Polytech'Paris Sud - S2 « Programmation Impérative » - 2018/2019

NOTES DE COURS - Partie II

Frédéric VOISIN
(frederic.voisin@u-psud.fr)

Programme du cours :

[…]
Modèle simple de la mémoire et de l'exécution d'un programme
Environnement et état mémoire – Portée et durée de vie
Gestion des appels de fonctions – Passage de paramètres par valeur et par référence
Retour sur les fonctions récursives
Pointeurs et références en C++, allocation dynamique de mémoire
Tableaux « simples » en C et C++ et correspondance avec les pointeurs
Passage d'arguments au lancement d'un programme.

Résumé du chapitre :

Donner un « modèle » de l'exécution d'un programme (par exemple : déroulement de l'évaluation des expressions, d'une instruction, d'un appel de fonction),

Aborder quelques problèmes liés à l'association entre les identificateurs de variables utilisés dans un programme et la « mémoire », distinguer entre les aspects *statiques* (tels que la « portée des identificateurs ») et les aspects *dynamiques* (tels que la « durée de vie » des objets).

Introduction à la récursivité des fonctions, aux pointeurs et à l'allocation dynamique de mémoire. Comprendre la différence entre références et pointeurs.

En guise d'introduction, regardons comment un compilateur (ici `g++`*) organise sa mémoire, c'est-à-dire comment il associe une adresse en mémoire à chacun des éléments définis dans le programme. Nous considérons d'abord une situation réelle avant de voir un modèle simplifié dans lequel il sera plus facile d'étudier des mécanismes tels que les appels de fonction, les fonctions récursives ou encore les pointeurs et l'allocation dynamique de mémoire.*

Nous considérons un programme restreint à sa fonction `main`, sans variable globale. Le fragment de programme ci-après sera complété ultérieurement. Ce programme déclare des identificateurs de différents types simples (`char`, `int`, `double`), un tableau prédéfini de `int` et deux `vector<int>`. Le programme utilise des constructions de C++ qui n'ont pas (encore) été vues : usage non classique du symbole `&`[1], notation `(void *)`, etc. Nul besoin de comprendre ces détails qui ne servent ici qu'à imprimer les adresses des objets lors d'une exécution particulière (ce que vous n'avez a priori aucune raison de vouloir faire) :

1 La notation `&s` correspond à accéder à l'adresse de l'élément `s`.

```
int main() {
  int v = 36;
  int e[10] = { 12, 1, 12385, 0, 10, 307, 327, 100, 101, 99};
  vector<int> r;              // initialise à un vecteur vide.
  vector<int> s;              // idem.
  double d = 3.14;
  char c = 'C';

  cout << "sizeof(int): "      << sizeof(int) << endl;
  cout << "sizeof(double): "   << sizeof(double) << endl;
  cout << "sizeof(char): "     << sizeof(char) << endl;
  cout << "sizeof(r): "        << sizeof(r) << endl;
  cout << "sizeof(s): "        << sizeof(r) << endl << endl;
  cout << "adresse de v:     " << &v << endl;
  cout << "adresse de e[0]:  " << &(e[0]) << endl;
  cout << "adresse de e[9]:  " << &(e[9]) << endl;
      // 10 n'est pas un indice correct ! On s'en sert ci-dessous pour
      // connaître l'adresse de la case memoire stuée juste après le tableau e.
  cout << "adresse de e[10]: " << &(e[10]) << endl;
  cout << "adresse de r:     " << &r << endl;
  cout << "adresse de s:     " << &s << endl;
  cout << "adresse de d:     " << &d << endl;
  cout << "adresse de c:     " << (void *) &c << endl << endl;
  // a suivre
```

Le programme est compilé et exécuté sur un processeur et un système d'exploitation 64 bits. Nous obtenons les affichages ci-dessous (les tailles sont exprimées ici en octets) :

```
sizeof(int): 4
sizeof(double): 8
sizeof(char): 1
sizeof(r): 24
sizeof(s): 24

adresse de v:     0x7fff72eaf050
adresse de e[0]:  0x7fff72eaf0a0
adresse de e[9]:  0x7fff72eaf0c4
adresse de e[10]: 0x7fff72eaf0c8
adresse de r:     0x7fff72eaf060
adresse de s:     0x7fff72eaf080
adresse de d:     0x7fff72eaf058
adresse de c:     0x7fff72eaf04f
```

Les adresses sont exprimées en hexadécimal. Pour la discussion qui suit, seuls les 3 derniers symboles comptent puisque ces adresses ont toutes le même début. Les premiers affichages ne font que rappeler la taille utilisée pour chacune des sortes d'éléments. Dans les affichages qui suivent, les valeurs précises des adresses peuvent changer d'une exécution à la suivante, mais pas l'ordre relatif des objets tel qu'il a été mis en place à la compilation pour ce programme (changer de compilateur, ou d'options de compilation, pourrait changer un peu l'organisation des variables).

Nous illustrons l'implantation des objets en machine dans le tableau de la page 4. Le tableau n'est volontairement pas à l'échelle (il faudrait un tableau de 120 adresses) mais il fait apparaître explicitement les adresses de début et de fin de la zone associée au stockage de chaque objet défini dans le programme. Il fait aussi explicitement apparaître des zones laissées volontairement

inoccupées pour des raisons dites « d'alignement » : par exemple un `double` ne peut être alloué qu'à partir d'une adresse qui est un multiple de 8 ; les quatre octets d'un `int` ne peuvent démarrer qu'à une adresse qui est un multiple de 4. Si l'ordre d'association de mémoire aux objets ne tombe pas spontanément sur le multiple voulu, on est obligé de laisser libre la mémoire jusqu'à la prochaine adresse qui est le multiple de 8 (ou de 4) voulu. De même, la zone allouée à `s` ne suit pas directement la zone allouée à `r` pour des raisons d'efficacité propres au compilateur ou à l'architecture du processeur. Pour minimiser le gaspillage, le compilateur réorganise l'ordre naturel de stockage des éléments pour glisser des objets plus petits dans les trous qu'on aurait sinon. Dans notre exemple, la variable `v` (de type `int`) se retrouve entre la variable `c` (de type `char`) et la variable `d` (de type `double`). Ces détails peuvent être ignorés pour ce qui nous concerne dans la suite. Ils ne sont précisés ici que pour expliquer pourquoi certaines zones restent inoccupées, sans raison apparente, ou pourquoi les objets ne sont pas forcément placés en mémoire dans l'ordre de leur déclaration.

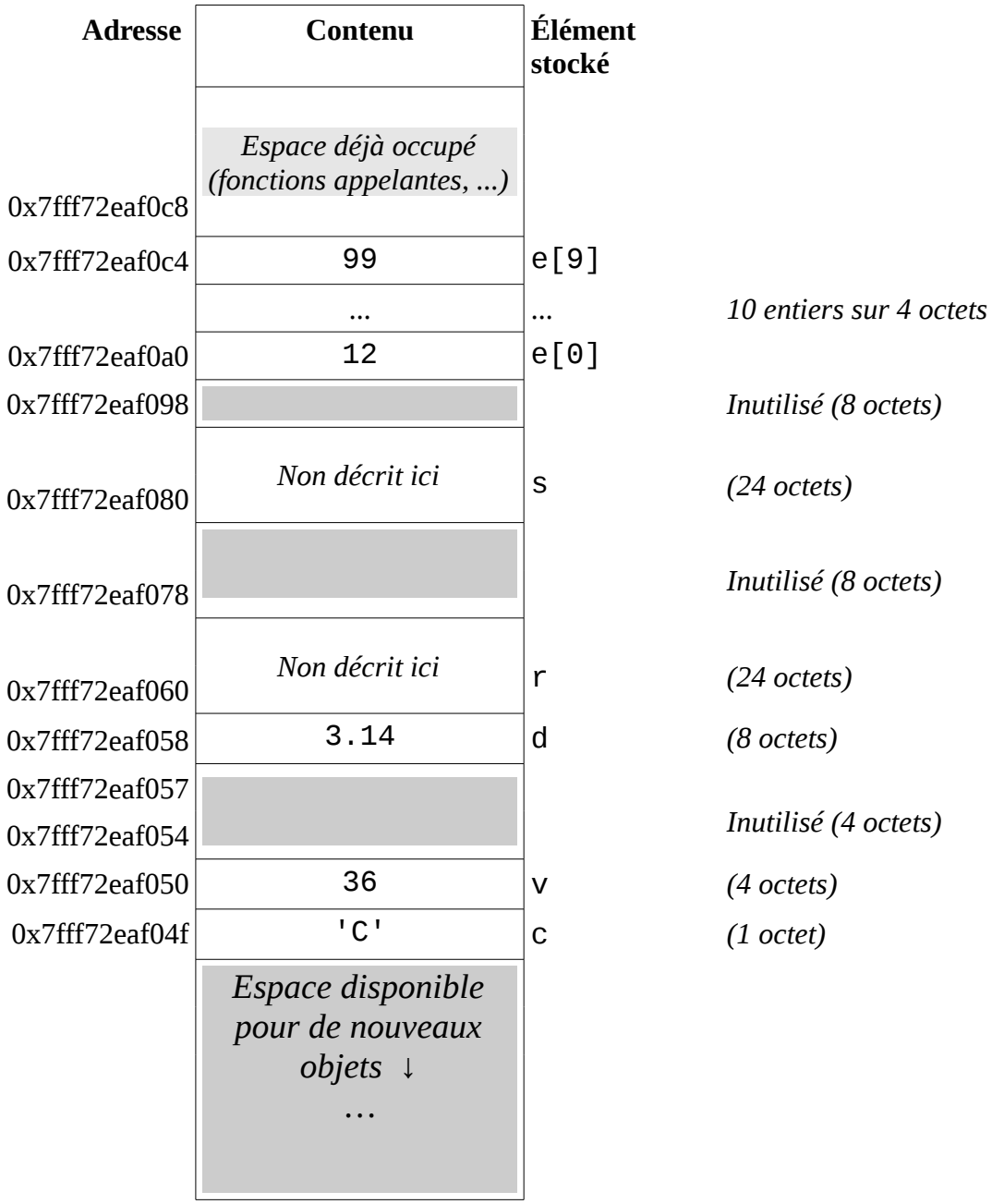
Remarquons que les deux instances de `vector` ont une taille fixe (ici 24 octets) alors que ceux-ci vont pouvoir contenir un nombre variable de valeurs au cours de l'exécution du programme. Les deux vecteurs sont pour l'instant vides, mais ils occupent une certaine place. Les éléments contenus dans le vecteur devront être stockés « ailleurs ». Les 24 octets contiennent l'adresse réelle de stockage des éléments et d'autres informations dont les implémenteurs des vecteurs ont besoin. Nous pouvons nous servir de ces `vector` sans connaître le détail de leur implémentation.

La représentation en mémoire du tableau `e` (déclaré par `int e[10]`) est très différente : nous trouvons directement ici l'adresse des différentes valeurs stockées. On pourrait construire une variante du programme dans laquelle il y aurait d'autres éléments stockés après ce tableau (voire un autre tableau) : on comprend donc facilement qu'on ne peut pas faire varier la taille d'un tableau prédéfini à l'exécution car il faudrait faire bouger en même temps tous les objets stockés après lui.

Dans le dessin de la page suivante, la partie haute du tableau est occupée par des informations liées aux fonctions qui ont appelée la fonction courante, par exemple ce qu'on appelle « l'adresse de retour » qui correspond à l'instruction qu'on devra exécuter quand on aura fini d'exécuter la fonction courante. Dans le cas de notre fonction `main`, qui démarre l'exécution d'un programme C++, il s'agit de la fonction du système d'exploitation qui a lancé le programme.

La partie basse du tableau (en gris clair) est la partie disponible dans laquelle on pourra stocker de nouveaux objets.

(voir tableau page suivante)



Placement en mémoire des variables du programme de la page 2.

Nous complétons notre programme illustratif pour nous intéresser maintenant à la façon dont sont stockés les éléments des vecteurs et aux opérations sur les vecteurs qui vont nous faire prendre conscience de deux manières différentes de gérer la mémoire : une partie sera gérée de façon disciplinée (gestion en « pile »), l'autre de façon beaucoup moins prévisible mais aussi plus souple (gestion en « tas »). On distingue dans la suite trois « instants » particuliers durant l'exécution du prgramme : après l'initialisation, après le remplissage, et après l'affectation entre vecteurs :

```
    // suite et fin du programme
    // A l'initialisation les vecteurs s et t sont vides.
    cout << "Adresse des vecteurs et des zone de stockage des elements\n";
    cout << "adresse de r:     " << &r << endl;           // instant 1
    cout << "adresse de r[0]:  " << &(r[0]) << endl;
    cout << "adresse de s:     " << &s << endl;
    cout << "adresse de s[0]:  " << &(s[0]) << endl ;
    cout << endl;

    cout << "On remplit les deux vecteurs\n";
    for(int i = 0; i < 50; i = i+1) { r.push_back(42); } // r contient donc 50 éléments
    s.push_back(42);                                     // s contient un seul élément
    cout << "sizeof(r):        " << sizeof(r) << endl;   // instant 2
    cout << "adresse de r:     " << &r << endl;
    cout << "adresse de r[0]:  " << &(r[0]) << endl;
    cout << "sizeof(s):        " << sizeof(s) << endl;
    cout << "adresse de s:     " << &s << endl;
    cout << "adresse de s[0]:  " << &(s[0]) << endl << endl;

    cout << "Affectation de r à s\n";                    // le plus grand dans le plus petit !
    s = r;    // affectation entre vecteurs              // instant 3
    cout << "sizeof(s):        " << sizeof(s) << endl;
    cout << "adresse de s[0]:  " << &(s[0]) << endl;
    cout << "adresse de s[50]: " << &(s[50]) << endl;    // juste après le dernier élément de s
    return 0;
}
```

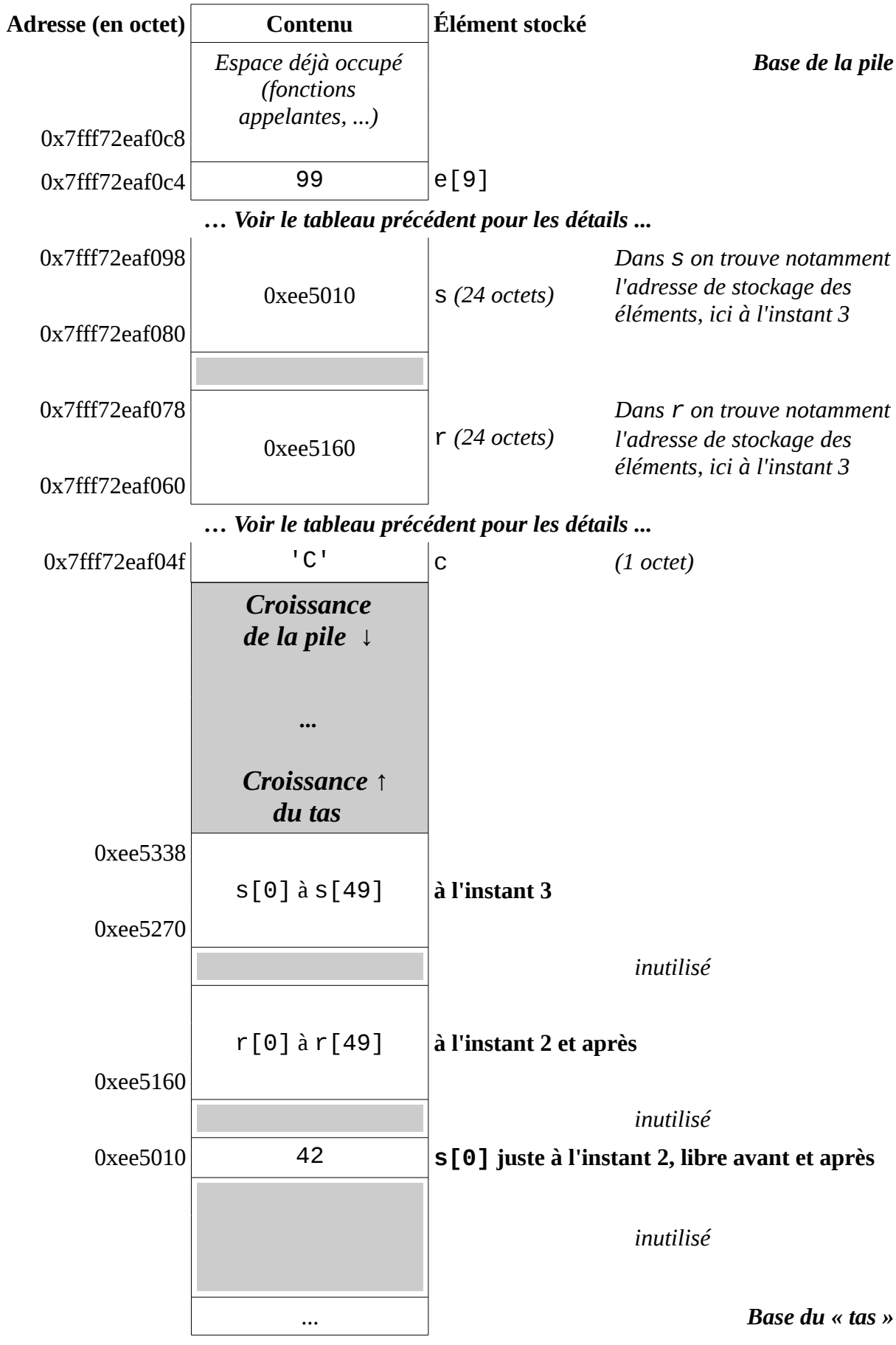
À l'exécution, nous obtenons les affichages ci-dessous :

```
Adresse des vecteurs et des zone de stockage des elements  // instant 1
adresse de r:     0x7fff72eaf060
adresse de r[0]:  0
adresse de s:     0x7fff72eaf080
adresse de s[0]:  0

On remplit les deux vecteurs                               // instant 2
sizeof(r):        24
adresse de r:     0x7fff72eaf060
adresse de r[0]:  0xee5160
sizeof(s):        24
adresse de s:     0x7fff72eaf080
adresse de s[0]:  0xee5010

Affectation de r à s                                       // instant 3
sizeof(s):        24
adresse de s[0]:  0xee5270
adresse de s[50]: 0xee5338
```

Le tableau de la page suivante donne une vue abrégée du tableau de la page 4, en se focalisant maintenant sur les zones mémoire associées aux vecteurs .


Adresse (en octet)
Contenu
Élément stocké
Espace déjà occupé (fonctions appelantes, ...)
Base de la pile
0x7fff72eaf0c8
0x7fff72eaf0c4
99
e[9]
… Voir le tableau précédent pour les détails ...
0x7fff72eaf098
0x7fff72eaf080
0xee5010
s (24 octets)
Dans s on trouve notamment l'adresse de stockage des éléments, ici à l'instant 3
0x7fff72eaf078
0x7fff72eaf060
0xee5160
r (24 octets)
Dans r on trouve notamment l'adresse de stockage des éléments, ici à l'instant 3
… Voir le tableau précédent pour les détails ...
0x7fff72eaf04f
'C'
c
(1 octet)
Croissance de la pile ↓
...
Croissance ↑ du tas
0xee5338
0xee5270
s[0] à s[49]
à l'instant 3
inutilisé
r[0] à r[49]
à l'instant 2 et après
0xee5160
inutilisé
0xee5010
42
s[0] juste à l'instant 2, libre avant et après
inutilisé
...
Base du « tas »

On remarque que pour le stockage des éléments d'un vecteur (`adresse de r[0]: 0xee5160` ou `adresse de s[50]: 0xee5338` par exemple) on utilise des valeurs d'adresses très différentes de celles utilisées pour les variables du programme : elles concernent une autre zone de l'espace mémoire alloué pour l'exécution de notre programme. Sur la page précédente, nous dessinons cette zone dans la partie basse du tableau, pour bien la distinguer de la partie haute du tableau, utilisée jusqu'à présent. Le contenu des vecteurs allant être modifié au fur et à mesure de l'exécution du programme, les affichages ont sur leur droite les références aux paragraphes qui suivent :

1. à l'instant **1**, les deux vecteurs sont initialement vides. La valeur `0` qu'on trouve associée aux adresses de `r[0]` et `s[0]` est une valeur conventionnelle pour dire « pas d'espace alloué ».
2. à l'instant **2**, comme indiqué précédemment, ni la taille des vecteurs (`24`) ni leur adresse n'ont changé suite à leur remplissage. Par contre les adresses des premiers éléments ont changé (`0xee5160` pour l'adresse de `r[0]`) : de l'espace a été alloué au cours de l'exécution pour pouvoir stocker les éléments du vecteur.
3. à l'instant **3**, on réalise une affectation de `r`, vecteur de 50 éléments, à `s` qui pour l'instant contient un unique élément. À nouveau, rien ne change pour la taille et l'adresse de `r` et `s`, ce qui change c'est l'adresse de la zone qui contient les éléments de `s` (l'adresse de `s[0]` passe de `0xee5010` à `0xee5270`). En séparant la zone mémoire où se trouve le descriptif de `s` de la zone mémoire qu'on utilise pour stocker réellement les valeurs contenues dans `s` on rend possible un changement de taille dynamique du nombre d'éléments contenus dans un vecteur sans modifier l'adresse des variables `r` et `s` à proprement parler.

Quel est le prix à payer pour la facilité d'utilisation donnée par le « tas »?

1. À force d'allocations et de désallocation, la zone gérée en « tas » peut ressembler à du gruyère ! Ceci est illustré dans la partie « tas » par les différentes zones en grisé.
2. La place mémoire utilisée pour un vecteur est plus grande que la place mémoire nécessaire pour stocker ses éléments (surcoût ici de 24 octets) ; par exemple le tableau prédéfini `e` n'occupe que les 40 octets dans la pile pour ses éléments, tandis qu'un `vector<int>` avec 10 `int` demandera 24 octets de plus : 24 dans la pile et 40 dans le tas.
3. Surtout il faut gérer beaucoup plus finement cette partie de la mémoire. Le nombre des variables de la fonction `main`, et leur type, ne va pas bouger lors de l'exécution du programme. On sait dès le départ le nombre de cases mémoires dont on a besoin et lesquelles sont utilisées : on les alloue au moment où on démarre `main` et on les libére au moment où on quitte `main`. Quand on considérera les appels de fonctions, il en ira de même : quand on entre dans une fonction appelée il faut faire de la place pour ses paramètres et ses objets locaux, quand on la quitte, on libère la place. On gère cette zone mémoire comme une **pile** : « dernier entré, premier sorti » ! Cette partie de la mémoire est gérée automatiquement par le compilateur puisqu'il en connaît le mode d'emploi : à chaque fois il connaît la taille de la zone dont il a besoin et comment est organisé le placement des objets dans cette zone. Pour la zone utilisée pour les éléments de vecteur, on ne peut pas prédire à l'avance ce qui va être utilisé puisqu'il faudrait pour cela exécuter le programme.

   La gestion de cette zone demande de :

   - savoir distinguer la partie déjà utilisée et les zones qui sont encore libres.
   - signaler explicitement au système qu'une zone mémoire n'est plus utilisée. Lors de l'affectation de `r` à `s`, au moment d'allouer une nouvelle zone mémoire pour les

éléments de `s`, il convient de signaler que l'ancienne zone mémoire est de nouveau libre. Sinon rien ne permet au système de savoir que cette zone n'est plus utilisée[2]. Pour les vecteurs, ceci est géré dans l'implémentation de l'opérateur d'affectation, ainsi que dans les autres opérations qui ont un impact sur cette zone mémoire : `clear`, `push_back`, `pop_back`, etc. On parle de « gestion dynamique » de la mémoire puisque celle-ci ne peut pas être anticipée par le compilateur et dépend de l'exécution précise du programme. On retrouvera cette notion avec l'allocation dynamique de mémoire en liaison avec les pointeurs.

Gérer les allocations et désallocations de mémoire sur le tas va coûter un peu de temps à l'exécution puisqu'il faut mettre à jour l'information sur les zones libres. Gérer la pile est plus simple et plus rapide puisqu'on alloue et désalloue de façon disciplinée : il suffit de mémoriser l'adresse du sommet actuel de la pile. Quand la gestion en pile de la mémoire sera possible, elle sera privilégiée ; mais l'exemple des vecteurs montre que ce n'est pas toujours possible.

Signalons que nous ne faisons pas apparaître la frontière entre la zone gérée en pile et la zone gérée en tas. Dans notre tableau la pile va s'étendre vers le bas tandis que le « tas » aura tendance à s'étendre vers le haut. Sur une architecture 64 bits (donc 2^{64} adresses différentes : calculez le nombre d'octets associés !) le problème qu'ils se croisent ne se pose pas en pratique ; sur une architecture 32 bits on pourrait avoir des problèmes de « débordement de pile » ou de « épuisement du tas ».

2 Dans d'autres langages, cette récupération de la mémoire non utilisée est réalisée automatiquement par un outil appelé « ramasse-miettes » (le terme anglo-saxon est moins élégant : « garbage collector »).

Modèle Mémoire, Environnement, Portée et Durée de vie

1. Modèle de la mémoire associée à un programme :

Dans notre modèle, la mémoire est vue comme un ensemble « d'emplacements » capables chacun de stocker des « valeurs ». La valeur stockée dans un même emplacement pourra changer au cours du temps (on parlera de « l'état de la mémoire » à un instant donné). Si on ne cherche pas à différencier entre différents types de « valeurs de base » possibles (les différentes classes d'entiers, les flottants, etc.), **on peut voir la « mémoire » comme une fonction partielle de l'ensemble (fini) des emplacements vers l'ensemble des valeurs.** La fonction est « partielle » au sens où la valeur associée à un emplacement n'est pas toujours « définie » (ce n'est pas le programme qui a fixé la valeur en cet emplacement) même si, en pratique, accéder au contenu de cet emplacement fournira une valeur arbitraire. On suppose dans notre modèle qu'un emplacement est capable de contenir une valeur de n'importe quel type scalaire (entier, flottant, pointeur...). La réalité est bien sûr toute autre comme on l'a vu dans le premier chapitre : toutes les valeurs des types de base n'occupent pas le même espace. Avoir un modèle plus précis serait bien sûr possible mais compliquerait inutilement la tâche pour ce qui nous intéresse dans la suite. Pour une valeur d'un type structuré (tableau, vecteur ou `struct`), il faudra par contre réserver autant d'emplacements que d'éléments scalaires et avoir un moyen d'accéder individuellement aux différents éléments. De même, une affectation à une `struct` devra être décomposée en autant d'affectations élémentaires que nécessaire.

Une autre simplification dans notre modèle sera de ne refléter dans la mémoire que les valeurs des variables du programme sans considérer celles gérées automatiquement par le système pour effectuer les calculs intermédiaires nécessités par l'évaluation d'une expression complexe. Par exemple une expression telle que `(x+3)*(y+7)` oblige à mémoriser le résultat de `x+3` le temps de calculer `y+7` et de faire le produit, bien qu'aucune variable n'a été explicitement déclarée pour mémoriser cette valeur intermédiaire. En pratique, il faut garder d'autres informations : référence sur la prochaine l'instruction à exécuter, sauvegarde de divers « registres » de calcul, valeur de retour d'une fonction, etc., dont la description dépasse très largement le cadre de ce module.

À ce modèle nous ajoutons une valeur spéciale $\perp$ (« bottom ») pour représenter le fait qu'un emplacement a une valeur qui n'a pas été définie explicitement par le programme. Au début de l'exécution d'un programme, la valeur de tout emplacement est « indéfinie » et vaut donc $\perp$ dans le modèle. Ce symbole $\perp$ n'est qu'un élément du modèle, une telle valeur n'existe pas dans la réalité.

Le modèle correspond finalement à une table (finie) dont les cases contiennent une valeur d'un type de base ou $\perp$. Pour simplifier, nous supposons que ces emplacements sont indexés par des entiers (les « adresses ») gérés de façon consécutive à partir de 0. L'aspect « fini » de la table reflète le fait qu'un certain espace mémoire a été alloué pour l'exécution du programme et que le système d'exploitation pourrait détecter que certaines adresses sont invalides[3]. Par contre, comme indiqué précédemment, il ne peut pas détecter que la valeur associée à une case valide est indéfinie ($\perp$).

Dans un premier temps, la mémoire sera gérée comme une **pile** : les adresses sont allouées à partir de 0 jusqu'à une certaine valeur mais la hauteur de la pile pourra varier au cours de l'exécution du programme. Par exemple quand on effectue un appel de fonction, le système va allouer de l'espace

3 On ne connaît pas la granularité avec laquelle le système alloue de la mémoire au programme : probablement pas emplacement par emplacement. Il pourra donc y avoir des emplacements que le système considère comme valides, même s'ils ne sont associés à aucune variable de notre programme. Dans le modèle nous représenterons ceci graphiquement par une table qui peut être plus grande que celle qui serait strictement nécessaire.

supplémentaire pour stocker les paramètres et variables locales de la fonction. On libérera logiquement cet espace quand la fonction aura terminée son exécution (on libère la même quantité que celle allouée à l'appel). Dire que la mémoire est gérée **en pile** veut dire que quand on alloue de la mémoire, cela se fait au sommet de la pile courante. Quand on la libère, cela se fait toujours à partir du sommet de la pile. Il n'existe pas de « trou » (emplacements logiquement non valides pour le programme à un instant d'exécution) au milieu de la pile : si une adresse est valide, toutes celles en dessous dans la pile le sont aussi.

Nous avons vu dans l'exemple introductif que la mémoire pour l'exécution d'un programme C++ peut être vue comme séparée en deux parties : l'une gérée en pile, l'autre gérée de façon non prévisible dans le temps (en liaison avec l'allocation dynamique de mémoire par le programmeur et les pointeurs). **Dans ce chapitre nous nous préoccupons uniquement de la pile.**

Nous ajoutons un élément au modèle : une marque (notée ● dans la suite) pour refléter les adresses qui sont logiquement valides à un instant de l'exécution du programme. Comme il s'agit d'une pile, les emplacements seront toujours « marqués » de façon consécutive à partir de 0, mais cela va nous permettre de visualiser des emplacements qui ont pu être valides (« marqués ») à un instant, mais qui ne le seront plus à un autre instant (même s'ils ont conservé la valeur qu'on y a mise).

Pour illustrer ce modèle, considérons l'exemple suivant d'un état mémoire $\mathbb{M}$ à un instant de l'exécution du programme. Pour faciliter la lecture, nous indiquons explicitement les adresses (des entiers) à gauche des emplacements :

0	●	`3`
1	●	`'a'`
2	●	⊥
3	●	⊥
4	●	`3.14`
5	●	`2`
6		`-3`
7		`7`
…		…

Les emplacements d'adresses 2 et 3 n'ont pas de valeur définie et correspondent à des emplacements associés à des variables non initialisées. Le contenu de l'emplacement 4 est ici interprété comme représentant un flottant (supposons qu'il est associé à une variable déclarée avec un tel type), celui de l'emplacement 1 est interprété comme un `char`. Enfin, seules les adresses comprises entre 0 et 5 sont actuellement valides, les adresses 6 et 7 ont pu l'être à un moment donné (ce qui a permis de définir les valeurs dans ces emplacements) mais ne le sont plus. Du point de vue de la logique de l'exécution, le sommet de pile est à l'adresse 5.

Ce modèle « avec marques » semble inutilement compliqué par rapport au modèle intuitif qui représente la mémoire comme un tableau de mots auxquels on accède via une « adresse », c'est-à-dire un entier naturel. Il nous permet par la suite de mieux illustrer certains aspects.

Dans notre modèle d'exécution d'un programme, il faut aussi expliciter l'association des identificateurs du programme aux emplacements. On se donne une fonction (« **l'environnement** ») qui associe à tout identificateur déclaré dans le programme un emplacement en mémoire. Dans l'exemple précédent on pourrait imaginer que le programme comportait les déclarations suivantes:

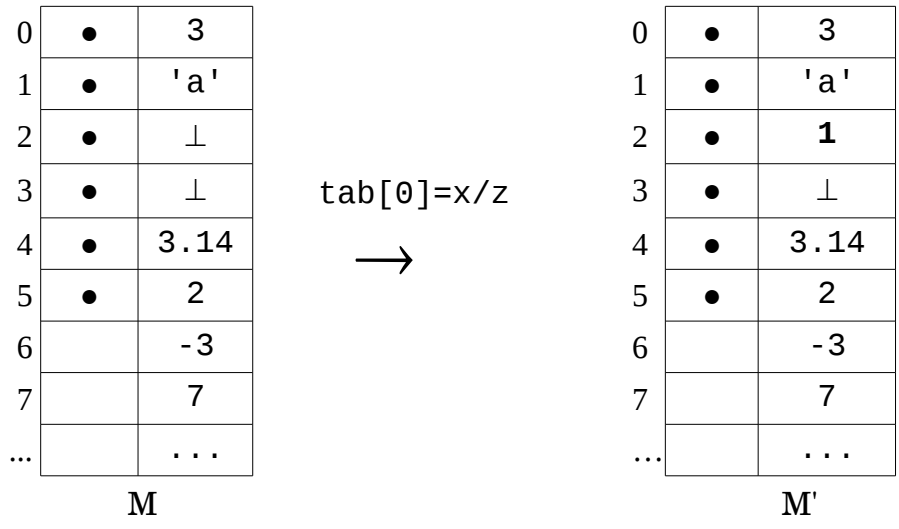
```
int x = 3; char c = 'a'; int tab[2]; double d = 3.14; int z = 2;
```

et que le compilateur a associé `x` à l'emplacement 0, `c` à 1, `z` à 5, `d` à 4, `tab[0]`et `tab[1]` aux emplacements 2 et 3 (de valeurs indéfinies puisque `tab` n'est pas initialisé).

Évaluer une expression « pure » (« sans effet de bord ») consiste à accéder aux valeurs contenues dans les emplacements associées aux variables concernées et à appliquer (la traduction dans notre modèle de) l'opérateur concerné du programme. Par exemple, dans l'état mémoire M, l'évaluation de l'expression `x/z` donne la valeur `1`, puisque le compilateur (via les déclarations de `x` et `z`) a mémorisé que `x` et `z` sont des variables de type `int`, et donc que `'/'` dénote la division entière appliquée à M[x] et M[z] ; de même pour l'évaluation de `x%z` puisque dans le langage l'opérateur `'%'` représente le modulo mathématique. L'évaluation de `x/tab[0]`donne une valeur que nous considérons comme indéfinie (⊥) puisque `tab[0]` vaut ⊥, même si dans la réalité le programme donnerait une valeur dépendante du contenu effectif de l'emplacement 2 à cet instant.

Dans ce modèle, accéder dynamiquement à un **identificateur non déclaré** serait impossible à réaliser puisqu'il ne lui serait pas associé d'emplacement: à l'exécution on ne saurait pas quoi faire. Le compilateur doit donc rejeter auparavant de tels programmes, ce qu'il est possible de faire statiquement [4]. Le modèle permet de distinguer deux types d'erreurs de programmation : accéder à un emplacement défini mais de valeur indéfinie (⊥), et accéder à un emplacement indéfini (non « marqué » par ●). Utiliser une variable non déclarée est un troisième type d'erreur mais comme elle est détectée dès la compilation, on ne s'en préoccupe plus à l'exécution ! Accéder à un élément de tableau en dehors des bornes d'un tableau n'est pas contrôlé en C/C++ et n'est pas défini.

Exécuter une affectation consiste à **modifier l'état mémoire** en définissant la nouvelle valeur de l'emplacement associé à cette variable ou élément de tableau. Partant de l'état mémoire M on arrive à l'état mémoire M' en exécutant l'instruction `tab[0]=x/z`.



4 Cela n'a pas de sens d'accéder à un élément de tableau via un indice incorrect : existe-t-il un emplacement associé ? Si c'est le cas quelle est la valeur obtenue ? Mais le compilateur ne peut pas le vérifier si l'indice est une expression dont la valeur n'est pas connue statiquement.

On peut voir l'affectation comme une **fonction entre états mémoire** (en étendant la définition de M de manière à la définir non seulement sur les **variables** déclarées dans le programme mais sur toutes les **expressions** construites sur les variables du programme et opérateurs du langage) :

$$\texttt{x = exp}$$

$$\mathbf{M} \quad \longrightarrow \quad \mathbf{M'}$$

où $\mathbf{M'}$ est définie sur les variables du programme par[5]

$\mathbf{M'}(\texttt{id}) = \mathbf{M}(\texttt{id})$ si $\texttt{id} \neq \texttt{x}$

$\mathbf{M'}(\texttt{id}) = \mathbf{M}(\texttt{exp})$ sinon,

et on définit $\mathbf{M}(\texttt{exp})$ par induction sur la structure de `exp`.

Par exemple $\mathbf{M}(\texttt{x+z/3}) = \mathbf{M}(\texttt{x}) + \mathbf{M}(\texttt{z/3}) = \mathbf{M}(\texttt{x}) + (\mathbf{M}(\texttt{z}) / \mathbf{M}(\texttt{3})) = \mathbf{M}(\texttt{x}) + (\mathbf{M}(\texttt{z}) / \texttt{3})$, la valeur d'une constante étant bien sûr indépendante de l'état mémoire.

Une expression (pure) accède à l'état mémoire, sans le modifier, et retourne une valeur; une affectation modifie cet état mémoire. Aucune des deux constructions ne modifie la manière dont on associe un identificateur du programme à un emplacement mémoire. Ce dernier point est géré par les **déclarations** du langage et la manière dont ces déclarations sont traduites dans le modèle.

2. Notion « d'environnement »

Dans la suite, pour simplifier, on ne considère que les identificateurs de variables (un identificateur peut aussi désigner un type, une fonction, etc).

Pour résoudre ce problème d'**association entre identificateurs et emplacements**, on peut le décomposer en deux sous-problèmes

- Connaître la liste des identificateurs visibles en un point du programme (**statique**)
- Garantir qu'en chaque point du programme on associe un emplacement à tous les identificateurs visibles en ce point, notamment lors des appels de fonction (**dynamique**).

Environnement: c'est la fonction qui associe un emplacement (ou « un ensemble d'emplacements » pour les types composés) à tout identificateur de variable du programme. Le passage de l'identificateur à sa valeur peut être décomposé en deux étapes très différentes :

Les déclarations influencent l'environnement, les instructions influencent l'état mémoire. Les appels de fonctions sont plus compliqués à traiter (ils ajoutent des variables à l'environnement) et sont vus dans un second temps.

5 Cette définition n'est correcte que si les expressions sont sans effet de bord, sinon il faut l'adapter comme on peut s'en rendre compte si on considère l'instruction `z=x++;` L'état mémoire est modifié pour `x` et pour `z`.

La gestion de l'environnement peut se voir de deux façons complémentaires:

– Quels sont les identificateurs visibles en un point donné du programme ?

– À quels fragments du programme s'applique telle déclaration d'identificateur ?

On appelle **contrôle de la portée** des identificateurs ce dernier problème. **Définir la portée d'un identificateur** c'est définir les zones du programme où cette déclaration est visible, compte-tenu des éventuelles redéclarations de l'identificateur et des règles de masquage définies par le langage.

Catégories d'identificateurs: on peut ranger les identificateurs de variables dans plusieurs catégories auxquelles seront associées des portées et des règles de gestion différentes :

– les **variables « globales »**, déclarées en dehors des fonctions. Leur portée est la portion de fichier qui suit leur déclaration (moins les endroits où l'identificateur est redéclaré). Il existe des qualificatifs supplémentaires (tels que `extern` ou `static`) que nous verrons si besoin.

– Les **paramètres formels** d'une fonction jouent le rôle d'interface avec les fonctions appelantes. Techniquement on peut les voir comme des variables locales à la fonction qui reçoivent leur valeur initiale de l'argument (ou « paramètre effectif ») correspondant à l'appel. Leur portée est limitée à la fonction qui les déclare, à l'exception des blocs où ils sont masqués.

– Les **variables locales à une fonction** : elles sont traitées comme les paramètres formels sauf qu'elles ne reçoivent pas de valeur initiale via le passage de paramètre. Elles ont les mêmes règles de portée : le corps de la fonction moins les blocs où elles sont masquées.

– Les **variables locales à un bloc** : groupe d'instructions éventuellement précédées de déclarations compris entre les symboles `{` et `}`. Leur portée est limitée au bloc qui les déclare, à l'exception des blocs internes dans lesquels elles sont masquées.

On a donc des règles de portée associées à des blocs emboîtés : une déclaration dans un bloc masque une déclaration homonyme d'un bloc englobant, et n'est valable que jusqu'à la fin de ce bloc interne. Cette déclaration peut elle-même être localement masquée dans un bloc plus interne. À l'intérieur d'un bloc, les déclarations sont organisées « linéairement » (c.à.d. l'ordre importe) et on ne doit pas avoir une redéclaration du même identificateur dans le même bloc. En C++, les paramètres formels et les variables locales du bloc principal d'une fonction forment un seul bloc : une variable locale du bloc principal de la fonction ne doit pas masquer un paramètre formel, de même que deux paramètres d'une fonction ne peuvent pas avoir le même nom ! Les variables globales peuvent être vues comme appartenant à un bloc englobant toutes les fonctions.

En C++, des déclarations de variables peuvent aussi apparaître au milieu des instructions et ont une portée jusqu'à la fin du bloc. Ceci complique légèrement l'énoncé des règles de portée tel que nous les avons données. Pour simplifier, nous ignorons cette possibilité dans la suite.

Remarque importante: le fait qu'un identificateur soit masqué par une déclaration homonyme ne signifie pas que cet identificateur (et surtout l'objet qu'il représente) n'existe plus, simplement qu'on ne peut plus localement le **nommer**: il devient « invisible ». Mais il suffit de quitter la portée de la déclaration masquante pour que sa déclaration retrouve sa visibilité (voir l'exemple suivant, et notamment les variables dans la fonction `f`). Les **notions « être visible » et « exister » ne sont pas équivalentes** et ne doivent pas être confondues.

Nous illustrons ces différents points sur l'exemple de la page suivante.

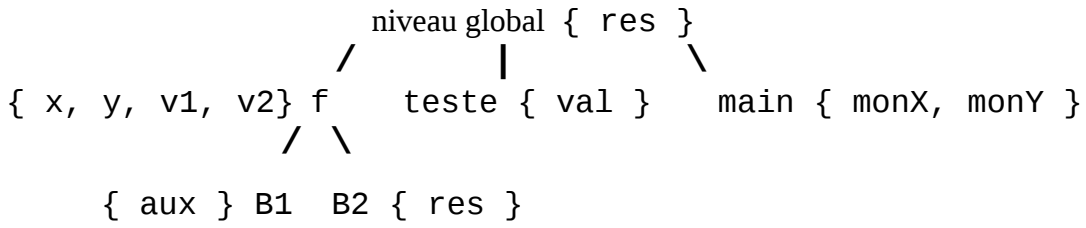
```
// res : variable globale. Sa portée est le reste du fichier sauf les portions de programme où elle
// est masquée par une redéfinition
int res = -1;

void teste(int val) {          // val : paramètre formel
  if (res > val) {             // ici, res désigne la variable globale
     printf("Cas 1\n");
  } else { printf("Cas 2\n");}
}

int f(int x, int y) {          // paramètres formels de f
  int v1, v2;                  // variables locales de f
  v1 = x + y;
  v2 = x * y;
  if (v1 > v2) {               // Bloc B1
     int aux = v1 * v2 + 2;    // aux : variable locale au bloc B1
     res = res + 3 * aux;      // ici, res est la variable globale
  }                            // fin du bloc B1 et de la portée de aux
  else {                       // Bloc B2
     int res = v1 – v2;        // res variable locale à B2, masque le res global
     teste(res);
  }                            // fin de B2 et de la portée du res local
  res = -12;                   // il s'agit à nouveau ici du res global
  return v1 + res;
}

int main() {
  int monX = 5, monY = 12;     // variables locales à main
  res = f(monX, 2*monY);       // ici res est la variable globale
  printf("Valeur finale de res: %d\n", res);
  return 0;
}
```

Les portées des déclarations peuvent être résumées par un « **arbre des portées** » dans lequel on indique les fonctions et les niveaux d'imbrication des blocs, et dans chaque bloc les déclarations locales de ce bloc. **La portée d'une déclaration** est le **sous-arbre** de racine le bloc qui effectue cette déclaration, **moins** les sous-arbres qui contiendraient une déclaration homonyme. Dans l'exemple, on obtient l'arbre ci-après, dans lequel on note sous forme ensembliste les variables à côté de l'endroit où elles ont été déclarées. Sont visibles dans `B2` les variables `x`, `y`, `v1`, `v2` (« héritées » de `f`) ainsi que la variable `res` du bloc `B2`, mais ni la variable globale `res`, ni la variable `aux` locale à `B1`. La variable globale `res` reste visible dans `B1` (ces situations sont à éviter car elles compliquent la compréhension du programme mais elles ne peuvent pas toujours être évitées dans de gros programmes notamment pour le masquage d'identificateurs globaux par des déclarations locales).



3. Gestion dynamique des emplacements: nous nous intéressons maintenant à l'association d'un emplacement à chaque identificateur. La gestion de l'environnement et des emplacements devra garantir qu'on associe une place aux identificateurs visibles selon les règles de portée, qu'on les libère de façon adéquate et qu'on retrouve les associations précédentes une fois qu'un bloc est quitté. Les portées étant gérées en bloc, l'environnement le sera aussi, en prenant cependant soin de pouvoir retrouver en sortie de bloc l'environnement précédent.

Appels de fonctions: nous rappelons que le passage de paramètres en C++ se fait soit « **par valeur »,** soit **« par référence »**. Dans le premier cas, on **copie** la valeur de l'argument dans le paramètre formel associé. Celui-ci peut donc être vu comme une variable locale qui serait initialisée automatiquement à la valeur de l'argument lors de l'appel de fonctions (« passage de paramètres ») : le paramètre formel et l'argument représentent deux objets différents. Dans le second cas, le paramètre formel n'est qu'une référence sur l'argument correspondant : cela revient à dire que le paramètre formel est une sorte « d'alias » pour l'objet correspondant de la fonction appelante.
Il en est de même pour la valeur retournée par une fonction : on peut choisir de renvoyer soit une valeur directement (on copie la valeur renvoyée dans l'emplacement chez l'appelant), soit une référence sur un objet (ce qui peut être dangereux comme nous le verrons). Si la fonction appelée « oublie » de retourner explicitement une valeur, une copie d'une valeur arbitraire ($\perp$ dans notre modèle) a lieu[6]. Si la fonction appelante ne se sert pas de la valeur de retour, celle-ci est « oubliée » et en C++ ça n'engendre ni erreur, ni warning.

La description du mode de passage de paramètre se fait grâce à l'en-tête de la fonction. Le compilateur doit connaître l'en-tête de toute fonction avant son utilisation dans un appel, car le travail requis diffère selon le mode de passage de paramètres.

Exemple : Dans la fonction `divisionEuclidienne` ci-dessous les paramètres `a` et `b` sont passés « par valeur », `quotient` et `reste` sont passés « par référence » pour que leur modification à l'intérieur de la fonction se répercute sur les arguments correspondant chez l'appelant. Notez au passage l'usage de la fonction `printf` pour des affichages « formatés » :

```
#include <iostream>
#include <cstdio>          // requis pour l'usage de printf ci-dessous.

using namespace std;

// déclaration pour indiquer l'en-tête de la fonction
bool divisionEuclidienne(int a, int b, int &quotient, int &reste);
```

6 Le compilateur `gcc` émet un « warning » si une fonction dont le type de retour n'est pas `void` a un chemin dans son code qui permet d'atteindre la fin de son bloc sans passer par une expression `return` avec une valeur de retour.

```
void teste(int a, int b) {
  int q, r;
  if (divisionEuclidienne(a, b, q, r)) {
    printf("Division euclidienne de %d par %d = (%d, %d)\n",
           a, b, q, r);
  } else {
   printf("Division euclidienne de %d par %d impossible\n",
           a, b);
  }
}
```

// définition complète de la fonction

```
bool divisionEuclidienne(int a, int b, int &quotient, int &reste) {
  int v = a, q = 0;
  if (b == 0) return false;      // valeurs de quotient et reste indéfinies
  while (v >= b) { v -= b; q += 1; }
  quotient = q; reste = v;
  return true;
}
```

Dans cet exemple on se sert de la valeur de retour booléenne pour indiquer si le calcul a été possible ou non : si la fonction renvoie `false`, dans `teste` les variables `q` et `r` ont une valeur indéfinie. Deux remarques importantes :

- à l'intérieur de la fonction `divisionEuclidienne` on manipule `quotient` et `reste` comme s'il s'agissait de variable de type `int` alors qu'elles sont de type `int &`. C'est le compilateur qui fait les efforts d'ajustement nécessaires. Il en est de même au niveau de l'appel : dans l'appel `divisionEuclidienne(a, b, q, r)`dans `teste`, rien n'y distingue un passage par valeur d'un passage par référence. Il est donc important que le compilateur (et le lecteur du code) connaisse à l'avance l'en-tête de la fonction.
- L'argument associé à un paramètre formel doit avoir une notion d'emplacement mémoire (une variable, une case de tableau, un élément de vecteur, …) et non pas une valeur : on ne peut pas définir une référence sur une constante, comme on peut s'en convaincre facilement en considérant un appel (incorrect) tel que `divisionEuclidienne(15, 4, 0, 3*2)`. Au retour de cet appel on ne s'attend pas à ce que la constante `0` se soit transformée en `3`. Il n'y a pas d'emplacement mémoire associé à `0` ou au résultat de `3*2`; on ne peut pas prendre leur adresse et l'appel est donc incorrect et rejeté à la compilation.

Dans la suite, comme indiqué au tout début, nous choisissons de ne représenter que les emplacements associés aux identificateurs introduits par le programme. Parmi les zones supplémentaires qui seraient nécessaires de représenter pour être plus fidèle à la réalité, on peut citer :

- la zone pour stocker le résultat d'un appel de fonction
- la zone pour mémoriser la prochaine instruction à exécuter dans la fonction appelante quand la fonction appelée aura retourné son résultat
- des emplacements temporaires liés à la décomposition d'expressions « complexes » telles que `(x+5)*(y+7)` pour mémoriser les résultats de sous-expressions.

Modèle pour la gestion de l'environnement et des emplacements : [7]

Dans un premier temps nous ne décrivons que le passage par valeur. Le passage par référence sera évoqué dans une section ultérieure.

Nous représentons en parallèle l'environnement et la mémoire :

Environnement

identificateur	emplacement

Mémoire

	marque	valeur

Dans l'environnement, nous ajoutons un ● derrière un identificateur s'il représente la **liaison courante** de cet identificateur **compte-tenu des règles de portée**. Nous continuons à associer un ● à un emplacement mémoire si à cet instant de l'exécution l'utilisation de cet emplacement est valide (car associé à une variable du programme qui existe encore).

1. Début du programme: on associe un emplacement aux variables **globales**[8], et initialisation:

res ●	0

0	●	-1
1		⊥
...		⊥

2. Début de l'exécution de `main` : association d'emplacements aux **paramètres formels** (il n'y en a pas) et aux variables **locales**. On indique la situation **après** l'affectation aux variables `monX` et `monY` :

`res` ●	0
`monX` ●	1
`monY` ●	2

0	●	-1
1	●	**5**
2	●	**12**
...		⊥

3. Appel à la fonction `f`. À la fin de son exécution il restera à copier son résultat dans `res`, variable globale du programme, c'est-à-dire à l'emplacement 0. L'environnement ci-après est celui qui existe **au début** du corps de `f`, **avant** l'exécution de sa première instruction : des emplacements sont associés à `x` et `y` (initialisés avec 5 et 24). Dans l'environnement défini (statiquement) par `f`, les variables locales `monX` et `monY` ne sont temporairement plus visibles et perdent leurs marques. La variable globale `res` reste visible à cet endroit : les marques dans l'environnement doivent refléter l'arbre de portée. Ne restent « marquées » que les variables visibles en cet endroit du programme d'après cet arbre des portées.

7 Ce qui suit **n'est pas applicable aux paramètres de type « tableau simple »** qui font l'objet d'un traitement spécifique décrit ultérieurement, en liaison avec les « pointeurs ».

8 Dans ce modèle nous allouons un emplacement pour les variables globales en fond de pile. Dans la réalité elles sont plutôt placées dans le « tas ».

`res` ●	0
`monX`	1
`monY`	2
`x` ●	3
`y` ●	4
`v1` ●	5
`v2` ●	6

0	●	-1
1	●	5
2	●	12
3	●	**5**
4	●	**24**
5	●	⊥
6	●	⊥
7		⊥

4. Affectation aux variables `v1` et `v2` (l'état mémoire change, mais pas l'environnement) :

`res` ●	0
`monX`	1
`monY`	2
`x` ●	3
`y` ●	4
`v1` ●	5
`v2` ●	6

0	●	-1
1	●	5
2	●	12
3	●	5
4	●	24
5	●	**29**
6	●	**120**
7		⊥

5. Test de la condition `v1 > v2` et entrée dans `B2` : La variable locale `res` de `B2` masque la variable globale `res` qui perd sa marque au profit de la nouvelle déclaration. On se place **après** l'initialisation de `res` :

`res`	0
`monX`	1
`monY`	2
`x` ●	3
`y` ●	4
`v1` ●	5
`v2` ●	6
`res` ●	7

0	●	-1
1	●	5
2	●	12
3	●	5
4	●	24
5	●	29
6	●	120
7	●	**-91**

L'environnement comporte actuellement deux variables `res`, mais seule celle du bloc `B2` est visible: la variable globale `res` existe toujours et est associée à l'emplacement `0` mais n'est plus l'objet désigné par `res` en cet endroit du programme : elle est masquée par la variable locale homonyme du bloc `B2`, associée à l'emplacement `7`.

6. Appel à `teste(res)`. On se place au début du corps de `teste`. **On notera les changements dans l'environnement**, conformes à l'arbre des portées : dans `teste`, mentionner `res`, c'est à nouveau mentionner la variable globale, pas celle du bloc `B2`.

res ●	0
monX	1
monY	2
x	3
y	4
v1	5
v2	6
res	7
val ●	8

0	●	-1
1	●	5
2	●	12
3	●	5
4	●	24
5	●	29
6	●	120
7	●	-91
8	●	**-91**

7. Exécution du corps de `teste` : on compare les valeurs du paramètre `val` et de la variable globale `res`. On imprime donc `Cas1` puis on quitte la fonction `teste` et on se retrouve donc à nouveau dans `f`, dans le bloc `B2.` L'emplacement 8 n'est plus valide puisqu'il a été alloué pour une variable locale d'un appel de fonction qui est terminé : l'espace occupé par cette variable locale est considéré comme libéré, donc n'est plus « marqué ». La situation illustrée est celle juste **après** le retour de l'appel de fonction et **avant** de quitter `B2`. L'environnement est celui qui existait à l'étape 5 mais l'état mémoire a été modifié depuis.

res	0
monX	1
monY	2
x ●	3
y ●	4
v1 ●	5
v2 ●	6
res ●	7

0	●	-1
1	●	5
2	●	12
3	●	5
4	●	24
5	●	29
6	●	120
7	●	-91
8		-91

8. On quitte `B2` et on retrouve l'environnement précédent. On indique la situation juste après avoir exécuté `res = -12`, où `res` est la variable globale d'après l'arbre des portées. Puisqu'on a quitté `B2`, il serait incohérent d'accéder à l'emplacement 7 qui était associé à une variable locale à `B2`.

`res` ●	0
`monX`	1
`monY`	2
`x` ●	3
`y` ●	4
`v1` ●	5
`v2` ●	6

0	●	-12
1	●	5
2	●	12
3	●	5
4	●	24
5	●	`29`
6	●	`120`
7		`-91`
8		`-91`

9. On évalue l'expression `return (17)`, on quitte la fonction `f` et on se retrouve dans la méthode `main` après l'affectation finale à `res:`

`res` ●	0
`monX` ●	1
`monY` ●	2

0	●	17
1	●	5
2	●	12
3		5
4		24
5		`29`
6		`120`
7		`-91`
8		`-91`

Lae modèle permet de faire apparaître explicitement des variables qui sont toujours associées à des emplacements mais qui ne sont pas référençable en un endroit donné du texte du programme, compte-tenu des règles de portée : celles qui ne sont pas marquées par ● **dans l'environnement**. Dans la **mémoire**, l'absence de la marque ● fait explicitement apparaître des emplacements dont la valeur a été définie à un moment de l'exécution mais qui ne doivent plus à un autre moment être considérés comme définis: y accéder doit être interprété comme une erreur de programmation. Cet emplacement pourra par exemple être réutilisé pour une variable locale d'une fonction appelée ultérieurement.

On remarque l'importance de l'**aspect statique de la portée des variables** : à l'étape 6 (pendant l'exécution de `teste`), la déclaration à considérer pour `res` est celle de la variable globale et non le `res` du `B2` de `f`, bien que ça soit cette déclaration qui ait eu lieu le plus récemment puisque l'appel à `teste` est fait depuis `B2`. Ce qui définit **la portée et la visibilité** c'est l'**imbrication statique des blocs**, pas le déroulement dynamique des appels de fonctions.

Résumé :

*Environnement: en C++ il est défini **statiquement** par :*

1. les variables globales non masquées localement dans une fonction ;
2. les paramètres formels et les variables locales non masquées localement par un bloc ;
3. les déclarations liées aux blocs du corps de la fonction, non masquées localement par un bloc plus interne.

Appel de fonction (**dynamique**, cas du **passage par valeur**):

1. évaluation (dans un ordre non spécifié) et mémorisation des valeurs des arguments de l'appel ;
2. allocation des emplacements associés aux paramètres formels de la fonction et initialisation à l'aide des valeurs des arguments ;
3. allocation des emplacements associés aux variables locales de la fonction ;
4. exécution du corps de la fonction jusqu'à atteindre un `return` ou la fin du corps de la fonction, en gérant les déclarations locales aux éventuels blocs internes ;
5. si le type de retour de la fonction n'est pas `void`, mémorisation de la valeur de retour, libération des emplacements associés aux paramètres formels et aux variables locales puis copie de la valeur de retour dans l'emplacement cible dans la fonction appelante.

Du modèle à la réalité (ou presque !) [9]

La gestion dynamique des environnements (représentée par le symbole ● qui suit un identificateur dans la partie environnement de notre modèle) peut sembler compliquée. En pratique on ne « marque » et « démarque » pas les identificateurs pour reconstituer l'environnement courant : **à la compilation,** le compilateur vérifie que les instructions des blocs n'accèdent effectivement qu'aux identificateurs auxquels ils ont droit: cela se fait facilement en considérant une pile de blocs de déclarations, qu'on empile et dépile au fur et à mesure qu'on entre et sort des blocs. En tout point du programme le compilateur peut donc savoir si un identificateur est global ou local (au sens large : y compris les blocs internes) à la fonction, et quel est son rang dans la liste des identificateurs de même statut (global/local). Dans le fichier compilé, il n'existe plus de noms de variables, juste l'indication d'un décalage par rapport au début des emplacements associés aux variables globales ou par rapport au début des emplacements associés aux objets **locaux à la fonction courante**. À l'exécution, il suffit de connaître d'une part l'emplacement de la première variable globale (cette adresse ne changera pas pendant toute l'exécution du programme), d'autre part l'emplacement de la première variable locale de la fonction courante (cette adresse dépend des appels de fonctions en cours et ne peut pas être prédite). En gérant à l'exécution pour chaque fonction ces deux adresses de base, il suffit d'ajouter le rang d'une variable dans la liste des variables de même statut global/local pour connaître l'emplacement recherché. Sans se soucier des autres variables qui existent mais auxquelles on sait qu'on n'accédera jamais pendant l'exécution de la fonction courante. Par exemple, à l'étape 6 du déroulement précédent, tout se passe comme si la situation était celle ci-dessous, dans laquelle ne sont présents que les deux seuls contextes logiquement accessibles : celui du niveau global et celui de la fonction courante. Les autres contextes (ceux de `main` ou de `f`) sont sauvegardés ailleurs (d'où la discontinuité entre les emplacements 0 et 8 qui apparaissent ci-dessous dans l'association entre l'environnement et la mémoire : les emplacements manquants sont toujours là mais « invisibles » tant qu'on est dans `teste`). La variable globale `res` est à un décalage 0 du début des emplacements pour les variables locales : l'emplacement d'adresse *contexte global + 0* donc *0 ;* La variable locale `val` est à un décalage 0 du début des emplacements pour les paramètres et variables locales de `teste`, donc à l'emplacement d'adresse *contexte*(`teste`) + 0 donc 8. Si `teste` avait une seconde variable locale, son rang dans la liste des variables locales serait 1, donc son emplacement serait à l'adresse *contexte*(`teste`)+1, donc 9. Il n'y a pas besoin de chercher dynamiquement dans l'environnement les emplacements associés à `res` et `val`.

Contexte de `teste`	`val` ●	8
Contexte global	`res` ●	0

0	●	-91
1	●	-91
2	●	120
3	●	29
4	●	24
5	●	5
6	●	12
7	●	5
8	●	-1

Quand on quitte `teste`, on retire le « contexte » de la fonction appelée (`teste`) et on restaure le contexte de la fonction appelante (`f`) sauvegardé à l'étape 5 pour retrouver la situation de l'étape 7.

9 **Cette partie peut être ignorée en première lecture !**

4. Appels récursifs de fonctions

Dans les exemples précédents, à chaque appel de fonction on associe de nouveaux emplacements (gérés en pile) pour les paramètres formels et les appels de fonctions. Ces emplacements sont libérés quand on quitte la fonction. On pourrait se demander s'il n'était pas possible de faire plus simple : puisque le compilateur est capable de connaître en chaque point le nombre de variables visibles (selon les blocs) pour gérer la portée, il est donc capable de connaître le nombre maximum de variables locales nécessaires pour chaque fonction, donc le nombre **maximum** d'emplacements disponibles. Pourquoi ne pas leur associer une fois pour toutes un emplacement fixe, ce qui serait plus simple et efficace (ce qui était fait dans certains langages plus anciens) ?

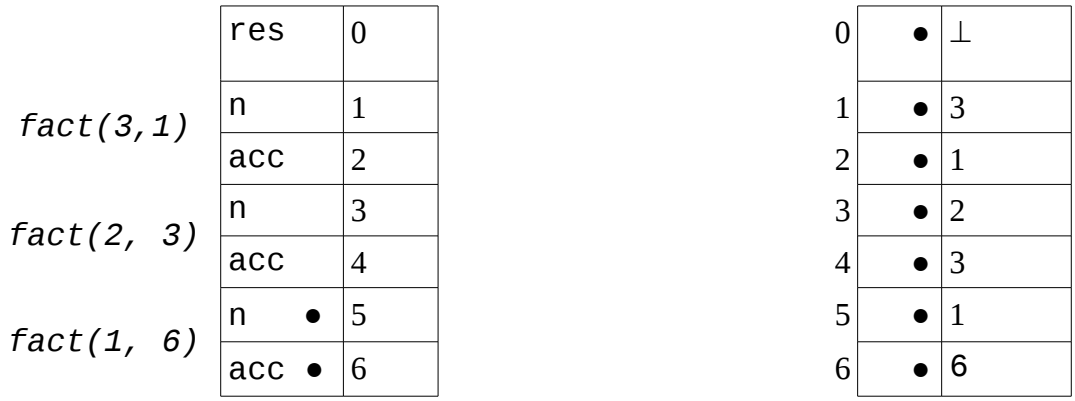
La réponse tient au fait que dans le langage C++ (comme dans tous les langages modernes), les fonctions peuvent être **récursives**, c'est à dire peuvent s'appeler elles-mêmes, directement ou non. A chaque nouvel appel à une fonction on suspend l'invocation de la fonction courante et on en crée une nouvelle instance en associant de **nouveaux** emplacements aux paramètres et variables locales de la nouvelle instance de la fonction. Ces emplacements seront libérés comme habituellement lorsque l'appel récursif de fonction terminera et on retrouvera alors les **associations précédentes** des identificateurs. Les règles de portée font que chaque fonction ne peut accéder qu'à ses propres objets locaux ou aux objets globaux, mais jamais aux objets locaux de ses instances précédentes. Les règles de gestion de l'environnement sont donc conservées sans changement. Nous l'illustrons ci-dessous pour un exemple très simple, que nous pourrions bien sûr programmer non récursivement mais qui a l'avantage d'être concis !

`// Suppose n ≥ 0.`
`// fact(n, acc)` calcule `n!*acc`. Donc l'appel `fact(3, 1)` dans `main` calcule `3!`

```
int fact(int n, int acc) {
    if (n <= 1) {  return acc; }          (*)
    else { return fact(n-1, n * acc); } // appel récursif
}

int main(void) {
  int res;
  res = fact(3, 1);
  printf("Valeur de 3! = %d\n", res);
}
```

Nous illustrons directement la situation juste avant d'exécuter l'instruction (*)



Les 5 premières lignes de l'environnement représentent les contextes créés par les appels à `main`, `fact(3,1)`et `fact(2,3)`qui sont tous suspendus dans l'attente du résultat de l'appel récursif `fact(1,6)`. La séquence d'appels récursifs à `fact` s'arrête quand on appelle `fact` avec `n ≤ 1`, la fonction renvoyant directement la valeur de `acc` (ici `6`). Vérifier qu'on n'aboutit pas à une chaîne infinie de calculs est un aspect crucial de la programmation à l'aide de fonctions récursives. La valeur renvoyée par le dernier appel récursif va être renvoyée successivement d'un niveau au niveau précédent, en redescendant la chaîne d'appels récursifs. On voit sur l'exemple que les paramètres `acc` et `n` de `fact` ne sont pas associés à des emplacements constants mais à des emplacements qui dépendent du nombre d'appels suspendus : chaque invocation de `fact` a entraîné l'allocation de nouveaux emplacements pour ses paramètres/variables locales. On ne peut plus associer une fois pour toutes un emplacement à une variable : son emplacement va dépendre de « l'histoire » du calcul.

Une autre version de la fonction `fact` :

```
int factBis(int n) { return (n <= 1 ? 1 : n * factBis(n-1)); }

int main(void) {
  int res = factBis(3);
}
```

Si on déroule l'exécution de `factBis`, on remarque que cette facilité a un coût caché en termes de place mémoire nécessaire : dans `factBis` les multiplications sont suspendues le temps que l'appel récursif termine et fournisse le second opérande. Dans `fact` le compilateur remarque qu'il n'y a aucun calcul à faire quand on redescend la chaîne d'appels récursifs: il peut optimiser l'exécution de telles fonctions en réutilisant les emplacements de `n` et `acc` puisque ces variables ne servent plus après la fin de l'appel récursif. Il est alors inutile d'allouer de nouveaux emplacements pour les variables locales de la fonction récursive et l'usage de la mémoire est plus économe.

```
int factTer(int n) { // version itérative. Ne nécessite que 3 emplacements.
  int res = 1;
  for(int i = n; i > 1; i--) { res = res * i; }
  return res;
}
```

L'intérêt des versions récursives, notamment `factBis`, est qu'elles reprennent directement la formulation mathématique (`0! = 0`, `1! = 1`, `n! = n * (n-1)!`) donc il est plus simple de vérifier qu'elles sont correctes. Dans d'autres exemples, les fonctions récursives sont le moyen naturel de programmation et la programmation itérative devient plus complexe. Un exemple célèbre est la **fonction d'Ackermann**, dont le résultat croît extrêmement rapidement en fonction des valeurs de `n` et `m` (voir une encyclopédie en ligne au sujet de l'intérêt de cette fonction en mathématiques et en algorithmique) :

$$A(m, n) = \begin{cases} n+1 & \textit{si } m = 0 \\ A(m-1, 1) & \textit{si } m > 0 \textit{ et } n = 0 \\ A(m-1, A(m, n-1)) & \textit{si } m > 0 \textit{ et } n > 0 \end{cases}$$

5. Passage par référence

Le passage par valeurs alloue pour les paramètres formels de **nouveaux** emplacements initialisés par les valeurs des arguments ; un passage de paramètre « par valeur » ne peut donc jamais modifier l'argument correspondant. Le « passage par référence »[10] va permettre de lever cette limitation : lors de l'appel de la fonction on ne passe plus la valeur de l'objet mais son adresse. Dans la fonction appelée, le compilateur sait qu'il gère une référence et non pas un objet standard et se charge directement de passer de l'adresse de l'objet à sa valeur selon le contexte (selon que l'identificateur apparaît en partie gauche ou en partie droite d'une affectation par exemple). Considérons l'exemple simple d'une fonction qui permute les valeurs de deux variables :

```
#include <iostream>
#include <cstdio>

using namespace std;

void echange(int &refa, int &refb) {
   int v1, v2;                                          (*)
   v1 = refa; v2 = refb;                                (**)
   refa = v2; refb = v1;                                (***)
}

int  main() {
   int x = 12, y = 13;
   printf("Valeur de x: %d, valeur de y: %d\n", x, y);
   echange(x, y);                                       (****)
   printf("Valeur de x: %d, valeur de y: %d\n", x, y);
   return 0;
}
```

Nous obtenons les impressions suivantes :

```
Valeur de x: 12, valeur de y: 13
Valeur de x: 13, valeur de y: 12
```

Nous donnons successivement les états aux instants référencés par (*) à (***). Dans les emplacements mémoire associés à `refa` et `refb` on trouve maintenant l'adresse des arguments correspondants et non leur valeur lors de l'appel. Pour visualiser que ces valeurs comme représentant des adresses, et non pas des entiers, nous les mettons en italique.

Situation à l'instant (*) : juste après la déclaration des variables locales `v1` et `v2`:

x	0
y	1
refa ●	2
refb ●	3
v1 ●	4
v2 ●	5

0	●	12
1	●	13
2	●	*0*
3	●	*1*
4	●	⊥
5	●	⊥

10 On peut aussi utiliser le passage par référence pour des raisons de performance afin d'éviter de copier des objets volumineux lors d'un appel de fonction. Il faut alors être prudent puisqu'on peut modifier par erreur l'argument correspondant. En TD nous avons vu l'usage du qualificatif `const` pour se protéger de telles erreurs.

Situation à l'instant (**) : lors des affectations `v1 = refa; v2 = refb;` le compilateur sait qu'il manipule des références, et donc que dans l'affectation `v1 = refa` la valeur à affecter à `v1` n'est celle stockée dans l'emplacement associé à `refa` mais celle stockée à l'emplacement dont l'adresse est la valeur stockée dans l'emplacement associé à `refa` (on parle « d'indirection » : le premier emplacement contient comme valeur l'adresse d'un autre emplacement).

x	0
y	1
refa ●	2
refb ●	3
v1 ●	4
v2 ●	5

0	●	12
1	●	13
2	●	*0*
3	●	*1*
4	●	12
5	●	13

Situation à l'instant (***) : on retrouve un raisonnement similaire pour les affectations `refa = v2` et `refb = v1` : le compilateur sait que les cibles réelle des affectations ne sont pas `refa` et `refb` mais les emplacements dont `refa` et `refb` contiennent les adresses.

x	0
y	1
refa ●	2
refb ●	3
v1 ●	4
v2 ●	5

0	●	13
1	●	12
2	●	*0*
3	●	*1*
4	●	12
5	●	13

Situation à l'instant (****) : Après le retour de la fonction `echange`, on se retrouve dans la situation ci-dessous : les paramètres et variables locales de `echange` ont été dépilés et les variables `x` et `y` ont bien eu une permutation de leurs valeurs initiales.

x ●	0
y ●	1

0	●	13
1	●	12
2		*0*
3		*1*
4		12
5		13

Pas d'affectation entre références: on a vu dans l'exemple précédent comment le compilateur traitait deux affectations telles que `v1 = refa` d'une part et `refa = v2` d'autre part. Que se passe-t-il avec la version légèrement plus concise ci-dessous :

```
void echange2(int &refa, int &refb) {
  int aux;
  aux = refa; refa = refb; refb = aux;
}
```

Que représente une affectation `refa = refb` ? Est-ce le même traitement que ci-dessus (c.à.d l'objet référencé par `refa` reçoit la valeur de l'objet référencé par `refb`) ou bien s'agit-il d'une affectation entre références (c.à.d. `refa` référence maintenant le même objet que `refb`) ? La bonne réponse est la première : dans le langage C++ il n'existe pas d'affectations entre références, l'affectation se fait toujours relativement aux objets référencés. Une référence a le statut d'une constante, elle doit être initialisée lors de sa déclaration et ne peut pas être modifiée par une affectation (ou une autre opération). Ceci ne veut pas dire qu'on ne peut pas écrire `refa = refb`, juste que cela n'a pas le sens d'une affectation entre références malgré l'apparence. On peut vérifier sa bonne compréhension avec l'exemple ci-dessous :

```
void mystere() {
  int v1 = 12, v2 = 35;
  int &p1 = v1;                          // p1 est une référence sur v1
  int &p2 = v2;                          // p2 est une référence sur v2
  p1 = 37;
  p1 = p2;
  p1 = 56;
  cout << "v1: " << v1 << ", v2: " << v2 << endl; // Imprime quoi ?
  int v3 = 42, v4 = 13;
  int &r = v3;
  r = v4;
  r = 55;
  // Qui vaut 55 : v3 ou v4 ?
  cout << "v3: " << v3 << ", v4: " << v4 << endl;
}
```

6. Durée de vie d'un objet: c'est l'intervalle de temps à l'exécution pendant lequel un emplacement peut légitimement être accédé. Il s'agit donc d'une notion **dynamique**. Comme on va le voir dans l'exemple suivant, s'intéresser de trop près aux adresses d'objets peut amener à détenir des adresses d'objets qui n'existent plus vraiment au moment où on les utilise ou qui représente autre chose:

```
int & f(int v) {      // f retourne une référence sur int
  int res = v;
  int &pv = v;        // pv est donc une référence sur le paramètre formel
  res = v + 1;
  // syntaxiquement correct, sémantiquement absurde[11] : retourne une référence sur un
  // objet dont la duree de vie va expirer avec la fin de l'exécution de l'appel de fonction
  return pv;                                  (*)
}
```

11 Le compilateur émet un « warning » !

```
// cette fonction ne sert qu'à occuper trois emplacements dans la pile
void g(int val) {
  int x = val + 1, y = val + 2;            (***)
}

int main(void) {
  int &p = f(56);                          (**)
  printf("Valeur : %d\n", p);
  g(25);                                   (****)
  printf("Valeur : %d\n", p);
}
```

Nous donnons les situations aux instants `(*)`, `(**)`, `(***)`et `(****)`. Comme précédemment les valeurs en mémoire qui représentent des adresses sont en italique.

p	0
v ●	1
res ●	2
pv ●	3

(*)

0	●	⊥
1	●	56
2	●	57
3	●	*1*

Ci-dessus, la valeur à l'adresse `0` associée à la variable `p` de `main` est ⊥ : la valeur **avant** affectation. **Après** l'affectation, au retour de `f`, elle vaudra *1*, la valeur de `pv` en sortie de `f`. On remarque qu'à l'instant (**), `p` contient l'adresse `1`, donc référence une case mémoire qui n'est plus valide (non marquée ●) dans ce contexte : la fonction `f` retourne l'adresse de l'emplacement associé à son paramètre (on aurait le même problème avec une variable locale), emplacement qui n'est plus valide dès la fin de l'appel !

p ●	0

(**)

0	●	*1*
1		56
2		57
3		*1*

Ci-dessous nous sommes dans `g` : l'emplacement `1` est de nouveau légitime mais est maintenant associé à `val` ! L'affectation à `val` fait changer subrepticement la valeur de l'objet sur lequel pointe la variable `p` du `main`.

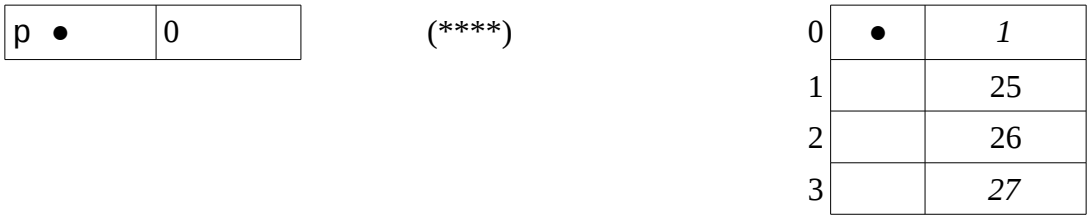
p	0
val ●	1
x ●	2
y ●	3

(***)

0	●	*1*
1	●	25
2	●	26
3	●	27

La situation finale `(****)` est donc celle ci-dessous qui amènera cette fois-ci (probablement) l'impression de 25 alors qu'aucune affectation à `p` n'a eu lieu entre les instants (**) et (***) ! En fait

les deux accès à `p` dans `main` sont syntaxiquement corrects mais illégaux. Nous avons indiqué qu'ils imprimaient respectivement 56 et 25 mais ils pourraient tout autant terminer en erreur (« accès illégal ») si on considère que ces emplacements mémoires n'appartiennent plus au programme. Nous devons donc aussi nous préoccuper de la **« durée de vie »** des objets et ne pas donner de moyens de **référencer un objet au delà de sa durée de vie**. La durée de vie d'un objet déclaré localement à une fonction est la durée de cette invocation de la fonction.



Pour finir, un exemple similaire, relatif au tas :

```
#include <iostream>
#include <vector>

using namespace std;

// la fonction ne retourne pas la valeur mais une référence sur l'élément du vecteur à cet indice.
// On suppose l'indice correct. Remarquez que vect est passé par référence.
int & valeur(int indice, vector<int> &vect) { return vect[indice]; }

void print(vector<int> & vect) {
  cout << "Contenu du vecteur:";
  for(size_t i = 0; i < vect.size(); i = i+1) {
    cout << " " << vect[i];
  }
  cout << endl;
}

int main() {
  vector<int> v = { 0, 1, 2, 3, 4 };
  print(v);                       // imprime { 0, 1, 2, 3, 4 }
  int &ref = valeur(2, v);
  ref = 57;                       // modifie la valeur contenue dans le vecteur
  print(v);/                      // imprime { 0, 1, 57, 3, 4 }
  v.clear();                      // réinitialise le vecteur
 // l'adresse référencée par ref est maintenant invalide puisque le vecteur a été réinitialisé. On met
 // 32 dans un emplacement mémoire qui n'a plus de sens pour notre programme.
  ref = 32;
  print(v);
  return 0;
}
```

7. Pointeur et références en C++ (et en langage C) :

Comme nous l'avons vu dans le chapitre précédent, les « références » permettent de travailler de façon implicite sur des objets via leur adresse. C'est notamment le cas pour le « passage par référence » de paramètres, que ce soit pour modifier dans une fonction un objet passé en paramètre, ou pour éviter de copier un « gros » objet comme cela serait le cas avec du passage par valeur. La dernière utilisation des références est d'autoriser des « partages » c'est-à-dire d'avoir plusieurs moyens de désigner un objet en conservant son adresse dans des références. Les références en C++ ont été conçues de manière à permettre une utilisation relativement sécurisée :

- une référence doit être initialisée lors de sa déclaration, ce qui évite d'avoir des références ne « pointant » sur rien (mais on a vu que si on prenait mal en compte la notion de « durée de vie » d'un objet, il était possible de conserver une référence sur un objet qui n'existe plus).
- On ne peut pas réaliser d'affectation entre références, ni d'égalité entre références. Ces opérations sont toujours relatives aux objets référencés, pas aux références elles-mêmes. Il n'est pas nécessaire de jongler mentalement entre la référence et l'objet référencé ; c'est le compilateur qui fait cette gymnastique avec certaines limitations.

Dans le langage C, ancêtre de C++, cette notion de référence n'existait pas et on devait manipuler explicitement les adresses des objets via des éléments d'une certaine catégorie, appelées « pointeurs ». Cette notion existe toujours en C++ même si elle est d'un usage moins fréquent puisque les références sont plus faciles et sûres à manipuler dans de nombreux cas. Nous présentons d'abord la notion avant de discuter les cas où les pointeurs explicites sont nécessaires en C++.

Définition et utilisation des pointeurs : On peut définir des pointeurs sur n'importe quel type d'objet. Définir un pointeur c'est vouloir manipuler explicitement l'adresse d'un objet :

- la définition d'un type « pointeur » correspond à indiquer que la valeur d'une variable de ce type ne représente pas directement une « valeur » au sens habituel mais l'adresse (emplacement) d'un objet du type pointé. Du point de vue de l'implémentation, un pointeur est représenté comme une référence : c'est l'adresse d'un objet. Dans notre modèle, la valeur d'une variable de type pointeur sera donc un numéro d'emplacement mémoire ;
- L'opérateur `&` correspond à obtenir l'adresse d'une variable : passer explicitement d'un objet à un pointeur sur cet objet. Il s'agit bien ici d'un opérateur (l'opérateur « adresse de … ») qui interviendra donc dans une **expression** : en partie droite d'affectation, pour l'initialisation d'une déclaration de variable, etc. À distinguer de l'usage de `&` pour indiquer qu'une variable est une référence, qui apparaît donc dans une **déclaration** (déclaration du paramètre d'une fonction, de son type de retour, du type d'une variable locale, etc).
- un opérateur « de déréférencement » (le `*` unaire préfixe) pour passer du pointeur à l'objet pointé. Si `p` est un pointeur, `*p` est l'objet pointé. C'est l'opérateur réciproque de `&`.
- L'opérateur `->` combine pointeur et accès à un champ de structure : `p->val` est un raccourci pour `(*p).val`.
  Attention: `*p.val` est syntaxiquement correct mais se lit `*(p.val)`. Si `p` est un pointeur sur une structure dont le champ `val` est un `int`, à la compilation vous aurez le message d'erreur « `request for member 'val' in something not a structure or union` » : en effet, `p` n'est pas une structure avec un champ `val`, c'est `*p` qui a cette propriété. On fera aussi attention au message « `assignment makes pointer from integer without a cast` » qui indique une confusion entre entiers et pointeurs.

Exemple. On considère l'exemple de pointeurs sur des `int`:

1. `int *ptr, *ptr2;` déclare `ptr` et `ptr2` comme des variables de type « pointeur sur `int` ». Le symbole **`*`** doit être explicitement répété devant `ptr2`, sinon `ptr2` sera une variable de type `int` et non pas « pointeur sur `int` ». Pour éviter ce problème, le plus simple est de nommer un type explicite via une directive `typedef` qui évitera d'avoir à rappeler le symbole `*` dans les déclarations :

   ```
   typedef int *IntPtr;    // IntPtr est de type « pointeur sur int »
   IntPtr ptr, ptr2;
   ```

2. On peut affecter à `ptr` l'adresse d'un objet de type `int` :

   ```
   int i = 32, j = 42;
   int *ptr1, *ptr2;
   ptr1 = &i;      // appel explicite à l'opérateur &, à la différence des références
   ptr2 = &j;      // de même ptr2 pointe sur j
   ```

Pour simplifier, nous ne faisons plus apparaître les marques et mettons en *italique* les valeurs qui doivent être interprétées comme des adresses, comme pour les références. Dans notre modèle, l'opérateur & consiste à lire dans le tableau « environnement » l'adresse de l'emplacement associé à la variable (la valeur de l'expression `&i` est donc `0`).

`i`	`0`
`j`	`1`
`ptr1`	`2`
`ptr2`	`3`

`0`	32
`1`	42
`2`	*`0`*
`3`	*`1`*

Attention : dans l'environnement nous n'avons **pas** associé à `ptr` et `ptr2` l'adresse de l'emplacement sur lequel la variable pointe. Comme toute variable, `ptr` et `ptr2` ont chacune leur emplacement propre. C'est le **contenu de ces emplacements** qui vaut l'adresse de l'emplacement de l'objet pointé.

3. Si `p` représente un pointeur sur un `int` alors `*p` représente la valeur de type `int` sur lequel pointe `p`. La notation `*p` peut apparaître dans une expression comme `*p+1` mais aussi en partie **gauche** d'affectation, pour indiquer qu'on veut modifier la valeur de l'objet pointé. Illustrons la situation après exécution de `*ptr1 = *ptr2+6` :

`i`	`0`
`j`	`1`
`ptr1`	`2`
`ptr2`	`3`

`0`	48
`1`	42
`2`	*`0`*
`3`	*`1`*

   Comme dans le cas des références, puisque `ptr1` pointe sur `i`, la modification effectuée via `*ptr1` a modifié le contenu de l'emplacement associé à `i`.

4. Pour l'affectation et l'égalité, il faut savoir si ces opérations portent sur les pointeurs ou sur les objets pointés. L'affectation `ptr2 = ptr1` est une **copie entre pointeurs**, qui permet d'accéder à un même objet via plusieurs pointeurs contenant la même adresse. Pour **copier les objets pointés** on écrit `*ptr2 = *ptr1`. L'égalité `ptr2 == ptr1` vérifie si `ptr1` et

`ptr2` contiennent la même valeur, c'est-à-dire pointent sur le même emplacement, pas si les objets sur lesquels ils pointent sont égaux, ce qui s'écrirait `*ptr2 == *ptr1`. Illustrons la situation avec le fragment de programme suivant:

```
int i = 32, j = 32; int *ptr1 = &i, *ptr2= &j;
```

i	0
j	1
ptr1	2
ptr2	3

0	32
1	32
2	*0*
3	*1*

Dans cet état, `ptr1 == ptr2` vaut `false` tandis que `*ptr1 == *ptr2` vaut `true`.

Si on exécute maintenant l'affectation `ptr1 = ptr2`, on obtient :

i	0
j	1
ptr1	2
ptr2	3

0	32
1	32
2	*1*
3	*1*

Bien sûr `ptr1 == ptr2` vaut `true` implique`*ptr1 == *ptr2` vaut `true` !

Initialisation des pointeurs et « pointeur nul » : on n'oubliera pas que les variables de type « pointeur » doivent être initialisées comme toute variable. Une déclaration `int *p;` définit `p` comme « pointeur sur un entier » mais ne définit pas sur quoi elle pointe. Par défaut la valeur courante de `p` sera interprétée comme une adresse arbitraire ! Pour exprimer que `p` ne pointe sur aucun emplacement on utilise en C++ (version 11) la notation `nullptr`:

```
int *p = nullptr;
…
if (p == nullptr) {
   // traitement du cas où p ne pointe sur aucun int
}
```

Résumons les différences d'écriture entre **référence** et **pointeur** :

Références	**Pointeurs**
`int i=42, j=33;`	`int i=42, j=33;`
`int &ref1 = i, &ref2 = j;`	`int *p1 = &i, *p2 = &j;`
`int &ref;  // KO : interdit`	`int *p;  // Ok mais p n'est pas initialisé`
`Ref1 = 25;  // OK : i passe à 25`	`*p1 = 25;`
`ref = ref2;  // entre objets référencés`	`p1 = p2;  // entre pointeurs`
	`*p1 = *p2; // entre objets pointés`
`ref == ref2  // entre objets référencés`	`p1 == p2  // entre pointeurs`
	`*p1 == *p2 // entre objets pointés`

Pointeurs et références représentent des objets de types différents et ne sont pas compatibles entre eux, même s'ils représentent tous deux des adresses. Leur logique d'usage est différente.

8. Allocation dynamique de mémoire et gestion du « tas » :

Jusqu'à présent, la seule manière que nous avions d'avoir des conteneurs de taille variable consistait à utiliser des conteneurs prédéfinis (`vector` ou `string`) qui permettent d'ajouter ou de retirer librement des éléments, sans souci de taille fixe à respecter. Comme indiqué dans l'introduction du chapitre II, ces conteneurs prédéfinis cachent dans le code de leurs fonctions l'allocation et la désallocation de mémoire sur le « tas ». En général on alloue sur le « tas » des objets dont la taille n'est pas connue à la compilation ou dont la durée de vie ne peut pas être prédite à l'avance.

Il existe d'autres cas où on a besoin de créer de nouveaux objets durant l'exécution d'un programme. Nous avons vu que si nous essayions de renvoyer une adresse (ou un pointeur) sur un objet local à une fonction, nous aurions des problèmes de durée de vie de l'objet retourné. Il faut donc que nous ayons un moyen d'allouer et de libérer de la mémoire sur le tas. Il existe pour cela deux notations fournies dans le langage C++ : `new` et `delete`. Notons un point important : il faut distinguer l'allocation ou la désallocation d'un objet isolé de l'allocation/désallocation d'un tableau d'objets. Dans ce dernier cas on doit le préciser au compilateur en utilisant la notation `new[]` et `delete[]`. L'opérateur `new` renvoie un **pointeur** sur l'objet (ou le tableau d'objets) alloué.

```
typedef struct _personne {
  string nom;
  struct _personne *parent;
  struct _personne *frere;          // 1 frère au maximum ici !
} Personne;

Personne * nouveauNe(string nom, Personne *parent) {
  Personne *res = new Personne;
  res->nom = nom; res->parent = parent; res->frere = nullptr;
  return res;
}

Personne * jumeaux(string nom1, string nom2, Personne *parent) {
  Personne *res = new Personne[2];
  res[0].nom=nom1; res[0].frere=&(res[1]); res[0].parent=parent;
  res[1].nom=nom2; res[1].frere=&(res[0]); res[1].parent=parent;
  return res;
}

void etatCivil(Personne *p) {
  if (p != nullptr) {
    cout << p->nom;
    if (p->parent != nullptr) {
       cout << " Parent: " << p->parent->nom;
    }
    if (p->frere != nullptr) { cout << " Frere:" << p->frere->nom; }
    cout << endl;
  }
}
```

```
void plusFrere(Personne *p1, Personne *p2) {
  if (p1->frere == p2 and p2->frere == p1) {
    p1->frere = nullptr; p2->frere = nullptr;
  }
}

int main() {
  Personne *p1 = nouveauNe("Adam", nullptr);
  Personne *p2 = jumeaux("Cain", "Abel", p1);
  etatCivil(p1);
  etatCivil(&p2[0]); etatCivil(&p2[1]);          // avant séparation des frères
  plusFrere(&p2[0], &p2[1]);
  etatCivil(&p2[0]); etatCivil(&p2[1]);          // après séparation des frères
  // assez d'histoire de famille
  delete p1;
  delete[] p2;
  return 0;
}
```

Notons quelques points importants :

- la définition (presque) « récursive » du type `Personne`, puisqu'on veut stocker dans chaque personne un lien vers un parent et un frère. L'identificateur `Personne` n'est connu du compilateur qu'une fois la déclaration du `typedef` terminée. On ne peut pas écrire directement `Personne *parent` ; On donne un nom à la `struct` et on utilise ce nom pour déclarer les champs.
- Les objets sont alloués dynamiquement durant l'exécution de `nouveauNe` et `jumeaux` mais comme ils sont alloués par `new`, ils sont créés sur le tas et non pas sur la pile. Ils subsisteront une fois qu'on aura quitté la fonction qui les a créés. Attention à ne pas écrire :

```
Personne * nouveauNeBugge(string nom, Personne *parent) {
  Personne res;
  res.nom = nom; res.parent = parent; res->frere = nullptr;
  return &res;                  // Bug : retourne l'adresse d'un objet local à la fonction
}
```

- Les notations un peu étranges comme `&p2[1]` : `p2` est un pointeur (puisqu'il doit contenir le résultat d'un appel à `new[]`), mais c'est surtout l'adresse d'un tableau de deux objets de type `Personne`. Pour obtenir un pointeur, il suffit d'appliquer l'opérateur `&` à l'objet considéré : `&p2[1]` se lit `&(p2[1])`, c.à.d. l'adresse de la case `p2[1]`. Nous verrons ultérieurement l'analogie tableau/pointeur dans les expressions en C++ (et C).

***Questions** :*

- *Peut-on faire cet exemple avec des vecteurs ou avec des références ?*
- *Ajouter dans `Personne` de quoi mémoriser la liste des enfants de la personne considérée. On ne connaît pas à l'avance le nombre d'enfants.*

Comme la fonction `plusFrere` prend en paramètre deux pointeurs, donc des adresses, il est possible à l'intérieur de la fonction de modifier les objets passés en paramètre à travers les pointeurs, comme dans le cas du passage par référence. Dans le langage C qui n'avait pas de notion de « référence », c'est comme cela qu'on simulait le « passage par référence » : on déclarait les paramètres considérés comme des pointeurs et on fournissait en argument les adresses des objets via l'opérateur &. Le langage C++ rend cela beaucoup plus clair.

Un exemple typique est celui de la fonction `scanf`, héritée du langage C (il faut ajouter `#include <cstdio>`) qui permet de faire des lectures « formatées ». Nous l'illustrons ci-dessous par une situation où nous souhaitons lire trois valeurs de différents types :

Un code utilisant l'opérateur `>>` de `istream` en C++ sur le flôt `cin` est le suivant :

```
int i; float f; char c;
cout << "Entrez un entier, un float et un caractère: ";
cin >> i >> f >> c;
```

L'opérateur `>>` de `istream` modifie la variable qu'il prend en paramètre (par référence) en lui affectant la prochaine valeur lue et renvoie en résultat son flôt d'entrée pour que la cascade de lectures puisse se poursuivre[12]. On va ainsi pouvoir lire successivement les 3 valeurs attendues.

Le code utilisant la fonction `scanf` est le suivant :

```
int i; float f; char c;
printf("Entrez un entier, un float et un caractère:");
scanf("%d%g%c", &i, &f, &c);
```

La fonction `scanf` n'est pas un opérateur comme `>>` qui pourrait procéder en cascade et elle ne peut pas non plus renvoyer en résultat les valeurs lues, puisqu'il lui faudrait ici renvoyer 3 résultats. Elle procède donc autrement : on lui passe en premier argument une chaîne qui décrit symboliquement le format de ce qu'il y a à lire (avec des conventions similaires à celles de `printf`) ainsi que les **adresses** des variables dans lesquelles stocker les valeurs lues. La fonction retourne le nombre de lectures cohérentes par rapport au format demandé (ce qui permet de savoir combien de variables ont reçu une valeur) et elle aura affecté les valeurs lues aux variables dont on lui aura passé les adresses. Dans l'exemple ci-dessous, attention à ne pas appliquer l'opérateur '`&`' à `pi` sinon on obtiendrait un « pointeur sur pointeur sur entier » et non pas le « pointeur sur entier » dont `scanf` a besoin.

```
int i, *pi = &i;
scanf("%d", i);       // Incorrect !
scanf("%d", &i);      // Correct !
scanf("%d", pi);      // Correct !
scanf("%d", &pi);     // Incorrect !
```

Désallocation d'objets créées dynamiquement : pour en terminer avec la notion de pointeurs et de gestion du « tas », mentionnons que si l'allocation de mémoire peut être faite explicitement, la désallocation doit aussi être faite explicitement : on ne peut plus compter sur la gestion en pile des paramètres et objets locaux aux fonctions. Manipuler des objets sur le « tas » induit deux risques :

- Ne pas désallouer des objets et « encombrer » le tas avec des objets devenus inutiles au risque de ne plus pouvoir en allouer de nouveaux. Dans l'exemple ci-dessous, la deuxième instruction écrase le seul moyen qu'on avait de désigner le premier objet alloué. Celui-ci occupera donc de la place dans le tas pour rien jusqu'à la fin de l'exécution du programme.

  ```
  Personne *p = new Personne;
  p = new Personne;
  ```

12 *Question* : que se passe-t-il dans une telle cascade de lectures si les valeurs entrées par l'utilisateur ne sont pas conformes au format attendu ?

- Continuer à utiliser un objet désalloué, donc qui n'existe plus vraiment :

```
Personne *p = new Personne;
Personne *p2 = p;
delete p;
cout << p2->nom << endl;    // accès erroné !
```

Le langage C++ définit des aides pour minimiser ces dangers ; ils seront à voir ultérieurement.

9. Tableaux « simples », Pointeurs, Accès indicé et « arithmétique de pointeurs »

Par « tableau simple » nous entendons un tableau tel que `int t[10]` et non pas des structures plus élaborées comme les `vector` qui offrent aussi un accès indexé aux éléments. Un tableau est une zone contiguë en mémoire contenant des éléments d'un même type :

```
int t[10];
```

L'accès indicé à un élément du tableau, via un indice `i` avec `0 ≤ i < 10`, correspond donc à accéder à la case située `i` cases après le début du tableau. L'accès indicé peut être vu comme un moyen spécifique aux tableaux de désigner une case mémoire, en profitant du fait que les cases sont contiguës. Comme tous les éléments sont du même type, connu du compilateur, ce dernier sait la valeur exacte du déplacement nécessaire en nombre d'emplacements.

Pointeur : une variable dont la valeur est l'adresse d'un autre objet :

```
int v = 12, w = 56;
int *p = &v, *p2 = &w;
```

À partir de `p` on peut accéder à l'objet pointé en déréférençant le pointeur : `*p = *p+1` et `p` au cours de sa vie peut pointer dynamiquement ou pas (`p = nullptr`) sur un objet, et changer l'objet sur lequel il pointe (`p = p2`). À l'opposé, un identificateur de tableau a le statut d'une constante et on ne peut pas lui faire changer la zone mémoire qu'il désigne implicitement : une affectation telle que `t = …` sera rejetée par le compilateur.

Par contre, C++ (et C) maintiennent une certaine analogie dans l'usage des tableaux et des pointeurs **dans les expressions** en permettant l'usage des expressions indicées à partir d'un pointeur et non pas seulement d'un identificateur de tableau. Dans une expression, c'est même l'écriture sous forme de tableau qui est traduite en une expression avec un pointeur ! La logique est que **si** un pointeur pointe sur une **zone contiguë d'éléments de même type**, une expression indicée à partir de ce pointeur devrait avoir le même effet que si on gérait globalement la zone comme un tableau. Si on écrit :

```
int t[10];
int *pt = &t[0];      // ou de façon plus concise: int *pt = t;
```

l'expression `pt[5]` doit avoir le même effet que `t[5]`. Le langage autorise même une certaine liberté dans les expressions mettant en œuvre des déplacements dans une zone contiguë (on parle abusivement « d'arithmétique de pointeurs », alors qu'il s'agit juste de décalages relatifs à l'intérieur d'une zone supposée allouée correctement) :

```
int t[10];
int *pt = &t[5];   // pt pointe sur la case d'indice t[5]
*pt = 12;          // équivalent à t[5] = 12;
pt = pt + 2;       // pt pointe 2 int après t[5], donc sur t[7]
*pt = 53;          // équivalent ici à t[7] = 53;
```

Cependant il convient de garder en mémoire que **ceci n'a de sens que si** :

- `pt` pointe bien à l'intérieur d'un « tableau » c'est-à-dire sur une zone **contiguë** d'éléments d'un même type ;
- `i` correspond à un déplacement correct dans cette zone contiguë, telle qu'elle a été allouée.

Par exemple, si après le fragment de programme précédent on écrivait `pt = pt +3;` l'instruction serait acceptée par le compilateur mais serait sémantiquement illégale puisqu'elle correspondrait à un emplacement d'adresse `t+5+2+3`, c'est-à-dire `t[10]`, qui est illégal puisque `t` est un tableau de 10 éléments rangés de l'adresse `&t[0]` à `&t[9]`.

Ce « décalage » peut être négatif (avec les mêmes réserves que ci-dessus) et est intelligent par rapport au type d'éléments stockés : considérons l'exemple suivant :

```
Personne t[10];
Personne *pt = t;
```

Un objet `Personne` nécessite trois emplacements[13], donc `t` va occuper 30 emplacements mémoire. Une instruction telle que `pt = pt+2;` correspond bien à un déplacement de 2 objets de type `Personne`, donc de 6 emplacements, et non pas juste deux emplacements après l'adresse `t`.

Cette analogie entre pointeurs et tableaux va permettre de généraliser certaines fonctions qui parcourent des tableaux :

Écrire une fonction qui recherche la première occurrence d'un entier `v` dans un tableau `t` d'entiers. On passe en paramètre supplémentaire un pointeur sur la fin du tableau. La fonction renvoie un pointeur sur l'occurrence correspondante et `nullptr` s'il n'existe aucune occurrence de l'entier

```
int * cherche(int *debut, int *fin, int v) {
   for(int *p = debut; p < fin; p = p+1) {
     if (*p == v) { return p; }
   }
   return nullptr;
}
```

Cette fonction n'est correcte que si on passe en argument à `cherche` un tableau d'entiers. Par contre elle ne suppose pas que `debut` pointe bien sur le début d'un tableau et `fin` sur sa fin On peut donc passer en paramètre différents pointeurs à l'intérieur du tableau pour trouver les occurrences dans un sous-tableau :

13 Rappel : dans notre modèle, à la différence de la réalité, tous les types de base (et aussi `string`) occupent **un** emplacement !

```
void teste() {            // remplace toutes les occurrences de 1 par 5
  int t[] = { 1, 2, 2, 3, 1, 1, 0 };
  int *p=t, *fin = t + 7;
  while (p < fin) {
     int *p2 = cherche(p, fin, 1);   // recherche de la prochaine occurrence
     if (p2 != nullptr) {
        *p2 = 5;
         p = p2+1;       // se placer après l'occurrence trouvée.
     } else { break; }  // plus d'occurrence => terminé.
  }
}
```

Si `t` est un tableau de 100 entiers, un appel `cherche(t+10, t+90, 1)` ne cherchera la première occurrence de `1` qu'entre les indices `10` et `89`.

Q : si on écrit la fonction `cherche` en utilisant des références au lieu de pointeurs, quels problèmes cela pose-t-il ?

Passage de tableaux en paramètre : les tableaux (tels que `int t[10]`) constituent un cas particulier en C++, comme en C, car ils sont **toujours** passés **par adresse** : en réalité le compilateur passe en paramètre le tableau sous la forme d'un pointeur sur son premier élément. Dans le code de la fonction, toute expression indicée est aussi traduite en pointeur.

```
void f() {
  int t[] = { 10, 20, 30, 40, 50 };
  g(t);
}
```

L'appel `g(t)` dans `f` est traduit en C++ par `g(&t[0])`, puisque `&t[0]` est l'adresse de la première case de `t`. L'en-tête de la fonction `g` peut être écrit sous l'une des deux formes suivantes :

```
void g(int tab[]) { … }
void g(int *tab) { … }
```

La première forme est même traduite automatiquement en la seconde. Il en est de même si on veut renvoyer un tableau en paramètre, sauf que la notation `int []` pour le type de retour est interdite et qu'on doit forcément utiliser `int *`.

10. Passage d'arguments au lancement d'un programme :

Pour lancer l'exécution d'un programme, C++ a repris la façon de faire du langage C : on démarre l'exécution par l'appel de la fonction `main`, à qui l'on peut passer des arguments (ce que nous n'avons encore jamais utilisé). L'en-tête de la fonction `main` est en fait le suivant (la fonction prend un troisième paramètre que nous ne décrivons pas ici) :

```
int main (int argc, char * argv[]) { … }
```

Le premier paramètre contient le nombre de paramètres fournis au programme, en comptant le nom du programme comme un paramètre ; le second argument est un tableau de « pointeurs sur caractères » : la dimension de ce tableau est `argc`. Chaque élément du tableau est une « chaîne de caractères » stockée selon les conventions du langage C : un tableau de `char` qui doit **forcément**

contenir le caractère noté `'\0'` (à ne pas confondre avec `'0'`). Ce caractère sert de « marqueur de fin de chaîne ». Par convention, le contenu de la chaîne est la séquence de caractères jusqu'au `'\0'` exclus. Le langage C ne définissait malheureusement pas de type `string`, ce qui était la source de nombreux bugs liés à l'oubli du caractère `'\0'`. Nous savons comment initialiser une `string` C++ à l'aide d'une chaîne littérale comme `"hello"` ou une chaîne C. Dans l'autre sens, pour passer d'une `string` C++ à une chaîne C on lui applique la méthode `c_str()`.

Exemple :

```
string st =  "hello";
printf("la chaine:%s\n", st.c_str());
```

Un appel `printf("la chaine:%s\n", st)` donnerait une erreur à la compilation puisqu'une `string` ne correspond pas au format attendu pour un argument décrit par `%s`.

Exemple d'utilisation des paramètres fournis au lancement du programme : chaque élément du tableau `argv` est une chaîne de caractères même s'il ressemble à un entier ou à un autre type de nombre. Il vous appartient de convertir la chaîne en un nombre le cas échéant. Montrons un exemple de traitement des arguments : ce programme prend en paramètre au plus 3 arguments donnés sous la forme d'une option `-c` ou `-d` ou `-p` suivi d'un entier. On a aussi l'option `-h` ou `-?` qui permet d'avoir la syntaxe de lancement du programme. L'ordre des options est libre et on peut les écrire en majuscules ou en minuscules. Chaque option est supposée être suivie d'un entier et permet de modifier la valeur d'une des variable du programme qui a sinon une valeur par défaut. Exemple d'appel : `monProgramme -d 32 -c 56 -p 3`

```
int main(int argc, char* argv[]) {
  int dimension=42, proportion=42, cible=42;  // valeurs par défaut des variables
  size_t i;
  // Première partie : Gestion des options à l'appel.
  // Chaque option est supposée fournir la valeur d'une des 3 variables entières
  i = 1;                    // Ignorer le nom du programme exécuté, ici monProgramme
  while (i < argc) {
   if (argv[i][0] == '-') {  // si ça commence par '-', c'est supposé être une option
    switch (argv[i][1]) {
      case 'd': case 'D':
        { istringstream buf(argv[i+1]); // pour lire dans une chaîne
          buf >> dimension;
          i += 2;                       //  passer les deux éléments traités
          continue;
        }
      case 'p': case 'P':
        { istringstream buf(argv[i+1]);
          buf >> proportion; i += 2;
          continue;
        }
      case 'c': case 'C':
        { istringstream buf(argv[i+1]);
          buf >> cible; i += 2;
          continue;
        }
```

```
      case 'h': case 'H': case ‘?’:      // on imprime la syntaxe d’appel
       cerr << "Usage: " << arvg[0]
            << " -d entier -p entier -c entier\n";
       exit(1);                          // ou return 1;
      default:                           // on imprime le caractère non reconnu
       cerr << "Option inconnue: -" << argv[i][1] << endl;
       exit(1);
      }
    } else { break; }                    // commence par autre chose que '-'
  }
  if (i < argc) { cerr << "Ignoring extra arguments! \n"; }

  // Partie 2: Ici commence vraiment la partie intéressante du programme...
}
```

Le type `istringstream` permet de transformer une chaîne de caractères en flot de lecture : on peut alors lui appliquer l'opérateur `>>` pour lire un élément à partir de cette chaîne plutôt qu'à partir de `cin` ou d'un fichier. Pour que ce type et les fonctions associées soient connus il faut ajouter en tête du programme la directive :

```
#include <sstream>
```