Programmation I :

Cours de programmation (C++)
Pointeurs et références

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C

Vidéos et transparents

https://www.coursera.org/learn/init-prog-cpp/home/week/7

☞ Semaine 7

Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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 en C++11 en raison de la *move semantic*)

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.

-  les « pointeurs intelligents » (***smart pointers***)

  gérés par le programmeur, mais avec des gardes-fous.
  Il en existe 3 : `unique_ptr`, `shared_ptr`, `weak_ptr`
  (dans la bibliothèque `memory`)

-   les `std::function`s  (dans la bibliothèque `functional`)

  pour désigner (« pointer sur ») des fonctions

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

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

Les différents « pointeurs »

☞ lesquels utiliser ?

utilisation	sur des données	sur des fonctions
référence	références (ou pointeurs à la C)	nom de la fonction
généricité	pointeur à la C ou index dans un tableau	`std::function` ou pointeur à la C
allocation dynamique	smart-pointers surtout `unique_ptr` (ou pointeurs à la C)	—

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

Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Le type référence

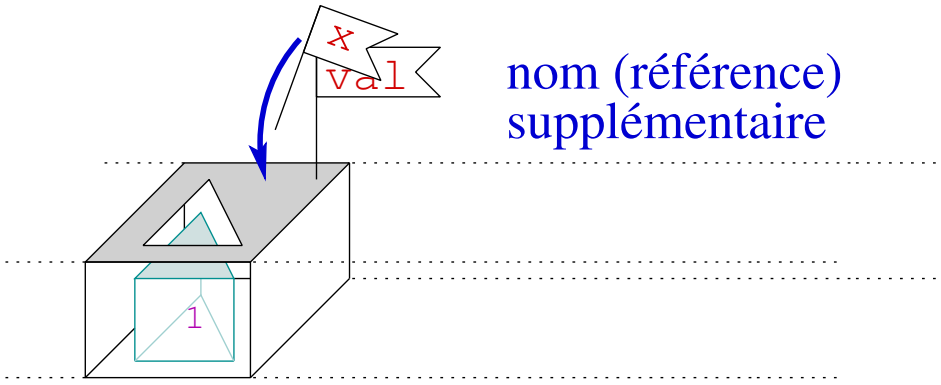
Une **référence** est un synonyme d'identificateur, un autre nom pour un objet existant.
La déclaration d'une référence se fait selon la syntaxe suivante :

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

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

Une référence permet donc de désigner un objet indirectement :

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



Les références sont très utiles lors du passage d'arguments aux fonctions, mais aussi parfois pour les valeurs de retour comme nous le verrons par exemple dans le cas de la surcharge d'opérateurs.

Attention aux pièges !

- sémantique de l'affectation

```
int i(3);
int& j(i); // alias
/* i et j sont la     *
 * MEME case memoire */
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 separement     */
i = 4; // j vaut encore 3
j = 6; // i vaut encore 4
```

- sémantique de `const`

```
int i(3);
const int& j(i); /* i et j sont les memes.      *
                  * 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

Une référence :

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

```
int i;
int& ri(i);  // OK

int& rj; // NON, la reference rj doit
         // etre liee a 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 ''pointe''  sur j
         // mais que i prend la valeur de j !
cout << i << endl; // affiche 2
```

- 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 :-(

C++11 r-value references

En C++11, il existe en plus des références vers les valeurs transitoires : les *r-value references*.

```
int&& temp(f());
```

Cela n'est utile que pour des raisons d'**optimisation** (éviter des copies, *move semantics*)

☞ Voir la mini-référence sur C++11

Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Pourquoi ne pas se contenter des pointeurs ?

Une référence c'est donc comme un pointeur mais avec quelques « gardes-fous »

☞ dans l'exemple précédent :

- `echange(int& a, int&b)` ne peut être appelée qu'avec `echange(i,j)`, où `i` et `j` sont des variables déclarées dans le bloc d'appel.
- avec `echange(int* a, int* b)` rien n'empêche l'appel où `b` par exemple ne pointe vers rien de valide !

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

Pourquoi ne pas se contenter des pointeurs ?

- L'usage de référence permet un style plus concis (le déréférencement explicite n'est plus nécessaire). Comparez :

```
void echange(int* a,
             int* b)
{
  int copie(*a);
  *a = *b;
  *b = copie;
}
```

```
void echange(int& a,
             int& b)
{
  int copie(a);
  a = b;
  b = copie;
}
```

☞ Le passage par référence **est le mode à privilégier absolument** en C++ lorsque l'on veut qu'une fonction puisse modifier une variable passée en argument

Houlala !

GARE AUX CONFUSION !

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

`type*` `id;` déclarare 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 !

Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Pointeurs sur fonctions

En C++, on peut en fait pointer sur n'importe quel objet. On peut en particulier **pointer sur des fonctions**.

La syntaxe consiste à mettre `(*ptr)` *à la place du nom* de la fonction.

Par exemple :
`double f(int i);` est une fonction qui prend un `int` en argument et retourne un `double` comme valeur
`double (*g)(int i);` est un **pointeur sur une fonction** du même type que ci-dessus.

On peut maintenant par exemple faire : `g=f;`

puis ensuite : `z=g(i);`

Note : pas besoin du `&` ni du `*` dans l'utilisation des pointeurs de fonctions.

Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Pointeurs sur fonctions en C++11

C++11 généralise la notion de « pointeur sur fonction » au travers du type `function` de la bibliothèque `functional`.

Exemple :

```
#include <functional>
...
function<double(double)> f;
...
double g(double x) { return x*x; }
...
f = g;
...
z = f(a + b);
```


Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Pointeurs sur fonctions, exemple

Supposons que vous ayez préprogrammé 5 fonctions :

```
double f1(double x);
...
double f5(double x);
```

et que vous souhaitiez écrire un programme capable d'intégrer l'une de ces fonctions :

```
do {
  cout << "De quelle fonction voulez-vous calculer "
       << "l'integrale [1-5] ? ";
  cin >> rep;
} while ((rep < 1) || (rep > 5))
```

Comment manipuler de façon générique la réponse de l'utilisateur ?

$\Rightarrow$ avec un **pointeur** sur la fonction correspondante.

Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Le programme complet (C++98) 1/2

```
#include <iostream>
#include <cmath>

double f1(double x) { return x*x;              }
double f4(double x) { return sqrt(exp(x));     }
double f5(double x) { return log(1.0+sin(x)); }

/* Fonction est un nouveau type : pointeur sur des fonctions *
 * prennant un double en argument et retournant un double    */
typedef double (*Fonction)(double);}

Fonction demander_fonction()
{
    int rep;
    Fonction choisie;
    do {
      cout << "De quelle fonction voulez-vous calculer "
           << "l'integrale [1-5] ? ";
       cin >> rep;
    } while ((rep < 1) || (rep > 5))

    switch (rep) {
      case 1: choisie = f1 ; break ;
      case 2: choisie = sin ; break ;
```


Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Le programme complet (C++98) 2/2

```
      case 3: choisie = exp ; break ;
      case 4: choisie = f4 ; break ;
      case 5: choisie = f5 ; break ;
    }

    return choisie;
}

double demander_nombre() { ... }
double integre({Fonction f, double a, double b) { ...f(a)... }

int main () {
  double a(demander_nombre());
  double b(demander_nombre());

  Fonction choix(demander_fonction());

  cout.precision(12);
  cout << "Integrale entre " << a
       << " et " << b << " :" << endl;
  cout << integre(choix, a, b) << endl;
  return 0;
}
```

Note : ce programme peut encore être amélioré, notamment en utilisant des tableaux...

Le programme complet (C++11)

Il suffit de remplacer

```
typedef double (*Fonction)(double);
```

par

```
typedef function<double(double)> Fonction;
```

Reste cependant une subtilité (avancée !) :
certaines fonctions de la librairie standard ont plusieurs protoypes (surcharge) et l'affectation d'une `function` est dans ce cas ambigue.
Par exemple, l'affectation du cas 2 :

```
choisie = sin;
```

provoque une erreur (d'ambiguité) du compilateur.
On est alors obligé de désambiguiser : indiquuer de quel `sin` il s'agit.
Cela se fait par :

```
choisie = (double(*)(double)) sin;
```

Pointeurs et effets de bord

Comme un pointeur contient l'**adresse mémoire** d'une valeur, si l'on passe un pointeur en argument d'une fonction, *toute modification faite sur cette valeur à l'intérieur de la fonction sera répercutée à l'extérieur*.
$\Longrightarrow$ **effet de bord**

Exemple (à ne pas suivre : utilisez plutôt le passage par référence) :

```
void swap(int* x, int* y) {
  int tmp(*x);
  *x=*y;
  *y=tmp;
}
main() {
  int x(3),y(2);
  cout << x << "," << y << endl; // affiche 3,2
  swap(&x, &y);
  cout << x << "," << y << endl; // affiche 2,3
}
```

Pointeurs constants et pointeurs sur des constantes

`type const* ptr;` (ou `const type* ptr`) déclare un **pointeur sur un objet constant** de type `type` : on ne pourra pas modifier la valeur de l'objet au travers de `ptr` (mais on pourra faire pointer `ptr` vers un autre objet).

`type* const ptr(&obj);` déclare un **pointeur constant sur un objet** `obj` de type `type` : on ne pourra pas faire pointer `ptr` vers autre chose (mais on pourra modifier la valeur de `obj` au travers de `ptr`).

Pour résumer : `const` s'applique toujours au type directement précédent, sauf si il est au début, auquel cas il s'applique au type directement suivant.

Pointeurs constants et pointeurs sur des constantes

Exemple :

```
   int i(2), j(3);
   int const* p1(&i);
   int* const p2(&i);

   cout << i << "," << *p1 << "," << *p2 << endl;    // 2,2,2

// *p1 = 5; /* erreur de compilation : on ne peut pas
             * modifier au travers de p1 */
   *p2 = 5;

   cout << i << "," << *p1 << "," << *p2 << endl;    // 5,5,5

   p1 = &j;
// p2 = &j; /* erreur de compilation : on ne peut pas
             *  modifier p2 */

   cout << i << "," << *p1 << "," << *p2 << endl;    // 5,3,5
```


Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Pointeurs & références

Déclaration : `type* pointeur;`

Déclaration/Initialisation :

```
type* pointeur(adresse);
unique_ptr<type>(new type(valeur));
type& reference(objet);
```

Adresse d'une variable : `&variable`

Accès au contenu pointé par un pointeur : `*pointeur`

Allocation mémoire :

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

Libération de la zone mémoire allouée :

`delete pointeur` (pour les « pointeurs classiques », obligatoire)

Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Révisions

- Révisions de quelques thèmes choisis avant la série notée
- La finalisation de l'étude de cas sur les structures et celle sur les pointeurs se fera la semaine prochaine

Jamila Sam
& Jean-Cédric Chappelier
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Pour préparer le prochain cours

- Vidéos et quiz du MOOC semaine 8 :
  - Puissance 4 : introduction [10:47]
  - Puissance 4 : premières fonctions [20:02]
  - Puissance 4 : fonction joue 1ère version [18:22]
  - Puissance 4 : fonction joue variantes et révision [08:43]
  - Puissance 4 : moteur de jeu [14:14]
  - Puissance 4 : fonctions est_ce_gagne et compte [15:51]
  - Puissance 4 : finalisation [08:07]