Cours d'introduction à la programmation (en C++)
Pointeurs et références (en C++)

Jean-Cédric Chappelier
Jamila Sam
Vincent Lepetit

Faculté I&C

Les « pointeurs », à quoi ça sert ?

En programmation, les « pointeurs » servent essentiellement à trois choses :

① à permettre à plusieurs portions de code de *partager* des objets (données, fonctions,..) *sans les dupliquer*
☞ **référence**

② à pouvoir *choisir des éléments* non connus *a priori* (au moment de la programmation)
☞ **généricité**

③ à pouvoir manipuler des objets dont la *durée de vie* dépasse la portée
☞ **allocation dynamique**
(moins important depuis C++11 en raison de la *move semantic*)

Les « pointeurs », à quoi ça sert ?

En programmation, les « pointeurs » servent essentiellement à trois choses :

① **référence**

② **généricité**

③ **allocation dynamique**

☞ **Important :** Il faut toujours avoir clairement à l'esprit pour lequel de ces trois objectifs on utilise un pointeur dans un programme !

Les différents « pointeurs »

En C++, il existe plusieurs sortes de « pointeurs » :

- les **références**

  totalement gérées en interne par le compilateur. Très sûres, donc ; mais sont *fondamentalement différentes des vrais pointeurs*.

- C++11 les « pointeurs intelligents » (***smart pointers***)

  gérés par le programmeur, mais avec des gardes-fous.
  Il en existe 3 : `unique_ptr`, `shared_ptr`, `weak_ptr`
  (avec `#include <memory>`)

- les « **pointeurs « à la C »** » (*build-in pointers*)

  les plus puissants (peuvent tout faire) mais les plus « dangereux »

Les différents « pointeurs »

☞ lesquels utiliser ?

«*Utilisez des références quand vous pouvez, des pointeurs quand vous devez.*»

référence : références
(ou pointeurs « à la C »)

généricité : pointeurs « à la C »
ou index dans un tableau
(le tableau des choix/des possibles, s'il existe)

allocation dynamique : smart-pointers,
surtout `unique_ptr` dans un premier temps
(ou pointeurs « à la C »)

Le type « référence »

Une **référence** est un *autre nom* pour un objet existant, un *synonyme*, un *alias*

Une référence permet donc de désigner un objet indirectement

C'est exactement ce que l'on utilise lors d'un *passage par référence*

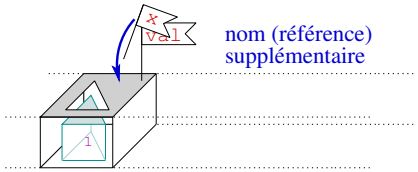
La déclaration d'une référence se fait selon la syntaxe suivante :

type`&` *nom_reference*`(`*identificateur*`);`

Après une telle déclaration, *nom_reference* peut être utilisé partout où *identificateur* peut l'être.

Exemple :

```
int  val(1);
int& x(val);
```



Attention aux pièges !

- signification de l'opérateur `=` :

```
int i(3);
int& j(i); // alias
/* i et j sont la MÊME *
 * case mémoire        */
i = 4; // j AUSSI vaut 4
j = 6; // i AUSSI vaut 6
```

```
int i(3);
int j(i); // copie
/* i et j vivent leur *
 * vie séparément     */
i = 4; // j vaut encore 3
j = 6; // i vaut encore 4
```

Attention aux pièges !

- sémantique de `const` :

```
int i(3);
const int& j(i); /* i et j sont les mêmes.                *
                  * On ne peut pas changer la valeur VIA J *
                  * (mais on peut le faire par ailleurs).  */
j = 12; // NON
i = 12; // OUI, et j AUSSI vaut 12 !
```

Spécificités des références (1/2)

(contrairement aux pointeurs) Une référence :

- doit absolument être initialisée (vers un objet existant) :

```
int i;
int& ri(i); // OK
int& rj;    // NON, la référence rj doit être liée à un objet !
```

- ne peut être liée qu'à un seul objet :

```
int i;
int& ri(i);
int j(2);
ri = j;  /* ne veut pas dire que ri est maintenant un alias de j, *
          * mais que i prend la valeur de j !!                    */
j = 3;
cout << i << endl; // affiche 2
```

Spécificités des références (2/2)

(contrairement aux pointeurs) Une référence :

- ne peut pas être référencée :

```
int   i(3)   ;
int&  ri(i)  ;
int&  rri(ri); // NON !
int&& rri(ri); // NON PLUS !!
```

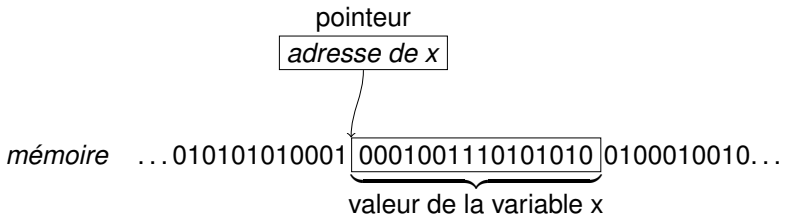
☞ on ne peut donc pas faire de tableau de références :-(

Les (vrais) pointeurs

Une variable est physiquement identifiée de façon unique par son **adresse**, c'est-à-dire l'adresse de l'emplacement mémoire qui contient sa valeur.

Un **pointeur** est une variable qui contient l'**adresse** d'un autre objet informatique.

☞ une « *variable de variable* » en somme



Note : une référence n'est pas un «vrai pointeur» car ce n'est pas une variable en tant que telle, c'est juste une «autre étiquette» :
différence entre «une autre étiquette» (référence) et «une variable contenant une adresse» (pointeur, un niveau de plus).

Les pointeurs : une analogie

Un pointeur c'est comme la *page d'un carnet d'adresse*
(sur laquelle on ne peut écrire qu'une seule adresse à la fois) :

déclarer un pointeur p.ex. `int* ptr;`	ajouter une page dans le carnet (mais cela ne veut pas dire qu'il y a une adresse écrite dessus !)
affecter un pointeur `ptr` p.ex. `ptr = &x;`	recopier sur la page `ptr` l'adresse d'une maison qui existe déja (mais `ptr` n'est pas la maison, c'est juste la page qui contient l'adresse de cette maison !)
allouer un pointeur `ptr` p.ex. `ptr = new int(123);`	aller construire une maison quelque part et noter son adresse sur la page `ptr` (mais `ptr` n'est pas la maison !)

Les pointeurs : une analogie

Un pointeur c'est comme la *page d'un carnet d'adresse*
(sur laquelle on ne peut écrire qu'une seule adresse à la fois) :

« libérer un pointeur » `ptr` (en fait, c'est « libérer l'adresse pointée par le pointeur » `ptr`) p.ex. `delete ptr;`	Aller détruire la maison dont l'adresse est écrite en page `ptr`. Cela ne veut pas dire que l'on a effacé l'adresse sur la page `ptr` ! ! mais juste que cette maison n'existe plus. Cela ne veut pas non plus dire que toutes les pages qui ont la même adresse que celle inscrite sur la page `ptr` n'ont plus rien (mais juste que l'adresse qu'elles contiennent n'est plus valide)

Les pointeurs : une analogie

Un pointeur c'est comme la *page d'un carnet d'adresse*
(sur laquelle on ne peut écrire qu'une seule adresse à la fois) :

copier un pointeur `p2` p.ex. `p1 = p2;`	On recopie à la page `p1` l'adresse écrite sur la page `p2`. Cela ne change rien à la page `p2` et surtout ne touche en rien la maison dont l'adresse se trouvait sur la page `p1` !
annuler/effacer un pointeur : `ptr = nullptr;`	On gomme la page `p1`. Cela ne veut pas dire que cette page n'existe plus (son contenu est juste effacé) ni (erreur encore plus commune) que la maison dont l'adresse se trouvait sur `p1` (c'est-à-dire celle que l'on est en train d'effacer) soit modifiée en quoi que ce soit ! ! Cette maison est absolument intacte !

Les pointeurs : la pratique

La déclaration d'un pointeur se fait selon la syntaxe suivante :

type identificateur;*

Cette instruction déclare une variable de nom *identificateur* de type pointeur sur une valeur de type *type*.

Exemple : `int* ptr;`
déclare une variable `ptr` qui pointe sur une valeur de type `int`.

L'initialisation d'un pointeur se fait selon la syntaxe suivante :

type identificateur(adresse);*

Exemples :

```
int* ptr(nullptr);

int* ptr(&i);

int* ptr(new int(33));
```

Opérateurs sur les pointeurs

C++ possède deux *opérateurs* particuliers en relation avec les pointeurs : `&` et `*`.

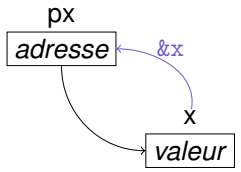
`&` est l'opérateur qui
retourne l'adresse mémoire de la valeur d'une variable

Si `x` est de type `type`, `&x` est de type `type*` (pointeur sur `type`).

Exemple :

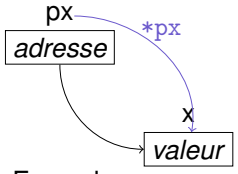
```
int x(3);
int* px(nullptr);

px = &x;
```



Opérateurs sur les pointeurs (2)

`*` est l'opérateur qui **retourne la valeur pointée par une variable pointeur**.
Si `px` est de type `type*`, `(*px)` est la valeur de type `type` pointée par `px`.



Exemple :

```
int x(3);          // x est de type entier
int* px(nullptr); // px est un pointeur sur entier

px = &x;       // px pointe sur la variable x
cout << *px    // affiche la valeur pointee par px : 3
     << endl;
```

Note : `*&i` est donc strictement équivalent à `i`

Houlala !

GARE AUX CONFUSIONS !

C++ utilise (malheureusement) deux *notations identiques* (`&` et `*`) pour *des choses différentes* !

`type& id` est une référence sur une variable `id` dans le passage par référence d'une fonction

`&id` est l'adresse de la variable `id`, par exemple en affectation d'un pointeur.

CE N'EST PAS LA MÊME CHOSE !

Houlala !

GARE AUX CONFUSIONS !

C++ utilise (malheureusement) deux *notations identiques* (`&` et `*`) pour *des choses différentes* !

`type* id;` déclare une variable `id` comme un pointeur sur un type de base `type`

`*id` (où `id` est un pointeur) représente le contenu de l'endroit pointé par `id`

CE N'EST PAS LA MÊME CHOSE !

Allocation de mémoire

Il y a **deux façons** d'*allouer de la mémoire* en C++.

① *déclarer des variables*
La réservation de mémoire est déterminée à la compilation :
allocation statique

② **allouer dynamiquement** de la mémoire **pendant l'exécution** d'un programme.

Les structures dynamiques comme les tableaux de taille variable (`vector`) ou les chaînes de caractères de type `string` sont des exemples de ce type.

Dans le cas particulier des *pointeurs*, l'allocation dynamique permet également de réserver de la mémoire **indépendamment de toute variable** : on pointe directement sur une zone mémoire plutôt que sur une variable existante.

Allocation de mémoire (2)

C++ possède deux opérateurs `new` et `delete` permettant d'**allouer** et de **libérer** dynamiquement de la mémoire.

pointeur = new *type*;

réserve une zone mémoire de type *type* et met l'adresse correspondante dans *pointeur*.

Il est également possible d'initialiser l'élément pointé directement lors de son allocation :

pointeur = new *type*(*valeur*);

Libération de la mémoire allouée

delete *pointeur*;

libère la zone mémoire allouée au pointeur *pointeur*.

C'est-à-dire que cette zone mémoire peut maintenant être utilisée pour autre chose. Il **ne** faut **plus y accéder !**...

 Conseil – Bonnes pratiques 

faire suivre tous les `delete` *de l'instruction* « *pointeur* = `nullptr;` »)

 Conseil – Bonnes pratiques 

Toute zone mémoire allouée par un `new` *doit impérativement être libérée par un* `delete` *correspondant !*

Exemple

```
int* px(nullptr);
px = new int;
*px = 20;
cout << *px << endl;
delete px;
px = nullptr;
```

```
int* px(nullptr);

px = new int(20);
...
```

```
int* px(new int(20));
...
```

Toujours allouer avant d'utiliser !

Attention ! Si on essaye d'utiliser (pour la lire ou la modifier) la valeur pointée par un pointeur pour lequel aucune mémoire n'a été réservée, une erreur de type `Segmentation fault` se produira à l'exécution.

Exemple :

```
int* px;
*px = 20;   // ! Erreur : px n'a pas été alloué !!
cout << *px << endl;
```

Compilation : OK

Execution

➥ *Segmentation fault*

Toujours allouer avant d'utiliser !

 Conseil – Bonnes pratiques 

Initialisez **toujours** vos pointeurs. Utilisez `nullptr` si vous ne connaissez pas encore la mémoire pointée au moment de l'initialisation :

```
int* px(nullptr);
```

C++11 Pointeurs intelligents (1/2)

Pour faciliter la gestion de l'allocation dynamique de mémoire et éviter l'oubli des `delete`,
C++11 introduit la notion de « **pointeur intelligent** » (*smart pointer*)
(dans la bibliothèque `memory`)

☞ Ces pointeurs font leur propre `delete` au moment opportun (*garbage collecting*)

Il existe trois types « pointeurs intelligents » :

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`

C++11 `unique_ptr`

Les `unique_ptr` pointent sur une zone mémoire n'ayant qu'un seul pointeur (« un seul propriétaire »)

☞ évite les confusions

$\Longrightarrow$ Ne peut être copié
mais peut être « déplacé », « transmis » plus loin

Note : si l'on veut libérer un `unique_ptr` avant le garbarge collector (c'est-à-dire faire le `delete` nous-même), on peut utiliser la fonction spécifique `reset()` :

```
ptr.reset()
```

Remet en plus `ptr` à `nullptr`.

C++11 `unique_ptr`, exemples

```
#include <memory>
// ...
unique_ptr<int> px(new int(20));
// ...
cout << *px << endl;
```

```
vector<unique_ptr<string>> noms({ new string("Pierre"),
                                  new string("Paul")    });
```

```
unique_ptr<Personne> naissance(string nom) {
  unique_ptr<Personne> bb(new Personne);
  // .. initialise le contenu pointé par bb
  return bb;
}
// ...
unique_ptr<Personne> adresse_quidam( naissance("Pierre") );
```

C++11 Pointeurs intelligents (2/2)

Les `unique_ptr` ne conviennent pas à toutes les situations.

Plus avancé :

- `shared_ptr` : zone mémoire partagée par plusieurs endroits du code
  (sans qu'aucun ne sache quand elle n'est plus utile aux autres)
- `weak_ptr` : presque comme un `shared_ptr`, mais peut avoir été détruit par ailleurs.
  (c'est-à-dire qu'on n'est pas compté dans les utilisateurs de cette zone mémoire).
  - ☞ utile pour «casser les cycles» de `shared_ptr`

Conclusion

Les pointeurs sont un vaste sujet, et il y aurait encore beaucoup à dire...