

Quand utiliser les références ?

Philippe PRADOS

pp@philippe.prados.name



*Préservez l'environnement,
n'imprimez pas ce document*

Avant propos

Ce document explique quand et où utiliser les références du C++.

Les anciens développeurs C préfèrent les pointeurs aux références car ils les maîtrisent déjà. Les références ont été ajoutées au langage pour pouvoir surcharger l'opérateur crochet. C'était le seul moyen de pouvoir simuler un tableau dans un objet. Cette notion n'est pas nouvelle, le langage Pascal par exemple, utilise abondamment cette notion. Un paramètre peut être passé par valeur ou par référence. Cela permet d'éviter d'utiliser la notion de pointeur qui n'est pas évidente à maîtriser rapidement.

Dans cet article, je vais traiter des différentes situations dans lesquelles une référence peut être utilisée :

- comme attribut,
- comme paramètre d'une fonction ou d'une méthode,
- comme retour d'une fonction ou d'une méthode.

1. ATTRIBUT

Peut-on utiliser une référence comme attribut d'une classe ? La réponse est « oui », mais cela est très pénalisant. En effet, il n'est pas possible après la construction d'une référence de la modifier pour référencer un autre objet.

```
int a=0,b=1;
int& rint=a;
rint=b; // a=b
```

L'affectation sur une référence modifie l'objet référencé, pas la référence elle-même. La construction et l'affectation ont des comportements différents pour les références. Cela entraîne qu'il n'est pas possible d'écrire un opérateur d'affectation sur une classe possédant un attribut en référence. L'opérateur par défaut n'est d'ailleurs pas généré par le compilateur pour les classes ayant un attribut en référence.

```
class CEmploye
{
  CEntreprise& _entreprise;
public:
  CEmploye(CEntreprise& x)
  : _entreprise(x) { }
};
```

Le compilateur ne génère pas l'opérateur d'affectation. Il ne faut donc pas utiliser les références comme attribut d'un objet.

2. PARAMETRES

L'utilisation d'une référence en paramètre est une question plus difficile à trancher. Il existe plusieurs sémantiques possibles pour un paramètre. Suivant les cas, les références sont à privilégier ou à bannir.

2.1 Consultation

Si le paramètre ne doit être que consulté par la méthode, il faut recevoir une référence. Si vous avez la tentation de recevoir un pointeur constant qui ne doit pas être adopté, utilisez une référence constante. En effet, cela permet à l'utilisateur de construire un objet temporaire lors de l'appel de la méthode.

```
void f(const CPersonne& personne);

void main()
{ f(CPersonne("Paul"));
}
```

Si la fonction `f()` recevait un pointeur, l'utilisateur devrait écrire :

```
void f(const CPersonne* personne);

void main()
{ CPersonne paul("Paul");
  f(&paul);
}
```

car une écriture comme `f(&CPersonne("paul"))` est refusée par le compilateur. De plus, le compilateur peut lui-même construire l'objet temporaire lors d'une écriture comme cela :

```
f("paul"); // Creation d'obj temporaire
```

Il est alors possible d'utiliser une conversion par construction. Un constructeur ne recevant qu'un seul paramètre est une conversion par construction. Par exemple, s'il existe une classe `CDieu` n'héritant pas de `CPersonne`, et que la classe `CPersonne` accepte un constructeur recevant comme paramètre un objet `CDieu` :

```
class CDieu
{ //...
};
class CPersonne
```

```
{ //...
  public:
    CPersonne();
    CPersonne(const CDieu& dieu);
};
```

La fonction `f()` peut directement recevoir un `CDieu` si elle attend un paramètre en référence.

```
void f(const CPersonne&);
void main()
{ CDieu zeus;
  f(zeus);
}
```

Si la fonction `f()` recevait un pointeur, il faudrait compliquer inutilement l'appel.

```
void f(const CPersonne*);
void main()
{ CDieu zeus;
  CPersonne tmp(zeus);
  f(&tmp);
}
```

Il est préférable de laisser le compilateur faire ce travail à votre place. Votre interface sera alors beaucoup plus riche. Sans le décrire directement, la fonction `f()` peut recevoir une `CPersonne` ou un `CDieu` ou tout objet ayant une conversion vers le type `CPersonne`.

2.2 NULL possible

Si le paramètre qu'une méthode reçoit peut ne pas être présent, c'est-à-dire que l'utilisateur peut envoyer la valeur `NULL`, il faut recevoir un pointeur. Il ne faut pas utiliser une référence artificielle sur l'adresse `NULL` ce qui permettrait à la méthode d'écrire

```
void f(const A& x)
{ if (&x==NULL) ... // A ne pas faire !
}
```

Il faut recevoir un pointeur à la place. Une meilleure approche est de surcharger les méthodes. Une version reçoit un objet, et une autre reçoit un pointeur qui doit toujours être `NULL`.

```
void f(const A& x)
{ ...
}
void f(const void* x)
{ assert(x==NULL);
  ...
}
```

2.3 Objet Null possible

Un objet `NULL` est une instance particulière de la classe permettant de signaler qu'un objet est erroné (Voir l'article "Comment gérer les erreurs dans les constructeurs en C++" du précédent numéro). Dans ce cas, le pointeur `NULL` n'a plus de raison d'exister. Il faut alors recevoir un paramètre du type référence si la méthode ne fait que consulter l'objet.

2.4 Adoption

Si le paramètre doit être adopté par la classe, il faut le recevoir en pointeur. Un paramètre est adopté par une classe, si celle-ci doit s'occuper de détruire l'objet en paramètre. La destruction s'effectuant par un `delete`, l'objet doit avoir été construit par un `new`. Une précondition peut d'ailleurs vérifier que le paramètre est bien alloué dans le tas à l'aide des outils de debug mémoire. L'utilisateur de l'objet manipule déjà un pointeur. Recevoir une référence l'obligerait à traduire ce pointeur en objet. De plus, l'objet receveur devrait écrire quelque chose comme `delete &x`, ce qui n'est pas très élégant, et entraînerait l'existence d'une référence invalide. Il faut donc recevoir un pointeur.

2.5 Tableau

Si le paramètre reçu est un tableau d'objet, il faut utiliser un pointeur. Les références ne permettent pas de manipuler un tableau. Recevoir un `const char*` équivaut à recevoir un tableau de caractères. Dans ce cas, il faut garder le pointeur.

```
void f(const char* p);
```

Une écriture plus claire serait par ailleurs

```
void f(const char p[]);
```

2.6 Modification

Si le paramètre doit être modifié par la méthode, il est possible de recevoir un pointeur ou une référence.

```
void f(CPersonne* personne)
{ personne->setNom("Philippe");
}
```

ou

```
void f(CPersonne& personne)
{  personne.setNom( "Philippe" );
}
```

Si le paramètre est reçu comme référence, l'utilisateur de la méthode peut être tenté de construire un objet temporaire. Cet objet serait modifié juste avant d'être détruit. Cela ne sert à rien. C'est d'ailleurs pour cela que le compilateur ne génère pas d'objet temporaire pour alimenter une référence non constante.

```
f("paul"); // ou
f(CPersonne("paul")); // warning ou erreur du compilateur
```

Il est donc préférable d'utiliser dans ce cas un pointeur. Cela signale à l'utilisateur que l'objet pointé sera modifié par la méthode et évite la construction hors norme d'un objet temporaire pour modification.

3. RETURN

3.1 Variable ou paramètre

Il n'est pas possible de retourner une référence ou un pointeur sur une variable ou un paramètre d'une méthode. Le compilateur le rejette.

3.2 Retourner une agrégation

Il faut bien identifier si l'attribut représente une agrégation ou une relation. Pour retourner un attribut, utilisez une référence constante. Un pointeur sur un attribut a une durée de vie qui n'est pas garantie. Si l'utilisateur de la méthode garde celui-ci alors que l'objet modifie l'emplacement de son attribut, le pointeur gardé par l'utilisateur deviendrait invalide.

```
class CEntreprise
{ CPersonne* _gerant;
public:
    CEntreprise(const CPersonne& gerant)
    : _gerant(new CPersonne(gerant))
    { }
    ~CEntreprise()
    { delete _gerant;
    }
    const CPersonne* getGerant() const
    { return _gerant; }
    void chgGerant(const CPersonne& gerant)
    { delete _gerant;
      _gerant=new CPersonne(gerant);
    }
};

void main()
{ CEntreprise WorldCompagnie(CPersonne( "paul" ));
  const CPersonne* gerant=WorldCompagnie.getGerant();
  WorldCompagnie.chgGerant(CPersonne("pierre"));
  // Le pointeur "gerant" n'est plus valide !
}
```

Si vous retournez une référence, l'utilisateur n'est pas tenté de garder le pointeur sur l'attribut.

```
class CEntreprise
{ CPersonne* _gerant;
public:
    CEntreprise(const CPersonne& gerant)
    : _gerant(new CPersonne(gerant))
    { }
    ~CEntreprise()
    { delete _gerant;
    }
    const CPersonne& getGerant() const
    { return *_gerant; }
    void chgGerant(const CPersonne& gerant)
    { delete _gerant;
      _gerant=new CPersonne(gerant);
    }
};
```

Manipuler une référence est plus difficile que de manipuler un pointeur. La durée de vie d'une référence est en général courte, car il n'est pas possible de changer la référence. Seul l'objet référencé peut être modifié. Ne pouvant pas être un attribut d'une classe, seule une fonction peut manipuler la référence. Le risque de pointeur erroné est alors réduit.

3.3 Retourner une agrégation avec NULL possible

Si un pointeur représente une agrégation et que ce pointeur peut posséder la valeur NULL, ce pointeur représente une agrégation optionnelle. Dans ce cas, il faut retourner un pointeur constant pour pouvoir transmettre l'information de l'absence de l'attribut. Il est préférable d'avoir un objet Null pour la classe de l'attribut.

3.4 Retourner une agrégation avec Objet Null possible

S'il existe une instance particulière indiquant que l'objet est en erreur ou vide, retournez alors une référence constante. Il n'est plus nécessaire de retourner un pointeur qui pourrait être gardé par erreur par l'appelant.

3.5 Retourner dans un conteneur

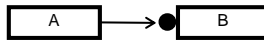
Un conteneur se contente de garder un ensemble d'objets indépendamment des objets eux-mêmes. L'accès à un des objets du conteneur doit permettre de modifier l'objet directement dans le conteneur. Dans ce cas, il faut retourner une référence non constante sur les objets contenus. Un objet de type « tableau » doit retourner une référence.

```
class CTableauA
{ A _tab[10];
public:
  A& operator[](int index)
  { return _tab[index]; }
};
```

Cela permet de modifier les objets A directement dans l'objet CTableauA. Même si les objets A sont des agrégations pour l'objet CTableauA, il faut offrir l'accès direct à ces attributs car le tableau ne fait que les regrouper, mais est indépendant des traitements sur les objets contenus.

3.6 Retourner une relation

Une relation indique par principe que plusieurs objets peuvent pointer sur un objet en relation. Si l'objet pointé doit être détruit dans le destructeur de la classe, ce serait une agrégation, donc un attribut. Pour un objet A en relation avec un objet B, l'objet A n'étant pas propriétaire de l'objet B, toute modification de l'objet B est valide.



De plus sa durée de vie n'est pas dépendante de l'objet A. Dans ce cas, il faut retourner un pointeur non constant. Modifier l'objet B n'entraîne pas, sémantiquement, la modification de l'objet A.

```
class A
{ B* _relaB;
public:
  B* getRelaB() const
  { return _relaB; }
};
```

3.7 Retourner this

Retourner `this` permet à l'utilisateur de la méthode d'enchaîner les appels. Il faut dans ce cas retourner une référence afin de pouvoir utiliser directement les opérateurs de la classe. Une écriture comme suit :

```
a=b=c;
```

n'est valide que si l'opérateur d'affectation retourne une référence sur lui-même.

```
A& A::operator =(const A& x)
{ //...
  return *this;
}
```

Une méthode constante doit retourner une référence constante, une méthode non-constante doit retourner une référence non constante.

4. RESUME

Pour conclure, voici un tableau récapitulatif des utilisations des références.

		Tableau	NULL Possible	NULL impossible
Déclarer un attribut	!const	*	*	*
	const	*	*	*
Recevoir un paramètre	!const	*	*	*
	const	*	* &	&
Retourner un attribut	!const	*	*	&
	const	*	*	&
Retourner une relation	!const	*	*	*
	const	*	*	*
Retourner un conteneur	!const	/	/	&
	const	/	/	&
Retourner this	!const	/	/	&
	const	/	/	&

/ : hors sujet * : pointeur & : référence

- Il est rare de recevoir une référence non `const` en paramètre.
- Un paramètre pointeur constant représente toujours un tableau.