

Information, Calcul, Communication (partie programmation) : Pointeurs

Jean-Cédric Chappelier

Faculté I&C

Rappel du calendrier

	MOOC	décalage / MOOC	exercices prog. 1h45 Jeudi 8-10	cours prog. 45 min. Jeudi 10-11
1 10.09.26	--	-1	prise en main	Bienvenue/Introduction
2 17.09.26	1. variables	0	variables / expressions	variables / expressions
3 24.09.26	2. if	0	if – switch	if – switch
4 01.10.26	3. for/while	0	for / while	for / while
5 08.10.26	4. fonctions	0	fonctions (1)	fonctions (1)
6 15.10.26		1	fonctions (2)	fonctions (2)
- 22.10.26				
7 29.10.26	5. tableaux (vector)	1	vector	vector
8 05.11.26	6. string + struct	1	array / string	array / string
9 12.11.26		2	structures	structures
10 19.11.26	7. pointeurs	2	pointeurs	pointeurs
11 26.11.26		-	entrées/sorties	entrées/sorties
12 03.12.26		-	erreurs / exceptions	erreurs / exceptions
13 10.12.26		-	révisions	théorie : sécurité
14 17.12.26	8. étude de cas	-	révisions	Révisions

Objectifs du cours d'aujourd'hui

- ▶ Rappels sur les pointeurs :
 - ▶ 3 cas d'utilisation à ne pas confondre
 - ▶ 3 types de « pointeurs » :
 - ▶ références (mais voir point suivant)
 - ▶ pointeurs « à la C »
 - ▶ pointeurs « intelligents »
 - ▶ ne pas confondre pointeurs et références !!
- ▶ Etude de cas

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**

totalément 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::functions`

pour désigner (« pointer sur ») des fonctions
(dans la bibliothèque `functional`)

- ▶ 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	<code>std::function</code> ou pointeur à la C
allocation dynamique	smart-pointers surtout <code>unique_ptr</code> (ou pointeurs à la C)	—

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

Création de `unique_ptr`

En C++11 (et dans les vidéos du MOOC), pour créer des `unique_ptr` on écrivait typiquement :

```
unique_ptr<int> px(new int(20));  
  
noms.push_back(unique_ptr<string>(new string("Pierre")));
```

C++14 a introduit une syntaxe plus simple : `make_unique` :

`make_unique<Type>(arguments)`

Par exemple :

```
unique_ptr<int> px(make_unique<int>(20));  
  
noms.push_back(make_unique<string>("Pierre"));
```

Spécificités des références

Une référence :

- ▶ doit absolument être initialisée (vers un objet existant) ;
- ▶ ne peut être liée qu'à un seul objet ;
- ▶ donc la sémantique de `=` est très différente de celle des pointeurs (copie de valeur au lieu de pointer vers un nouvel objet)
- ▶ ne peut pas être référencée.
 - ☞ 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*.

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

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

- peu utile en soit, mais très utile pour l'optimisation d) le passage d'arguments et de valeur de retour de fonctions : retransmettre plus loin des objets sans les recopier.

Exemple :

```
GrosObjet x;  
... // remplissage de x  
vector<GrosObjet> hangar;  
  
/* Je veux mettre x au hangar et n'ai plus besoin de x lui-même *  
 * (mais utiliserai la "copie" mise au hangar).                               */  
  
hangar.push_back(x); // bof.. : DEUX GrosObjets :-(  
  
hangar.push_back(move(x)); // beaucoup mieux !  
// mais x n'est plus utilisable ici en tant que tel !
```

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

➡ *probablement* un « **Segmentation fault** »

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

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



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);  
unique_ptr<type>(new type(valeur));
```

Libération de la zone mémoire allouée :

```
delete pointeur (pour les « pointeurs à la C », obligatoire)  
pointeur.reset() (pour les « pointeurs intelligents », pas nécessaire)
```



Pointeurs (avancés)



Pointeur sur une constante : `type const* ptr;`

Pointeur constant : `type* const ptr(adresse);`

Pointeur sur une fonction :

`type_retour (*ptrfct)(paramètres...)`

C++11 `function<type_retour(paramètres...)> ptrfct`

Etudes de cas

- ▶ « exercice 0 » : listes chaînées
- ▶ réseau d'amis :
 - ▶ `struct Personne` qui a d'autres `Personnes` comme amis :
 1. combien d'amis ?
 2. comment représenter les amis ?
 - ▶ faire la fonction `ajoute_ami`
(attention aux pièges : pas de copie de `Personnes` !)
 - ▶ faire la fonction `est_ami`
(attention aux pièges : comparaison d'adresses)
 - ▶ améliorer la fonction `ajoute_ami` : pas 2 fois le même ami
 - ▶ (si temps) faire la fonction `cherche_amis(distance)`

Annexes

Les pointeurs sont un vaste sujet, et il y aurait encore beaucoup à dire en plus de ce qui est montré dans la vidéo...

Voici quelques thèmes choisis pour ceux souhaitent aller plus loin.



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
```



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*.

⇒ **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;  
}  
int main() {  
    int x(3), y(2);  
    cout << x << ", " << y << endl; // affiche 3,2  
    swap(&x, &y);  
    cout << x << ", " << y << endl; // affiche 2,3  
    return 0;  
}
```



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 a un `int` comme paramètre 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.



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);
```



Pointeurs sur fonctions, exemple



Vous souvenez vous de la fin de la *série 5*?

Comment faire pour ne pas recompiler le programme pour chaque nouvelle fonction ?

Supposons que vous ayez préprogrammé 5 fonctions :

```
double f1(double x);
```

```
...
```

```
double f5(double x);
```

et vous donnez le choix à l'utilisateur :

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

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

⇒ avec un **pointeur** sur la fonction correspondante.



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 *
 * ayant un double en paramètre et retournant un double */
typedef double (*Fonction)(double);

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

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



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 prototypes (surcharge) et l'affectation d'une `function` est dans ce cas ambiguë.

Par exemple, l'affectation du cas 2 :

```
choisie = sin;
```

provoque une erreur (d'ambiguïté) du compilateur.

On est alors obligé de désambigüiser que quel `sin` il s'agit.

Cela se fait par :

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



Allocation dynamique de tableaux « statiques » (= « à la C »)



Si l'on souhaite allouer dynamiquement un tableau « à la C » (ou « tableau statique »), on peut faire :

```
pointeur = new type[taille];
```

(puisque `type[taille]` est en effet un type C++ en tant que tel).

Exemple :

```
...  
size_t taille_tab(j+5);  
double* tab = new double[taille_tab];  
...  
tab[i]...
```

Note : il vaut bien sûr mieux utiliser un tableau dynamique !

```
vector<double> tab(j+5);
```

Pour détruire un tableau alloué de la sorte il faut utiliser :

```
delete[] pointeur;
```

Exemple : `delete[] tab;`