreur à la compilation
d += 1; // NON!
d = jour_t(int(d)+1); // OK
```

Il n'est pas possible d'afficher un `enum class` sans le caster en entier ou surcharger l'opérateur « << ».

Quel que soit le type de définition choisi, les type énumérés rendent le code plus clair en évitant le recours à des *constantes magiques* ; comparez :

```
if (d == 2) {
  // [...]
}
```

```
if (d == mercredi) {
  // [...]
}
```

L'autre intérêt est que le compilateur peut repérer des erreurs dans le code car il connaît l'ensemble des valeurs permises pour une variable énumérée. Dans un `switch`, par exemple, il peut avertir le programmeur que tous les cas ne sont pas couverts.

2.3.3 Les structures

Une structure permet d'agréger des éléments de types différents qui font sens ensemble :

```
struct personne_t {
  std::string nom;
  std::string prenom;
  uint8_t age;
};
```

Les différentes variables à l'intérieur de la structure sont ses *champs*. À chaque définition d'un objet de type `personne_t`, on crée trois champs accessibles à partir d'une seule variable :

```
personne_t p;
p.nom = string("Dupont");
p.prenom = string("Jean");
p.age = 24;
```

En mémoire, chaque champ est stocké l'un à la suite de l'autre. Pour des raisons techniques liées à la notion d'*alignement des données en mémoire*, la taille d'une structure peut être plus grande que la somme des tailles de ses champs (figure 13) :

```
#include <iostream>
#include <iomanip>
using namespace std;

struct agregat_t {
   double a;
   int b;
   uint8_t c;
};

int main(void)
{
   agregat_t x { 1.0, -5, 2 };
   cout << sizeof x.a << endl; // Affiche 8
   cout << sizeof x.b << endl; // Affiche 4
   cout << sizeof x.c << endl; // Affiche 1
   cout << sizeof x << endl;   // Affiche 16
}
```

Afin d'avoir une taille de structure qui soit un multiple de 8, le compilateur ajoute trois octets de *padding* pour le champ `c`.

On peut maintenant créer des tableaux de `personne_t`, ce qui est plus simple que de devoir gérer séparément un tableau pour chaque champs :

```
std::vector<personne_t> pT;
pT.push_back(personne_t{"Durand","Paul",32});
```

On aura fréquemment à manipuler des pointeurs sur structures. L'accès aux champs via un pointeur peut se faire en déréférençant d'abord le pointeur pour accéder au champ par la notation pointée, ou en utilisant la syntaxe fléchée :

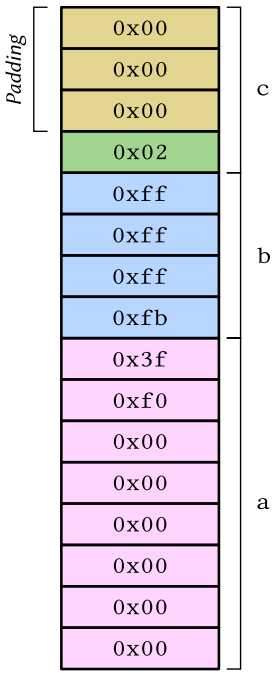
```
personne_t *p = new personne_t{"Dupont","Jean",35};
cout << (*p).nom << endl;
cout << p->prenom << endl;
```

Notez que l'étoile ayant une priorité moins élevée que le point, le code suivant ne fonctionnera pas :

```
cout << *p.age << endl; // NON!
```

La notation « `*p.age` » est équivalente à « `*(p.age)` » et non à « `(*p).age` ».

FIGURE 13 : Représentation d'un `agregat_t` en mémoire.



2.3.4 Les unions

Comme une structure, une union définit un nouveau type avec des champs. À la différence d'une structure, les champs d'une union sont mutuellement exclusifs : lorsque l'on utilise un des champs, on n'utilise pas les autres. Considérez l'exemple de la table 5 : on veut définir un type `animal_t` pour représenter des animaux ; un animal a un nom et un âge exprimé en années ou en mois. Si l'âge est exprimé en années, on utilise un entier sur 8 bits car il est peut probable d'avoir un animal de plus de 255 ans ; si l'âge est exprimé en mois, on utilise un entier sur 16 bits. On décide de stocker l'âge d'une manière ou de l'autre exclusivement. On ne veut donc pas définir le type `animal_t` par :

```
struct animal_t {
  string nom;
  uint8_t n_ans;
  uint16_t n_mois;
};
```

car on représenterait en mémoire les deux champs `n_ans` et `n_mois` alors qu'un seul serait utilisé. D'ailleurs, pour savoir quel champs a été initialisé, il faudrait rajouter une information supplémentaire :

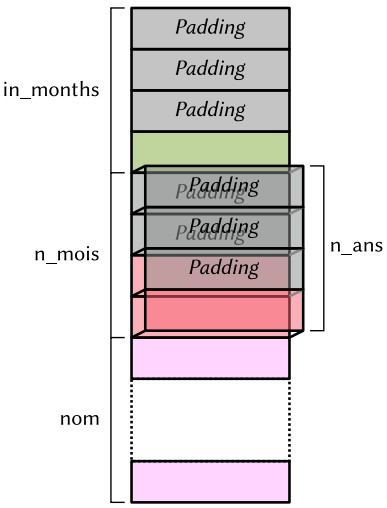
```
struct animal_t {
  string nom;
  uint8_t n_ans;
  uint16_t n_mois;
  bool age_en_mois;
};
```

Le champ `age_en_mois` est à `true` s'il faut utiliser le champ `n_mois` et à `false` si c'est le champ `n_ans` qui importe.

En utilisant une union comme dans le programme de la table 5, on demande au compilateur d'utiliser le même espace mémoire pour tous les champs de l'union (figure 14). La taille de l'union correspond donc à la taille du champ le plus grand, sans compter le *padding* éventuellement nécessaire pour des questions d'alignements des données en mémoire. On est aussi obligé de rajouter le booléen `in_months` pour savoir quel champ est utilisé.

Une autre possibilité, illustrée par la définition du type `vanimal_t` est d'utiliser le type paramétré `variant<>`, en incluant l'en-tête variant. Les deux solutions ont sensiblement les mêmes avantages et inconvénients. Le choix entre les deux reste donc une affaire de goût[14].

FIGURE 14 : Représentation de l'union `animal_t` en mémoire.



2.4 Le *linkage*

Seuls les programmes d'exemple à portée pédagogique sont écrits dans un seul fichier. La bonne pratique pour toute application sérieuse est de la décomposer en un ensemble de fichiers compilés séparément. Chaque fichier rassemble les éléments communs à une fonctionnalité. Cela permet de développer l'application à plusieurs facilement, de tester séparément les différentes parties et cela réduit le temps de développement de l'application en limitant les recompilations aux seuls fichiers qui ont été modifiés. Cette section présente les éléments à connaitre pour pouvoir développer ce type d'applications.

La première chose importante à savoir est la différence entre une *déclaration* et une *définition* :

- Une *déclaration* annonce au compilateur l'existence d'un certain nom (de variable, fonction, ...) et son type ;
- Une *définition* décrit en détail un objet. Elle donne le code d'une fonction, les champs d'un type structuré ou la valeur initiale d'une variable, par exemple. Une définition est aussi une déclaration.

[14] Il faut aussi tenir compte du support apporté par le compilateur C++ utilisé : le type `variant<>` a été introduit avec le standard C++17 et peut ne pas être reconnu par certains compilateurs anciens.

TABLE 5 : Utilisation de union et variant<>.

```cpp
#include <iostream>
#include <string>
#include <variant>
using namespace std;

struct animal_t {
   string nom;
   union {
      uint8_t n_ans;
      uint16_t n_mois;
   } age;
   bool in_months;
};

struct vanimal_t {
   string nom;
   variant<uint8_t,uint16_t> age;
};

int main(void)
{
   animal_t rex {"Rex", {.n_ans=3},false};
   animal_t medor {"Medor", {.n_mois=38},true};

   vanimal_t azor {"Azor", {uint16_t{38}}};
   vanimal_t minouche {"Minouche", {uint8_t{7}}};

   if (rex.in_months) {
      cout << rex.nom << ": " << rex.age.n_mois << " mois." << endl;
   } else {
      cout << rex.nom << ": " << int(rex.age.n_ans) << " ans." << endl;
   }

   if (holds_alternative<uint16_t>(azor.age)) {
      cout << azor.nom << ": "
           << get<uint16_t>(azor.age) << " mois." << endl;
   } else {
      cout << azor.nom << ": "
           << int(get<uint8_t>(azor.age)) << " ans." << endl;
   }
   cout << sizeof rex << endl;
   cout << sizeof azor << endl;
}
```

Quelques exemples de déclaration :

```
// Déclare l'existence d'un type énuméré `couleur_t`
enum couleur_t;
// Déclare l'existence d'une fonction `sqr`
double sqr(double x);
// Déclare l'existence d'une variable entière `v`
extern int v;
```

Pour les types non fondamentaux, une déclaration ne permet pas de connaitre l'espace mémoire requis pour représenter l'objet. Cette information n'est cependant pas toujours requise.

Quelques exemples de définition :

```
// Définition du type énuméré couleur_t
enum couleur_t {bleu, rouge, vert};
// Définition de la fonction `sqr`
double sqr(double x)
{
  return x*x;
}
// Définition de la variable entière `v`
int v = 5;
```

On peut déclarer un objet autant de fois que nécessaire. Par contre, on ne peut définir un objet qu'une seule fois dans l'application[15].

— Note importante —

Le compilateur lit chaque fichier de haut en bas ; un *identifiant* (nom de variable, fonction, type, ...) ne peut pas apparaître dans le code s'il n'a pas déjà été déclaré auparavant dans le fichier.

Il existe deux types de fichiers C++ :

— Les fichiers d'*en-tête*, qui ne contiennent que des déclarations ;

— Les fichiers de code, qui contiennent des déclarations et des définitions.

Un fichier d'en-tête (d'extension « .hpp » pour un fichier C++, mais il peut aussi avoir parfois l'extension « .h », voire pas d'extension du tout pour les fichiers de la bibliothèque standard) sert d'interface entre le fichier de code qui contient les définitions et un autre fichier de code qui souhaite utiliser ce que le premier définit.

Un fichier de code C++ a généralement l'extension « .cpp », mais d'autres possibilités existent (« .cc », « .cxx », ...).

Dans la figure 15, le fichier ficA.cpp définit la variable globale `v_globale` et la fonction `fun()` ; le fichier ficA.hpp déclare ces deux identifiants. Le fichier ficB.cpp utilise la fonction `fun()` et la variable `v_globale` ; ces deux identifiants doivent donc avoir été déclarés avant. C'est le cas grâce à la ligne `#include "ficA.hpp"`[16], qui sera remplacée par le pré-processeur par le contenu du fichier ficA.hpp avant la compilation. Pour la compilation de ficB.cpp, le compilateur n'a pas besoin des définitions de `v_globale` et `fun()` ; il n'y a donc aucune relation entre ficA.cpp et ficB.cpp, et le fichier ficA.cpp peut être modifié comme l'on veut pendant le développement de ficB.cpp tant que l'*interface* (l'ensemble des identifiants déclarés dans ficA.hpp) reste la même. L'accès au code de ficA.cpp est nécessaire seulement au moment de l'édition des liens pour la création du fichier exécutable ficB. Dans l'exemple de la figure 15, on a compilé séparément le fichier ficA.cpp en le fichier objet ficA.o et c'est ce fichier que l'on fournit à l'éditeur de liens pour créer ficB.

[15] Les règles précises concernant le nombre de définitions autorisées varient en fonction du type des objets. Je considérerai par simplification que tout objet doit être défini une seul fois et je vous renvoie vers des documents plus complets pour les règles précises.

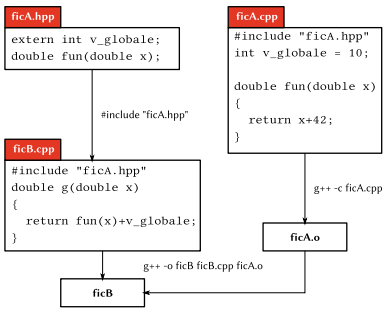


FIGURE 15 : Compilation séparée et fichier d'en-tête

[16] On a vu précédemment des fichiers d'en-tête, par exemple iostream, inclus avec la directive `##include` suivie du nom du fichier entre « <> ». Lorsque que le nom d'un fichier est entre « <> », le pré-processeur le cherche dans une liste de chemins standards qui dépend du compilateur. Avec g++, la liste de chemins peut être modifiée par la variable d'environnement `CPLUS_INCLUDE_PATH`. Lorsque le nom est entre guillemets, la recherche du fichier se fait à partir du chemin du fichier qui contient la directive `##include`.

Notez que le fichier ficA.cpp inclut lui aussi le fichier A.hpp. Ce n'est pas strictement nécessaire ici mais c'est une bonne pratique de faire en sorte que chaque fichier de code source inclut systématiquement son fichier d'en-tête car cela permet au compilateur de vérifier la cohérence de l'interface avec ce qui est implémenté et de prévenir le programmeur en cas de différence (modification de l'interface mais oubli de modifier le code source, par exemple).

Dans la figure 15, on voit qu'il suffit pour ficB.cpp d'avoir une déclaration de `fun()` et `v_globale` pour pouvoir les utiliser. Lorsque l'on écrit une application découpée en de multiples fichiers de code source, il suffit de définir une fonction dans un seul fichier pour pouvoir l'utiliser dans tous les autres fichiers (à condition de donner sa déclaration dans chaque fichier qui l'utilise). Les fonctions et les variables globales ont un *linkage* externe (on peut les utiliser à l'extérieur du fichier qui les définit).

Lorsque l'on parle du *linkage* d'un identifiant, on ne considère pas exactement la notion de fichier mais celle de d'*unité de compilation*. Une *unité de compilation* est le contenu d'un fichier après pré-processing (inclusion de tous les fichiers d'en-tête, expansion des macros). Lorsqu'un identifiant a un *linkage* interne, cela veut dire qu'il n'est visible que dans l'unité de compilation où il est défini. Le mot-clé `static` permet d'imposer le linkage externe à un identifiant :

Les fichiers d'en-tête.

Un fichier d'en-tête a vocation à être inclus par tous les fichiers nécessitant les déclarations qu'il contient. Il peut devenir alors difficile de s'assurer qu'un fichier ne sera pas inclus plusieurs fois, directement et indirectement : si a.hpp inclus b.hpp et que c.cpp inclus à la fois a.hpp et b.hpp, le fichier b.hpp se retrouve deux fois dans c.cpp). La solution classique à ce problème est de mettre des *gardes* à l'intérieur de chaque fichier d'en-tête :

```
#ifndef __ficA_hpp__
#define __ficA_hpp__

// Déclarations du fichier ficA.hpp

#endif // __ficA_hpp__
```

La première fois que l'on inclue le fichier ficA.hpp, la macro `__ficA_hpp__` n'est pas définie. On la définit donc et l'on inclut les déclarations. Toutes les fois suivantes, la macro `__ficA_hpp__` est déjà définie et le pré-processeur saute donc à la fin du fichier sans rien inclure.

De nombreux compilateurs proposent aussi d'insérer le *pragma* « once » au début du fichier d'en-tête :

```
#pragma once
// Déclarations du fichier ficA.hpp
```

Cette solution est cependant à éviter car ce pragma ne fait pas partie du standard C++ et possède certains inconvénients (voir la discussion sur StackOverflow).

```
// Fichier ficD.cpp
static int x = 10;
```

```
// Fichier ficE.cpp
#include <iostream>

extern int x;

int f(void)
{
   return x + 2;
}

int main(void)
{
   std::cout << f() << std::endl;
}
```

Le fichier ficD.cpp définit une variable globale avec l'attribut `static`. La variable `x` a donc un linkage interne au fichier ficD.cpp et ne peut pas être utilisée à l'exterieur. Le fichier ficE.cpp déclare la variable `x`. Lors de la compilation de chaque fichier, tout se passera bien :

```
$ g++ -c ficD.cpp
$ g++ -c ficE.cpp
```

Lors de l'édition des liens pour réunir tous les codes en un seul exécutable, on aura une erreur car l'éditeur de liens ne trouvera pas de variable x à utiliser dans ficE.cpp :

```
$ g++ -o ficE ficE.o ficD.o
ficE.o: In function `f()':
ficE.cpp:(.text+0x6): undefined reference to `x'
collect2: error: ld returned 1 exit status
```

L'utilisation à bon escient du linkage interne évite de polluer l'espace de noms global avec des fonctions ou des variables qui ont vocation à n'être utilisées que dans une seule unité de compilation.

2.5 Les fonctions

Lors de l'appel d'une fonction, les paramètres sont empilés sur la pile (voir la section 1.2). L'ordre d'évaluation des paramètres avant leur empilement n'est pas spécifié par le standard C++ et dépend donc du compilateur utilisé :

```
#include <iostream>
using namespace std;

int g(int x)
{
   cout << x;
   return x;
}

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

int main(void)
{
   cout << f(g(1),g(2)) << endl;
}
```

L'exécution du programme ci-dessus peut aussi bien afficher « 123 » que « 213 ». Il est donc important de s'assurer que la valeur des paramètres ne dépend pas de l'ordre dans lequel ils sont considérés pour éviter des bugs difficiles à trouver.

2.5.1 Les différents types de passage des paramètres

Lors d'un appel de fonction, chaque paramètre est copié sur la pile, puis le contrôle est passé à la fonction appelée. La fonction appelée alloue alors un espace sur la pile pour stocker ses variables globales : le *cadre de pile*. Juste avant de rendre la main à la fonction appelante, la fonction appelée détruit son cadre de pile.

Passage par copie. Considérons le cas d'une fonction prenant un paramètre par copie :

```
#include <iostream>

int f(int x)
{
   int y = x+1;
   return y;
}
```

```
int main(void)
{
   int a = 10;
   std::cout << f(a) << std::endl;
}
```

La figure 16 montre l'évolution de la pile avant, pendant et après l'appel de `f(x)` Lors de l'appel de `f()`, la fonction `main()` stocke directement sur la pile la valeur de `a`[17] ; la fonction `f()` alloue ensuite quatre octets de plus pour stocker sa variable local `y`. À la fin de l'appel, la fonction `f()` ramène le pointeur de pile à sa position initiale, ce qui a pour effet de « détruire » son cadre de pile ainsi que l'espace alloué aux paramètres.

[17] Notez que, comme indiqué dans la section 1.2, la pile croît du haut de la mémoire vers le bas.

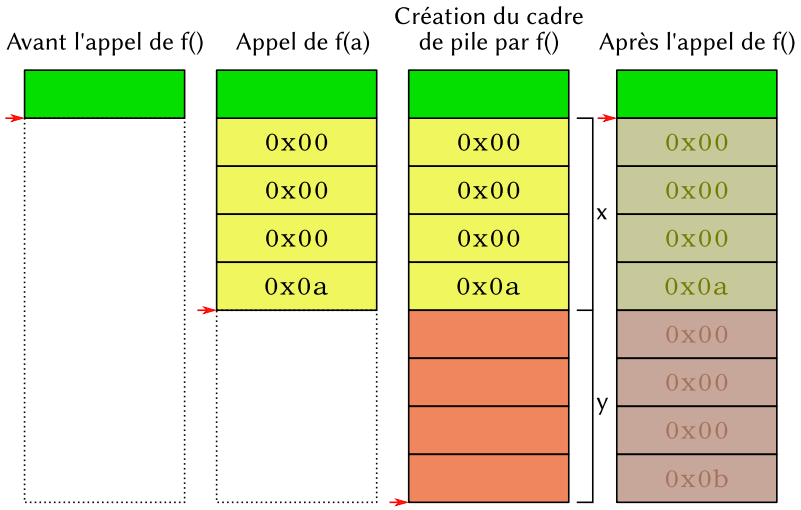


FIGURE 16 : Passage des paramètres par copie pour la fonction `f()`. La flèche rouge pointe sur la dernière case occupée dans la pile.

Passage par adresse. Pour le passage par adresse, on a les mêmes mécanismes que l'appel par copie mais on empile l'adresse d'une variable plutôt que sa valeur :

```
#include <iostream>

int g(int *x)
{
   int y = (*x) + 1;
   (*x) = 12345;
   return y;
}

int main(void)
{
   int a = 10;
   std::cout << g(&a) << std::endl;
}
```

Désormais, comme on le voit dans la figure 17, la fonction `main()` empile l'adresse de sa variable locale `a` avant d'appeler la fonction `g()`. L'adresse $b2345678_{16}$ se trouve plus haut dans la pile car `a` est une variable locale de `main()` et se trouve donc dans son

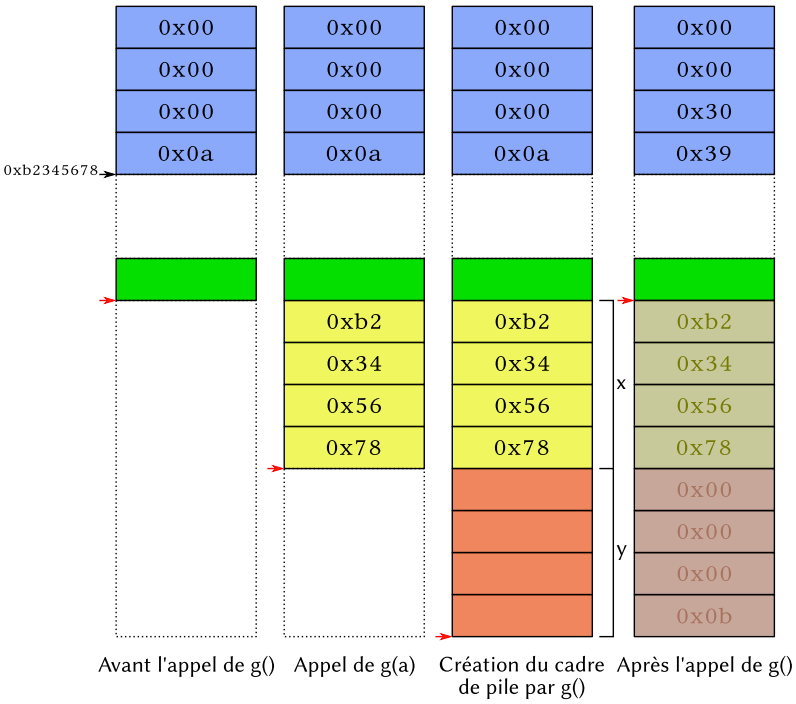


FIGURE 17 : Passage des paramètres par adresse pour la fonction `g()`. La flèche rouge pointe sur la dernière case occupée dans la pile. On suppose une architecture 32 bits (pointeurs sur quatre octets).

propre cadre de pile. Dans le code de `g()`, on doit déréférencer le paramètre formel `x` pour accéder à la valeur de `a` (en bleu dans la figure). Ce faisant, on accède directement à la variable `a` elle-même et donc toute modification via le pointeur de la valeur pointée se fait directement sur `a` et non sur une copie.

Dans l'exemple ci-dessus, on dit que la variable `a` a été passée par adresse à `g()`. On pourrait aussi considérer que l'on a passé le pointeur sur entier `&a` par copie. Donc, si l'on a besoin de modifier un pointeur passé en paramètre, il faut passer un pointeur dessus pour que cela soit un passage par adresse du pointeur. Exemple :

```
#include <iostream>
void allouer_tableau(int **ptr, size_t sz)
{
   *ptr = new int[sz];
}

int main(void)
{
   int *T {nullptr};
   allouer_tableau(&T,10);
   // [...]
}
```

On passe en entrée à `allouer_tableau()` un pointeur initialisé à `nullptr` et contenant au retour l'adresse allouée pour stocker un tableau de `sz` entiers. Comme le paramètre `ptr` doit changer de valeur entre l'entrée et la sortie de `allouer_tableau()`, on passe un pointeur sur le pointeur d'entier.

Pour conclure cette section concernant le passage par adresse, souvenons-nous que

les noms de tableaux sont des pointeurs sur leur première case. Donc un tableau est toujours passé par adresse. Les deux codes ci-dessous sont équivalents :

```
int cumul(int T[], size_t sz)
{
   int accu {0};
   for (size_t i = 0; i < sz; ++i) {
      accu += T[i];
      T[i] = 0;
   }
   return accu;
}
```

```
int cumul(int *T, size_t sz)
{
   int accu {0};
   for (size_t i = 0; i < sz; ++i) {
      accu += *(T+i);
      *(T+i) = 0;
   }
   return accu;
}
```

Une bonne pratique lorsque l'on écrit une fonction prenant en entrée un tableau qui n'est pas censé être modifié dans le corps de la fonction est de déclarer le tableau avec l'attribut « `const` » pour être prévenu par le compilateur si cette contrainte est violée :

```
int cumul(const int T[], size_t sz)
{
   int accu {0};
   for (size_t i = 0; i < sz; ++i) {
      accu += T[i];
      // NON! Erreur à la compilation
      T[i] = 0;
   }
   return accu;
}
```

```
int cumul(const int *T, size_t sz)
{
   int accu {0};
   for (size_t i = 0; i < sz; ++i) {
      accu += *(T+i);
      // NON! Erreur à la compilation
      *(T+i) = 0;
   }
   return accu;
}
```

Une alternative possible est de remplacer le tableau C/C++ par un `array<>`, car celui-ci est toujours passé par copie.

Passage par référence Le passage d'un paramètre par référence ressemble à la fois au passage par copie, dans le corps de la fonction, et au passage par adresse, en ce que le paramètre formel est un alias pour le paramètre effectif :

```
void echanger(int &x, int &y)
{
   int t = x;
   x = y;
   y = t;
}

int main(void)
{
   int a = 3, b = 4;
   echanger (a,b);
   echanger (3,5); // NON! Erreur à la compilation
}
```

Lors du premier appel de la fonction `echanger()` les paramètres formels `x` et `y` sont des alias pour les paramètres effectifs `a` et `b` : tout accès à `x` fait directement référence à `a` et de même pour `y` et `b`. L'implémentation de l'aliasing se fait généralement par l'utilisation de pointeurs et un déréférencement automatique dans le corps de la fonction. C'est pourquoi il n'est pas permis de passer en paramètre un objet qui n'aurait pas d'adresse. L'appel `echanger(3,5)` est une erreur qui sera reconnue lors de la compilation.

3 Licence de ce document

Les remarques, suggestions et indications d'erreurs peuvent être transmises à l'auteur : frederic.goualard@univ-nantes.fr.

— Note importante —

Lors de l'écriture de fonction prenant en entrée un paramètre par copie dont le type a une taille non négligeable, il vaut mieux remplacer le passage par copie par un passage par référence constante : le code de la fonction reste identique mais cela évite le coût de la copie du paramètre sur la pile :

```
struct personne_t {
   string nom;
   string prenom;
   unsigned int age;
   string adresse;
};

void fun(personne_t p)
{
   // [...]
}
```

```
struct personne_t {
   string nom;
   string prenom;
   unsigned int age;
   string adresse;
};

void fun(const personne_t& p)
{
   // [...]
}
```

Les deux codes ci-dessus sont équivalents mais celui de droite est plus efficace.

***L'attribut* `const`.**

Le mot-clé « `const` » ajouté à un type informe le compilateur que l'objet concerné ne doit pas être modifié. Le processus de compilation s'arrête avec une erreur si le programme contient du code qui modifie l'objet :

```
// Une variable entière
int x = 1;
// Une constante entière
const int y = 10;
// Un pointeur sur constante
const int *z = &y;
// Un pointeur constant sur variable
int *const t = &x;
// Un pointeur constant sur constante
const int *const u = &y;
```

Notez la différence entre le pointeur sur constante et le pointeur constant : l'un est un pointeur dont la valeur peut être changée mais que l'on ne peut utiliser pour modifier la valeur pointée ; l'autre, à l'inverse ne peut pas recevoir de nouvelle adresse mais il peut servir pour modifier la valeur pointée. Le pointeur constant sur constante cumule les deux : le pointeur ne peut pas changer de valeur et la valeur pointée ne peut pas être modifiée à travers le pointeur.
L'ajout de l'attribut `const` devant une déclaration de variable globale lui donne aussi un *linkage* interne ; elle ne peut donc plus être utilisée que dans l'unité de compilation où sa définition apparaît.