*C*entre *I*nformatique pour les *L*ettres
et les *S*ciences *H*umaines

C++ : Leçon 12
Références

Nous connaissons déjà deux façons d'accéder à une donnée représentée en mémoire : il est possible d'utiliser son nom (s'il s'agit d'une variable ou d'une constante) ou de déréférencer un pointeur contenant son adresse. Le langage C++ introduit une troisième méthode, l'usage de *références*, qui s'avère absolument indispensable dans certains cas et fournit, de plus, une alternative à l'usage d'un pointeur en tant que paramètre pour certaines fonctions.

1 - Notion de référence

En C++, une référence est un moyen d'associer un nouveau nom à un objet existant déjà.

Il ne s'agit donc **pas** d'un moyen permettant de créer un nouvel objet, même si, comme nous allons le voir, la syntaxe permettant de créer une référence conduit à une expression ressemblant étrangement à la définition d'une variable initialisée.

Cette attribution d'un nouveau nom ne prive pas l'objet concerné de son nom initial, s'il en avait un : le nouveau nom devient simplement un synonyme de l'ancien, c'est à dire qu'ils désignent tous deux le même objet.

Définir une référence

La syntaxe utilisée est calquée sur la définition d'une variable et fait (malheureusement) appel à un symbole que nous connaissons déjà, puisque c'est celui utilisé par ailleurs pour l'opérateur "adresse de" : `&`. Une ligne définissant une référence commence par le type de l'objet qui sera désigné par la référence, suivit du signe `&` et du nouveau nom. L'objet auquel est attribué ce nouveau nom est ensuite désigné comme s'il s'agissait d'une valeur utilisée pour une initialisation.

```
1 int unEntier = 4;                  //définition et initialisation d'une variable
2 int & aliasDunEntier = unEntier;//création d'un synonyme du nom unEntier
3 aliasDunEntier = 12;               //unEntier vaut maintenant 12
```

Une fois défini, le nom d'une référence désigne l'objet spécifié lors de l'initialisation. Cette spécification est donc **obligatoire** (on ne peut pas définir une référence sans l'initialiser) et **irréversible** (l'objet désigné par une référence ne changera jamais).

Lorsque le type de l'objet qui sera désigné par la référence est "pointeur sur...", l'application de la syntaxe énoncée ci-dessus conduit à une expression qui peut paraître étrange :

```
int unEntier = 4;
int * unPointeur = & unEntier; //ce & désigne l'opérateur "adresse de"
int * & aliasDunPointeur = unPointeur;
```

L'apparition consécutive des caractères `*` et `&` n'a rien ici de paradoxal, car ils n'ont pas dans ce cas les significations contradictoires qu'auraient l'opérateur de déréférencement et l'opérateur "adresse de". Intervertir ces deux caractères conduit à tenter de définir un objet de type "pointeur sur référence", ce que le langage interdit.

Les références ne sont pas des pointeurs

Lorsqu'une variable fait l'objet d'une référence, celle-ci constitue un moyen permettant d'accéder à la variable sans utiliser son nom. Cet "accès alternatif" rappelle un peu celui rendu possible par l'existence d'un pointeur contenant l'adresse de la variable.

```
1 int uneVariable = 4;
2 int * unPointeur = & uneVariable;
3 int &uneReference = uneVariable;

  //On peut maintenant accéder à uneVariable en :
4 uneVariable = 5;   //utilisant son nom
5 *unPointeur = 6;   //déréférençant un pointeur contenant son adresse
6 uneReference = 7;  //utilisant une référence qui lui a été associée
```

Cette similarité d'effet ne doit pas masquer une profonde différence de nature : un pointeur est un objet bien différent de celui dont il contient l'adresse, alors qu'une référence n'est qu'un moyen d'accéder à l'objet auquel elle est associée, et n'a aucune existence autonome. Cette différence de nature se traduit par deux conséquences pratiques importantes :

- La définition d'un pointeur est une création de variable, qui a un prix en termes d'usage de la mémoire pendant l'exécution du programme, alors que la définition d'une référence ne crée qu'un identificateur reconnu par le compilateur (et, donc, utilisable dans le texte source).
- Un pointeur étant une variable, il est possible d'en modifier le contenu, et le même pointeur peut permettre d'accéder successivement à des variables différentes. L'association entre une référence et l'objet qu'elle désigne est, en revanche, fixée définitivement lors de l'initialisation de celle-ci.

On pourrait penser que l'obligation d'initialisation et l'impossibilité de modification ultérieure confèrent aux références un avantage important en garantissant leur validité permanente. Il n'en est cependant pas ainsi, et l'exemple suivant produit une référence invalide :

```
int * ptr = new int;
int & reference = *ptr;
delete ptr;
reference = 4; //HORREUR : cette reference n'est plus valide !
```

2 - A quoi servent les références ?

Quel intérêt peut-il bien y avoir à disposer de plusieurs moyens pour désigner la même chose ?

Dans le cas des pointeurs, un des arguments est que leur valeur peut changer et que, dans le cas d'une boucle, par exemple, la même ligne de code peut opérer sur des variables différentes, ce qui serait impossible si on ne pouvait accéder à celles-ci qu'en mentionnant leur nom. Cet argument ne tient pas dans le cas des références, puisqu'elles ne peuvent en aucun cas se mettre à désigner un autre objet que celui qui leur a été associé lors de leur création.

Les références comme paramètres

L'usage des pointeurs possède un autre avantage : lorsqu'une fonction dispose d'un paramètre de type "pointeur sur", on peut lui transmettre l'adresse d'une variable pour initialiser ce paramètre. Si la fonction déréférence le pointeur en question, elle accède alors à la zone de mémoire attribuée à la variable et se trouve donc en mesure de modifier le contenu de celle-ci. Le même phénomène se produit lorsqu'une fonction dispose d'un paramètre de type référence : si on lui transmet une variable pour initialiser le paramètre en question, le nom de celui-ci devient un synonyme de celui de la variable, avec toutefois une différence importante : la fonction peut utiliser le nom de son paramètre, alors qu'elle ne peut pas utiliser celui d'une variable qui ne lui appartient pas[1]. Si une fonction est définie ainsi :

```
1 void fonction(int & parametre) //un parametre de type référence
2 {
3 parametre = 4;
4 }
```

elle peut être utilisée comme ceci :

```
1 int k = 6;
2 fonction(k);
  //k contient maintenant 4
```

L'effet obtenu est donc très analogue à celui d'un paramètre de type pointeur, mais la notation est plus légère, puisque la fonction accède à la variable visée sans avoir à déréférencer un pointeur et peut être appelée sans avoir à utiliser l'opérateur "adresse de". Le code équivalent à l'exemple précédent serait en effet :

```
1 void autreFonction(int * parametre) //un parametre de type pointeur
2 {
3 *parametre = 4;
4 }
```

et

```
1 int k = 6;
2 autreFonction(&k);
3 //k contient maintenant 4
```

[1] Et qui n'est ni globale (bheurk...) ni membre de la même classe que la fonction.

Les fonctions qui ont un paramètre de type pointeur disposent parfois d'une valeur par défaut permettant à l'appelant de ne pas s'inquiéter de ce paramètre lorsqu'il n'est pas pertinent. Dans la plupart des cas, cette valeur par défaut est la constante NULL, et la fonction comporte alors des fragments de code du genre :

```
if (leParametre != NULL)
   * leParametre = resultat;
```

Un effet analogue peut être obtenu avec un paramètre de type référence, dont la valeur par défaut peut être fixée en déréférençant explicitement un pointeur NULL obtenu par transtypage d'une constante. Dans le cas d'une référence à un int, nous écrirons donc par exemple :

```
void fonction (int & leParametre = *static_cast<int *>(NULL));
```

Le corps de la fonction peut ensuite comporter des fragments de code du genre :

```
if (&leParametre != NULL) //attention : ce & est l'opérateur "adresse de" !
   leParametre = resultat;
```

Il va sans dire que la construction délibérée d'une référence désignant la zone de mémoire d'adresse 0 implique les mêmes précautions que la présence d'un pointeur NULL : il ne faut en aucun cas essayer de modifier le contenu de cette zone de mémoire !

Si l'utilisation d'un paramètre de type référence plutôt que de type pointeur se traduit en général par des expressions plus simples, elle présente toutefois un inconvénient du point de vue de la lisibilité du code : l'absence de toute notation particulière lors de l'appel de la fonction ne permet pas de distinguer que c'est la variable qui est transmise (et non simplement sa valeur), et qu'elle est donc susceptible d'être modifiée par l'exécution de la fonction appelée. Certains programmeurs préfèrent donc réserver, dans la mesure du possible, les paramètres de type référence aux cas où il s'agit de références à des objets constants.

Référence à un objet constant

Lorsqu'une fonction dispose d'un paramètre de type référence, elle se trouve en mesure de modifier l'objet qui lui est transmis. Lorsque cette possibilité de modification n'est pas souhaitable, il est possible de préciser que l'objet référencé est constant, c'est à dire que toute modification en est interdite. La logique de l'opération et la syntaxe utilisée sont tout à fait analogues à celles rencontrées précédemment lors de la définition de pointeurs sur objets constants (Leçon 11) :

```
1 void fonction(const int &parametre)
2 {
3 parametre = 2; //ERREUR : cet objet est constant !
4 }
```

L'idée même d'une référence à un objet constant peut sembler paradoxale. En effet, si la fonction ne doit pas modifier l'objet qui lui est transmis, pourquoi utiliser un paramètre de type référence ? Le mécanisme de base du passage de paramètre assurerait une simple initialisation du paramètre avec la valeur de l'objet, et garantirait donc parfaitement l'intégrité de celui-ci.

La raison d'être des références à des objets constants est précisément qu'elles permettent d'obtenir exactement le même effet que le passage "ordinaire" de paramètres (communication de la valeur d'un objet, avec impossibilité de modifier celui-ci), *sans avoir à effectuer de copie de cet objet* lors de l'initialisation du paramètre de la fonction. Il existe au moins deux bonnes raisons d'éviter de faire une copie de l'objet transmis :

- Lorsque l'objet en question est très volumineux, l'opération de copie peut s'avérer coûteuse en durée d'exécution et en place mémoire.
- L'opération de copie peut, dans certains cas, être totalement inenvisageable. Nous rencontrerons une situation de ce genre dans la Leçon 13, lorsqu'il s'agira précisément de définir la fonction qui assure l'initialisation d'une instance d'une classe à l'aide des valeurs contenues dans une autre instance de la même classe. Cette fonction devra évidemment avoir accès à l'objet servant de modèle, mais il ne lui sera pas possible d'initialiser un de ses paramètres à l'aide de ce modèle, puisque cette opération de recopie est justement celle que nous chercherons à rendre possible en définissant la fonction.

Fonctions renvoyant une référence

Lorsqu'une fonction renvoie une référence, la fonction appelante peut utiliser l'expression provoquant l'appel comme une lvalue. En d'autres termes, l'appel d'une fonction renvoyant une référence peut figurer *à gauche* d'un opérateur d'affectation. La fonction appelée se trouve donc en position de décider quelle sera la variable qui subira l'affectation effectuée par la fonction appelante. Imaginons par exemple une fonction acceptant comme paramètres deux références et renvoyant l'une d'entre elles, en fonction d'un tirage au hasard (voir à ce propos l'Annexe 6).

```
1  int & uneDesVariables(int &uneVar, int & uneAutre)
2  {
3  if (rand() % 2 == 1)
4    return uneVar;       //lorsque le nombre tiré est impair
5  else
6    return uneAutre;     //lorsque le nombre tiré est pair
7  }
```

Une telle fonction pourra être utilisée pour laisser le hasard décider laquelle de deux variables doit être remise à zéro :

```
1  int a = 2;
2  int b = 3;
3  uneDesVariables(a,b) = 0; //on ne sait pas laquelle !
```

Il est, bien entendu, tout à fait possible d'obtenir un résultat équivalent à l'aide d'une fonction renvoyant un pointeur et disposant de deux paramètres de type pointeur. Elle sera alors définie ainsi :

```
int * uneDesVariables(int * unPtr, int * unAutre)
{
if (rand() % 2 == 1)
   return unPtr;     //lorsque le nombre tiré est impair
else
   return unAutre;   //lorsque le nombre tiré est pair
}
```

L'utilisation de cette fonction nécessite la transmission des adresses des variables concernées, et le déréférencement de l'adresse renvoyée :

```
int a = 2;
int b = 3;
*uneDesVariables(&a, &b) = 0; //on ne sait pas laquelle !
```

Bien que l'intérêt de cette technique puisse sembler a priori assez marginal, nous rencontrerons dès la Leçon prochaine des situations où le renvoi d'une référence s'avère indispensable.

3 - Le point sur l'art et les manières de passer un paramètre

Maintenant que nous disposons des notions de constance et de référence, nous sommes en mesure de dresser un panorama complet des différents cas de figures qui peuvent se présenter lorsqu'une fonction appelante transmet un paramètre à une autre fonction.

Les exemples ci-dessous illustrent la transmission de paramètres entiers. Ils restent évidemment valides si un type quelconque (y compris un type de votre propre cru) est substitué à `int`.

L'objet à transmettre est un tableau (vrai ou faux)

Dans ce cas, la situation est simple : la fonction recevra nécessairement l'adresse du premier élément du tableau, ce qui ne lui laisse que deux options :

- Soit elle utilise cette adresse pour initialiser un paramètre de type pointeur, et elle aura accès au tableau aussi bien pour en consulter le contenu que pour modifier celui-ci.
- Soit elle utilise cette adresse pour initialiser un paramètre de type pointeur sur objet constant, et elle n'aura accès au tableau que pour en consulter le contenu.

Ces deux options ne se distinguent que par la présence du mot `const` dans la déclaration du paramètre de la seconde fonction :

```
1 void exemple1(int * parametre);
2 void exemple2(const int * parametre);
```

Que le paramètre soit de type pointeur ou de type pointeur sur objet constant, il s'agit d'un pointeur que la fonction doit nécessairement déréférencer pour accéder à l'objet qui lui a été transmis. Si la fonction a, par exemple, besoin d'initialiser une variable locale avec la valeur contenue dans le premier élément du tableau, on écrira :

```
1 int uneVar = parametre[0]; //déréférencement à l'aide de la notation indexée
2 int uneAutre = *parametre; //déréférencement à l'aide de l'opérateur "étoile"
```

> Rappel : la fonction ne dispose a priori d'aucun moyen lui permettant de connaître le nombre d'éléments du tableau. Dans un programme réel, il vous appartient de choisir et de mettre en place un des dispositifs permettant de contourner ce problème (paramètre supplémentaire, utilisation du premier élément pour stocker la taille du tableau, valeur sentinelle en fin de tableau, etc).

Dans tous les cas, la transmission du paramètre lors de l'appel de la fonction s'effectue en mentionnant simplement le nom du tableau :

```
1 int vraiTableau[100];
2 fonction(vraiTableau);
3 int * fauxTableau = new int[100];
4 if (fauxTableau != NULL)
5   fonction(fauxTableau);
```

La fonction n'a pas besoin de modifier l'objet transmis (qui n'est pas un tableau)

Dans ce cas, il *faut* faire en sorte que la fonction ne *puisse* pas modifier l'objet transmis, et il existe trois méthodes permettant d'obtenir ce résultat.

a) Transmission de la valeur de l'objet

Il s'agit là de la méthode la plus simple, puisqu'elle n'implique aucune notation particulière. La fonction dispose d'un paramètre d'un type compatible avec celui de l'objet mentionné lors de l'appel, et ce paramètre est initialisé avec la valeur de l'objet en question. La déclaration d'une telle fonction sera donc :

```
void fonction(int parametre);
```

La fonction est parfaitement libre de faire ce qu'elle veut avec son paramètre, dont un éventuel changement de valeur reste sans effet sur l'état de l'objet dont la valeur à servi à l'initialiser.

Cette méthode possède une caractéristique importante : elle rend la fonction capable d'accepter une constante littérale, et nous pouvons écrire

```
fonction(36);
```

aussi bien que

```
1 int var = 12;
2 fonction(var);
```

> Bien entendu, comme la fonction n'attend pas une adresse pour initialiser son paramètre, l'utilisation d'un objet auquel on accède via un pointeur contenant son adresse implique de déréférencer ce pointeur lors de l'appel, exactement comme on doit le déréférencer pour affecter une valeur à l'objet :
>
> ```
> int * ptr = new int;
> if(ptr != NULL)
>    {
>    *ptr = 12;
>    fonction(*ptr);
>    }
> ```
>
> Cette opération de déréférencement n'a, en fait, rien à voir avec la question du passage d'un paramètre à la fonction, mais découle directement de la façon dont la fonction appelante a créé l'objet concerné.

La transmission de la valeur de l'objet offre l'avantage de la simplicité d'écriture (les symboles * et & ne sont nécessaires ni dans la déclaration de la fonction, ni dans sa définition, ni dans son appel). L'inconvénient majeur de ce mode de transmission est qu'il implique, à chaque appel de la fonction, la création d'une variable (le paramètre) et son initialisation. Lorsque le paramètre est d'un type volumineux (pensez, par exemple, à une classe complexe), cette copie coûte de la place en mémoire et du temps d'exécution.

b) Transmission d'un pointeur sur un objet constant

Il est également possible que la fonction reçoive l'adresse de l'objet dont on souhaite lui communiquer la valeur. Si cette adresse est stockée dans un "pointeur sur objet constant", l'intégrité de l'objet sera garantie, quels que soient les agissements de la fonction. La déclaration de la fonction adopte alors la forme suivante :

```
void fonction(const int * parametre);
```

Bien entendu, puisque la fonction ne dispose que d'un pointeur contenant l'adresse de l'objet dont la valeur l'intéresse, elle doit déréférencer ce pointeur pour prendre connaissance de cette valeur. Une tentative de déréférencement du pointeur dans le contexte d'une instruction qui modifierait la valeur de l'objet pointé conduira, pour sa part, à une erreur lors de la compilation de la fonction.

Lors de l'appel, le fait que cette fonction attende une adresse impose l'usage de l'opérateur "adresse de" lorsque c'est la valeur d'une variable qui doit être communiquée :

```
1 int var = 12;
2 fonction (&var)
```

Par ailleurs, la nécessité de transmettre une adresse empêche évidemment la fonction de recevoir une constante littérale (une constante littérale n'est pas représentée dans la zone de mémoire stockant les données du programme, et ne peut donc pas être désignée par son adresse). Il est en revanche possible d'utiliser une constante symbolique, car le langage C++ crée dans ce cas une véritable variable (qui a donc une adresse) dont il interdit ensuite la modification (ce qui lui donne son caractère constant)[2]. On peut donc écrire :

```
1 const int DOUZE = 12;
2 fonction(& DOUZE);//OK : DOUZE est une "variable constante" qui a une adresse
```

alors qu'il est impossible d'écrire

```
fonction(& 12); //ERREUR : l'opérateur "adresse de" ne peut pas être appliqué
                //à une constante littérale
```

Bien entendu, si la valeur à transmettre est contenue dans un objet auquel on accède en déréférençant un pointeur, le fait que la fonction attende une adresse nous dispense de déréférencer le pointeur lors de l'appel :

```
int * ptr = new int;
if(ptr != NULL)
   {
   *ptr = 12;
   fonction(ptr);
   }
```

L'utilisation d'un paramètre de type "pointeur sur un objet constant" présente l'inconvénient d'exiger une bonne maîtrise de l'utilisation des symboles * et &, car la déclaration de la fonction utilise nécessairement l'étoile dans sa signification "déclaration d'un pointeur", la définition de la fonction utilise (probablement) l'étoile dans sa signification "opérateur de déréférencement", et l'appel de la fonction fait (le plus souvent) intervenir le symbole & dans sa signification "adresse de". L'avantage de cette approche est qu'elle permet de communiquer une valeur à la fonction, sans que chaque appel de celle-ci implique une duplication de la représentation de cette valeur en mémoire.

[2] Cette faculté qu'a le langage de décider de façon optimale si les constantes symboliques doivent ou non être effectivement représentées dans les données du programme est une raison supplémentaire pour éviter d'utiliser des constantes littérales.

c) Transmission d'une référence désignant un objet constant

Comme nous l'avons vu précédemment, la déclaration de la fonction est, dans ce cas :

```
void fonction(const int & parametre);
```

L'écriture et l'appel de la fonction n'exigent aucune notation particulière, ce qui est à la fois un avantage et un inconvénient. Toute expression admissible lors de l'initialisation d'une référence peut être utilisée lors de l'appel d'une telle fonction, ce qui inclut bien entendu les noms de variables, mais aussi les pointeurs déréférencés :

```
1 int var = 0;
2 int * ptr = &var;
3 fonction(var);
4 fonction(*ptr);
```

L'utilisation d'une référence à un objet constant permet d'obtenir les avantages associés à l'utilisation d'un "pointeur sur objet constant" (rapidité et économie de mémoire) tout en évitant la lourdeur d'écriture impliquée par le recours à un pointeur : le symbole * n'apparaît ni dans la déclaration de la fonction, ni dans sa définition, ni lors de son appel, et le symbole `&` ne figure (dans son sens "déclaration d'une référence") que dans la déclaration de la fonction.

La fonction a besoin de modifier l'objet transmis (qui n'est pas un tableau)

Nous disposons, dans ce cas, de deux méthodes : le passage de l'adresse de l'objet et le passage d'une référence à l'objet. Déclaration, définition et appel de la fonction présentent alors exactement les mêmes caractéristiques que dans les cas d'un pointeur ou d'une référence désignant un objet constant (à l'exception, bien entendu, du mot `const`).

Le passage d'une référence simplifie l'écriture (la fonction appelée n'a pas à déréférencer un pointeur), mais l'usage d'un pointeur est plus lisible (l'auteur de la fonction appelante ne peut pas ignorer qu'il existe un risque que l'objet qu'il passe soit modifié).

Il existe un cas particulier où le fait de dispenser la fonction de l'obligation de déréférencer un pointeur lorsqu'elle doit modifier l'objet qui lui a été confié est un gros avantage pour les programmeurs débutants : c'est le cas où l'objet à modifier est lui-même un pointeur. Ne pas utiliser de référence dans cette situation conduit inéluctablement à doter la fonction d'un paramètre de type "*pointeur sur pointeur*", variable dont la manipulation nécessite une certaine habitude[3]. En effet, un tel paramètre doit être déréférencé deux fois lorsqu'on souhaite l'utiliser pour accéder à l'objet dont l'adresse est stockée dans le pointeur dont l'adresse a été transmise à la fonction. Il semble malheureusement que ce genre de raisonnement suscite trop rarement l'enthousiasme immédiat des étudiants, et ces situations (assez fréquentes dans les programmes mettant en place des structures de données dynamiques du genre liste) seront donc avantageusement traitées à l'aide de paramètres de type "référence à un pointeur", qui n'introduisent aucun niveau d'indirection supplémentaire.

4 - Bon, c'est gentil tout ça, mais ça fait quand même 7 pages. Qu'est-ce que je dois vraiment en retenir ?

1) La définition d'une référence permet d'attribuer un nouveau nom à un objet existant déjà.
2) La définition d'une référence ressemble à celle d'une variable dont le nom serait précédé du signe `&`, et qui serait initialisée à l'aide d'une autre variable du même type.
3) Une référence ne désignera jamais un autre objet que celui qui lui a été associé lors de son initialisation.
4) Les références sont surtout utiles en tant que types de paramètre et de valeur de retour pour les fonctions. Elles permettent alors d'obtenir des effets analogues à ceux obtenus à l'aide de pointeurs, sans les complications syntaxiques liées à la présence de ceux-ci.
5) Lorsqu'une fonction n'a besoin que de connaître le contenu d'un objet, il ne faut pas lui laisser la possibilité de le modifier. Si la seule transmission de la valeur de l'objet ne convient pas, il faut donc passer un pointeur ou une référence à un objet constant.

[3] Les pointeurs sur pointeurs apparaissent aussi assez fréquemment lors de la manipulation de tableaux multidimensionnels (cf. Annexe 3).