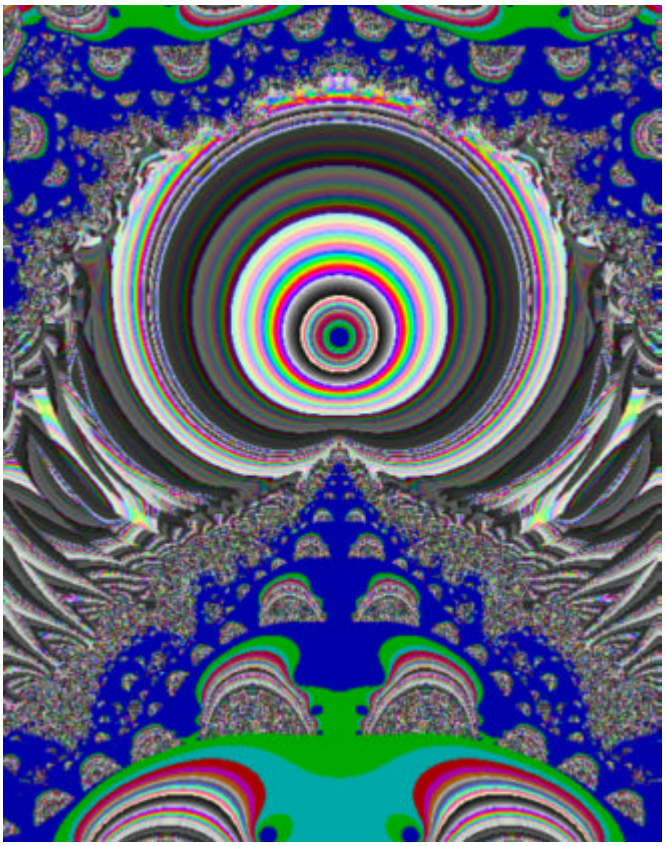
CHAPITRE 7

Types constants, Références



C++ introduit deux notions importantes par rapport au C traditionnel connu pour les compilateurs de type K&R. Le type `const`, déjà connu en C ANSI, et la référence. La référence est une notion très importante, qui permet de remplacer bien des utilisations de pointeurs potentiellement dangereuse par un **alias** (autre identification) d'un identificateur donné.

7.1 *Types constants*

7.1.1 *Réquisitoire contre #`define`*

Il est malaisé, en K&R C tout au moins, de définir des constantes. Utiliser des variables en guise de constante n'est pas une solution, puisque par définition, n'importe qui pourra les modifier. Utiliser la macro `#define` du préprocesseur n'est pas non plus une solution satisfaisante, car le préprocesseur se bornera à remplacer chaque occurence de l'expression suivant #define par l'expression suivante sur la ligne. Cette manière de faire peut conduire à des messages d'erreur très obscurs de la part du compilateur, et à une recherche de faute ardue. Prenons l'exemple suivant :

Soit un groupe de développement dont vous faites partie, et dont vous utilisez les résultats, comme vos collègues utilisent les vôtres d'ailleurs. Vous définissez un module de programmation qui comprend l'instruction suivante :

```
#include "file1.h"
```

file1.h est un de vos fichiers en-tête, que vous utilisez pour définir vos propres types. Il contient la définition d'un type que vous utilisez pour réfléter l'état de vos modules de programme. Plutôt que d'utiliser un int directement, vous avez préféré définir un type intermédiaire (louable intention):

```
typedef int   Status;
```

Vous utilisez un autre fichier, qui lui n'est pas défini par vos soins, mais par un de vos collègues, pour accéder à du code qu'il a écrit et qui satisfait vos besoins.

```
#include "file2.h"
```

Votre collègue ignore ce que vous avez défini, et définit de son côté un état (Status) fixe, égal à 10. Il utilise pour ce faire le préprocesseur :

```
#define Status 10
```

Soit vous n'avez pas examiné en détail son fichier `file2.h`, soit il a été modifié depuis que vous l'aviez examiné: cette ligne (ou ses implications) vous a en tous cas échappé. Dans la suite de votre programme, vous avez défini la ligne suivante :

```
Status   *uneFonctionQuelconque(int sonParamètre) { };
```

En arrivant sur cette ligne, le compilateur va vous signaler une erreur (vraisemblable-

ment quelque chose comme "`syntax error`", le message d'erreur le plus frustrant que puisse fournir un compilateur de langage) sur la déclaration de votre fonction. Lorsque vous allez tenter de découvrir l'erreur, vous n'en découvrirez pas, parceque l'erreur n'est pas dans le code source, mais uniquement dans un code que vous ne regardez jamais et qui normalement n'est pas disponible, à savoir le code généré par le préprocesseur qui a produit, en lieu et place de votre déclaration de fonction :

```
10   *uneFonctionQuelconque(int sonParamètre) { };
```

ce qui conduit au message d'erreur :

```
mainprog.C, line XXX : error : syntax error //HPC++
```

On remarque que le compilateur signale l'erreur dans le programme principal. La ligne qu'il signale a de grandes chances de se trouver très loin de la cause effective de l'erreur, et cette cause même n'est pas signalée de manière interprétable. Vous pouvez bien sûr demander un listing intermédiaire du code produit avant compilation, mais ces listings sont souvent très longs, et vous préférerez sans doute recourir à votre instinct de programmeur.

Pour localiser cette erreur, vous allez devoir examiner à la main tous les fichiers que vous incluez dans votre code (en admettant que vous ayez immédiatement le réflexe de chercher dans ce sens, ce qui n'est pas forcément évident). Si au lieu de `#define`, votre collègue avait utilisé une constante, par exemple :

```
const     int       Status = 10;
```

Le compilateur aurait remarqué que Status est redéfini par file2.h, et aurait immédiatement protesté, en indiquant, par exemple (HPC++)

```
file2.h, line YYY : error, Status declared as identifier and typedef
```

Ce qui localise l'erreur immédiatement, bien sûr.

Cet example peut vous paraître artificiel. Il ne l'est pas: en fait il est tiré de plusieurs expériences personnelles, qui étaient peut-être un peu moins triviales quand même.

7.1.2 *Utilisation de `const`*

L'utilisation de constantes ne se limite pas à ce genre de cas. Il faut utiliser des constantes chaque fois que l'on désire manipuler des données effectivement constantes. Le compilateur vérifie que vous ne modifiez pas la valeur d'une constante et vous signale une éventuelle violation de cette règle, comme en PASCAL. Il en va de même si vous désirez que l'on ne modifie pas le résultat d'une fonction : déclarez son type de retour constant, comme dans :

```
const     int*      uneFonctionQuelconque()

     {
     static    int       aValue = 10;
     return &aValue;
     }

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

     {
     int       *pointeurEntier = uneFonctionQuelconque();
     }
```

```
CC: "myprog.C", line XX : error : bad initializer type const int* for poin-
teurEntier (int* expected) // HPC++
```

Vouloir assigner un pointeur sur une constante à un pointeur sur une valeur non-constante permettrait par la suite de modifier la valeur que l'on désire conserver constante, d'où la réclamation du compilateur. Pour se conformer à la règle de permissivité de C, vous avez néanmoins la possibilité de contourner cette erreur par un casting explicite (coercion de types). Il est possible, dans la ligne ayant causé l'erreur, d'ajouter l'opérateur de casting de C pour obtenir :

```
     int       *pointeurEntier = (int*) uneFonctionQuelconque();
```

Le programme compile dorénavant sans problèmes. Mais la responsabilité d'une erreur potentielle dûe à cette coercion explicite repose désormais entièrement sur le programmeur. A partir du moment où le programmeur recourt à des conversions de type en C ou en C++, il court-circuite tous les mécanismes de contrôle qu'avait défini le langage. C'est le fameux échappatoire, absent de PASCAL, que Brian Kernighan considérait indispensable.

Un type constant occupe une certaine place en mémoire: il est donc possible d'en calculer l'adresse et d'assigner cette adresse à un pointeur sur un constant de type adéquat. Avec la conversion décrite çi-dessus, on peut également assigner cette adresse à un pointeur sur un non-constant quelconque. On peut donc, techniquement, modifier n'importe quel identificateur, bien qu'on l'ait déclaré constant. Se servir de ce genre de possibilité qu'offre C est une grave erreur, même si cela semble, sous le coup de contingences de délais de livraison d'une correction d'erreur, une méthode rapide pour arriver à son but. En principe, il faut s'en tenir aux règles d'utilisation suivantes lorsque l'on définit des expressions constantes, en particulier lorsque l'on utilise des pointeurs ou des références (dans le cas de passages par valeur, il n'y généralement pas de précautions spéciales à prendre) :

Déclarer const tout ce qui l'est effectivement.

Dans les en-tête de procédure, signaler les valeurs que la procédure ne modifie pas en les déclarant const. Une valeur non constante peut être sans problème passée à une procédure

qui attend une valeur de type const. Dans le cas inverse, ce n'est pas vrai lorsqu'il s'agit de pointeurs. Ainsi, le code suivant produit une erreur :

```
#include    <iostream.h>

int     aProc(int *aVal)

    {
    return 10*(*aVal);
    }

int     main(int , char*)

    {
    const int aVal = 10;
    cout<<aProc(&aVal)<<endl;
    }
CC: myProg.C, line 13 : bad argument type 1 for aProc() : const int* (int*
expected) // HPC++
```

alors que, en revanche :

```
#include    <iostream.h>

int     aProc(const int *aVal)

    {
    return 10*(*aVal);
    }

int     main(int , char*)

    {
    int aVal = 10;
    cout<<aProc(&aVal)<<endl;
    }
```

compile sans erreur.

Plutôt que de caster un pointeur sur une constante pour le passer à une procédure externe qui attend un pointeur sur une valeur non constante, copier la valeur de la constante dans une variable auxiliaire, et passer un pointeur sur cette dernière à la procédure en question. Cette dernière règle est malheureusement la moins respectée, en raison des librairies C existantes, qui ne définissent pas de paramètres de type const dans les prototypes qu'elles exportent, ce qui nécéssiterait, en respectant strictement cette règle, un nombre très élevé de copies, et éventuellement une baisse de performances du programme.

Dans le cas de pointeurs, on peut définir un pointeur sur une valeur constante, un pointeur constant sur une valeur non constante, ou un pointeur constant sur une valeur constante. La notation, dans chaque cas, est la suivante :

```
char*     p             = "Coucou";
                                  //   Pointeur non constant
                                  //   Valeur non constante
const     char*     p     = "Coucou";
                                  //   Pointeur non constant
                                  //   Valeur constante
char*     const     p     = "Coucou";
                                  //   Pointeur constant
                                  //   Valeur non constante
cont      char*     const     p     = "Coucou";
                                  //   Pointeur constant
                                  //   Valeur constante
```

La meilleure façon de mémoriser ces règles est de représenter ces règles de la manière suivante :

```
Valeur sur laquelle on pointe est constante         Pointeur constant
char                              *                      p = "Coucou"
const         char                *                      p = "Coucou"
char                              *             const    p = "Coucou"
const         char                *             const    p = "Coucou"
```

7.2 *Types Référence*

Le type référence vient combler une des plus grosses lacunes de C. Pour passer l'adresse d'un objet en C classique, on passe un pointeur sur cet objet, alors qu'en PASCAL par exemple, on passe plus communément une référence à cet objet (VAR chose : integer), et on réserve les pointeurs à des utilisations moins courantes, comme des listes chaînées, par exemple. Physiquement, il n'y a que peu de différence entre un pointeur et une référence. Du point de vue informatique, ce sont des choses fondamentalement différentes. Un pointeur contient l'adresse de l'objet sur lequel on le fait pointer à un moment donné. On peut à tout moment, décider de le faire pointer sur autre chose, donc le déplacer.

```
int       val = 10;
int       &refVal = val;     // Correct référence un objet existant
          refVal++;          // Correct, val = 11.
int       &otherRef;         // Faux, référence non-initialisée
int       *intPtr = &val;    // Correct, pointeur sur val
int       *intPtr2 = &refVal;// Correct, également pointeur sur val.
          (*intPtr)++;       // Correct, val = 11
          intPtr++;          // Correct du point de vue langage,
                             // pointe sur l'entier après val.
          (*intPtr)++;       // Probablement faux lors de l'execution.
```

Par contraste, la référence à un objet ne peut être déplacée: elle est intimement liée à l'objet lui-même. Il ne s'agit pas d'une variable contenant une adresse, mais de l'adresse elle-même. On ne peut pas déclarer un type référence sans immédiatement l'initialiser, car une référence non initialisée n'a véritablement pas de sens. Une référence est en ce sens de beaucoup plus sûre qu'un pointeur, car on court peu de risques qu'elle ne réfère pas un objet existant. Mais attention: il n'y a pas de sécurité absolue en C ou en C++, non plus qu'en d'autres langages d'ailleurs. Essayez de comprendre pourquoi l'exemple suivant vous conduira presque certainement à une erreur fatale à l'éxecution:

```
int&      aProcedure(int aValue)// HPC++

     {
     int       retValue = aValue++;
     return retValue;
     }

void      main(int, char[])

     {
     int       anInt& = aProcedure(1);
     anInt++;
     }
```

Certains compilateurs (en particulier ceux qui s'en tiennent strictement à Cfront 3.5) signalent cette erreur sous la forme d'un avertissement (*warning*) que le programmeur peut choisir d'ignorer.

On entend souvent des programmeurs dire que la référence n'est en fait qu'une amélioration sémantique, et n'apporte rien au niveau performances ou facilités par rapport à une implémentation avec des pointeurs. Rien n'est plus faux: il est plus facile de travailler avec des références qu'avec des pointeurs: vous pouvez vous en rendre compte vous-même en examinant les définitions de iostream, par exemple (classe ostream, par exemple). La notation utilisée pour travailler avec des références est plus simple, puisque le fait qu'il s'agit d'une référence est dissimulé dans la déclaration de l'en-tête de la procédure ou dans la déclaration de la variable, alors que l'utilisation de pointeurs vous condamne à rappeler cette implémentation à chaque utilisation d'un élément adressé par le pointeur. D'autre part, du fait que la référence est fixe, il est plus aisé pour un compilateur d'optimiser les accès aux éléments d'une structure passée par référence plutôt que par pointeurs, car il peut s'appuyer sur le fait que la référence ne change pas, même si elle est passée à des éléments externes au programme, donc non contrôlables au cours du passage d'optimisation du compilateur.

7.3 *Test*

6. Dans le code suivant, remplacer le pointeur par une référence :

```
int unEntier = 10;
int* ptrEntier;
ptrEntier = &unEntier;
*ptrEntier++;
```

7. Le code suivant est-il correct ?

```
const int* xPtr;
int xx = 1;
int& refXX;
refXX = xx;
xPtr = &refXX;
```

si vous estimez ce code incorrect, suggérez une correction.

8. Dans le code suivant, remplacer les pointeurs par des références, en corrigeant les éventuelles erreurs commises par l'auteur :

```
int       add(const int* x, int* y)
     {
     return (*x + *y);
     }

int main()
     {
     int a = 1;
     int b = 2;
     cout<<"somme de "<<a<<"+"<<b<<"="<<add(&a, &b)<<endl;
     }
```

9. Peut-on remplacer les pointeurs par des références dans la fonction suivante ?

```
int  stringLength(char* theString)
     {
     int lgt = 0;
     if (theString != NULL)
          while (*theString++) lgt++;
     return lgt;
     }
```