LANGAGE C++

POINTEURS ET RÉFÉRENCES

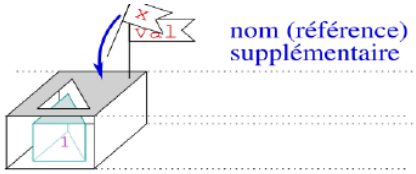
Les pointeurs

Dans un programme, pour garder un lien vers une donnée (une variable), on utilise des références ou des pointeurs.

En programmation, les pointeurs et références servent essentiellement à trois choses :

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

=> référence



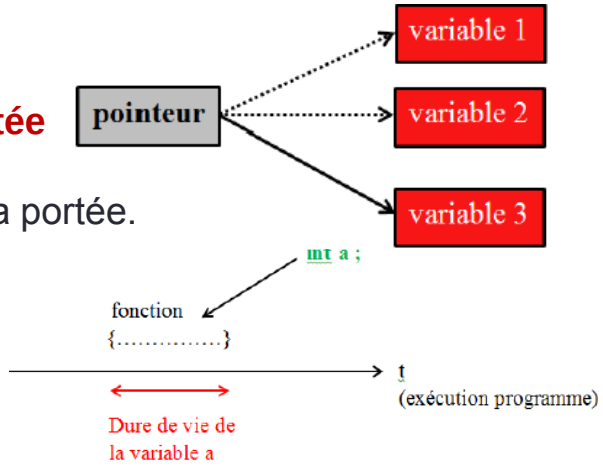
2. à pouvoir choisir des éléments non connus a priori (au moment de la programmation)

⇒ généricité

3. à pouvoir manipuler des objets dont la durée de vie dépasse la portée

=> allocation dynamique

La durée de vie de la variable a est égale au temps d'exécution de sa portée.



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 et références

En C++, il existe plusieurs sortes de « pointeurs » (pointeur et références) :

- **les références**

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

- **les « pointeurs « hérités de C » » (build-in pointers)**

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

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

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

Régle générale

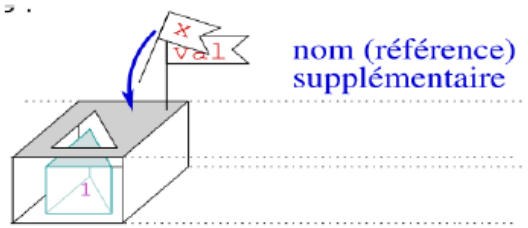
«Utilisez des références quand vous pouvez, des pointeurs quand vous devez.»

Référence

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

- Déclaration d'une référence, syntaxe : `type& nom_reference(identificateur);`

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



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

```
#include <iostream>
using namespace std;
void f(int& a);
int main()
{
    int b = 1;
    f(b);
    cout << "APRES b = " << b;
    return 0;
}
void f(int& a){
    ++a;
}
```

Les références

```cpp
int i(3);
int& j(i); // alias
/* i et j sont la MÊME *
* case mémoire */
cout << "AVANT (" << i << ", " << j << ")" << endl;
i = 4; // j AUSSI vaut 4
j = 6; // i AUSSI vaut 6
cout << "AVANT (" << i << ", " << j << ")" << endl;
```

```
AVANT (3, 3)
AVANT (6, 6)

Process returned 0 (0x0)
Press any key to continue.
```

```cpp
int i(3);
int j(i); // alias
/* i et j sont la MÊME *
* case mémoire */
cout << "AVANT (" << i << ", " << j << ")" << endl;
i = 4; // j AUSSI vaut 4
j = 6; // i AUSSI vaut 6
cout << "AVANT (" << i << ", " << j << ")" << endl;
```

```
AVANT (3, 3)
AVANT (4, 6)

Process returned 0 (0x0)
Press any key to continue.
```

Sémantique de const

```cpp
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; // Erreur de compilaion
i = 12; // OUI, et j AUSSI vaut 12 !
cout << "j = " << j;
```

```
j = 12
Process returned 0 (0x0)
Press any key to continue.
```

Spécificités des références

Contrairement aux pointeurs, une référence :

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

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

```
j = 12
Process returned 0 (0x0)
Press any key to continue.
```

- **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
```

```
2

Process returned 0 (0x0)
Press any key to continue.
```

- **ne peut pas être référencée**

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

```
Message
=== Build: Debug in test (compiler: GNU GCC Compiler) ===
In function 'int main()':
error: conflicting declaration 'int&& rri'
error: 'rri' has a previous declaration as 'int& rri'
warning: unused variable 'rri' [-Wunused-variable]
=== Build failed: 2 error(s), 1 warning(s) (0 minute(s), 0 second(s)) ===
```

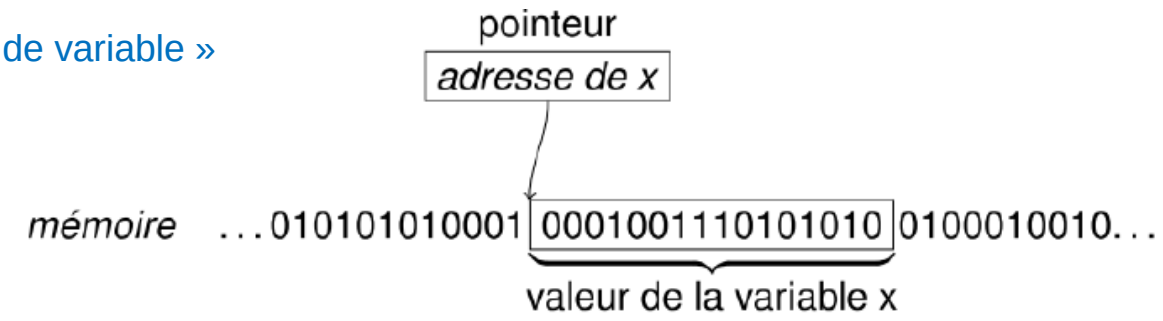
=> on ne peut donc pas faire de tableau de références

Les 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 (par ex, variable).

=> une « variable de variable »



Notes :

- Une référence n'est pas un «vrai pointeur» car ce n'est pas une variable en tant que telle.
- Une référence est «une autre étiquette»
- Un pointeur «une variable contenant une adresse»

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 : déclare une variable ptr qui pointe sur une valeur de type int.

```
int* ptr;
```

Les pointeurs

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

```
type* identificateur(adresse);
```

```
int* ptr(nullptr);
int* ptr(&i);
int* ptr(new int(33));
```

nullptr : mot clé C++, spécifie que le pointeur ne pointe sur rien

- Pour le compilateur, une variable est un emplacement dans la mémoire,
- Cet emplacement est identifié par une adresse
- Chaque variable possède une et une seule adresse.
- Un pointeur permet d'accéder à une variable par son adresse.
- Les pointeurs sont des variables faites pour contenir des adresses.

Opérateurs sur les pointeurs

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

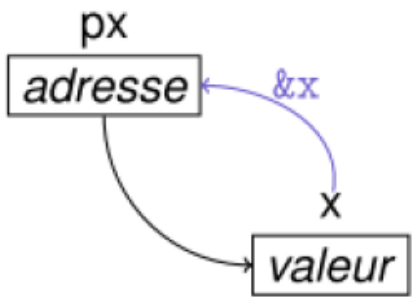
- **& est l'opérateur qui retourne l'adresse mémoire de la valeur d'une variable**

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

```
int x(3);
int* px(nullptr);
px = &x;
cout << "Le pointeur px contienT l'adresse " << px << endl;
```

```
Le pointeur px contient l'adresse 0x28fee8

Process returned 0 (0x0)   execution time :
Press any key to continue.
```

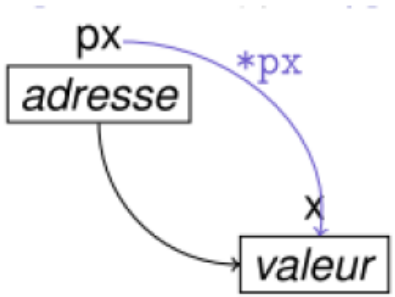


- *** est l'opérateur qui retourne la valeur pointée par une variable pointeur.**

Si px est de type type*, (*px) est la valeur de type type pointée par px.

```
int x(3);
int* px(nullptr);
px = &x;
cout << "Le pointeur px pointe sur la valeur " << *px;
```

```
Le pointeur px pointe sur la valeur 3
Process returned 0 (0x0)   execution time
Press any key to continue.
```



Note : *&i est donc strictement équivalent à i

Opération sur les pointeurs

Attention aux confusions

- **int& id(i)** - id est une référence sur la variable entiére i
- **&id** est l'adresse de la variable id

```
int i(3);
int& id(i);
cout << "i = " << i << " id = " << id << endl;
cout << "adresse de i = " << &i;
```

```
i = 3 id = 3
adresse de i = 0x28fee8
Process returned 0 (0x0)
Press any key to continue.
```

- **int* id**; déclare une variable id comme un pointeur sur un entier
- ***id** (où id est un pointeur) représente le contenu de l'endroit pointé par id

```
int* p(nullptr);
int i = 2;
p = &i;
cout << "Le pointeur p pointe sur la valeur " << *p;
```

```
Le pointeur p pointe sur la valeur 2
Process returned 0 (0x0)   execution
Press any key to continue.
```

Allocation mémoire

Il y a trois façons d'allouer de la mémoire en C++

1. allocation statique

La réservation mémoire est déterminée à la **compilation.**
L'espace alloué statiquement est déjà réservé dans le fichier exécutable du programme lorsque le système d'exploitation charge le programme en mémoire pour l'exécuter.
L'avantage de l'allocation statique se situe essentiellement au niveau des performances, puisqu'on évite les coûts de l'allocation dynamique à l'exécution

2. allouer dynamiquement

La mémoire est réservée pendant **l'exécution** du programme.
L'espace alloué dynamiquement ne se trouve pas dans le fichier exécutable du programme lorsque le système d'exploitation charge le programme en mémoire pour l'exécute.
Par exemple, les tableaux de taille variable (**vector**), les chaînes de caractères de type **string**.

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

Syntaxe réserve une zone mémoire de type type et affecte l'adresse dans la variable **pointeur.**

```
pointeur = new type;
```

Allocation mémoire

```
int* p;
p = new int;
*p = 2;
cout << "p = " << p << " *p = " << *p;
```

```
int* p;
p = new int(2);
cout << "p = " << p << " *p = " << *p;
```

```
p = 0x350cf0 *p = 2
Process returned 0 (0x0)
Press any key to continue.
```

Libérer la mémoire allouée

Syntaxe `delete pointeur;` libère la zone mémoire allouée au pointeur

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

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

```
int* px(nullptr);
{
    px = new int(4);
    int n = 6;
}
cout << "*px = " << *px;
delete px;
px = nullptr;
cout << "n = " << n;
```

```
Message
=== Build: Debug in test (compiler: GNU GCC Compiler) ===
In function 'int main()':
warning: unused variable 'n' [-Wunused-variable]
error: 'n' was not declared in this scope
=== Build failed: 1 error(s), 1 warning(s) (0 minute(s), 0 second(s)) ===
```

Bonnes pratiques

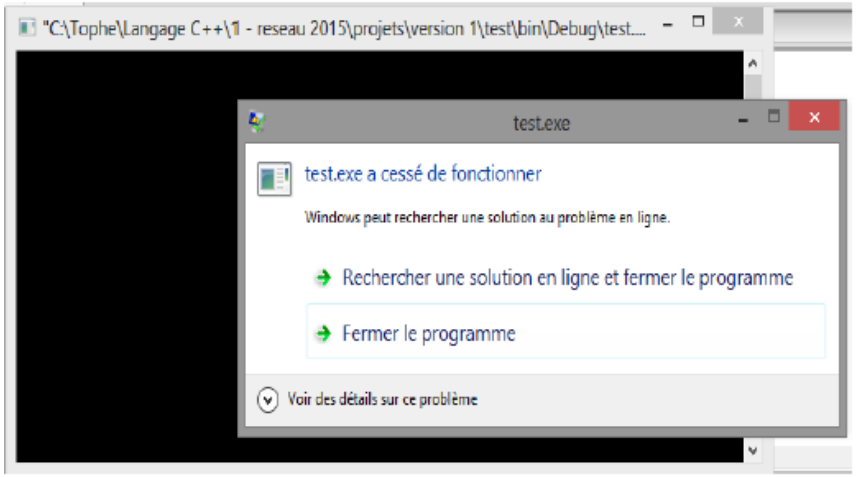
- faire suivre tous les **delete** de l'instruction **« pointeur = nullptr; »**)
- toute zone mémoire allouée par un **new** doit impérativement être libérée par un **delete** correspondant ! (fuite mémoire)

Notes :
Une variable allouée statiquement est désallouée automatiquement (à la fermeture du bloc).
Une variable (zone mémoire) allouée dynamiquement doit être désallouée explicitement par le programmeur.

Libérer la mémoire allouée

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.

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



Compilation : Ok

Exécution : arrêt programme (Segmentation fault)

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);
if(px != nullptr){
    *px = 20;
    cout << *px << endl;
}
```

Exemple

- gérer soi-même le moment de l'allocation et désallocation mémoire (maîtriser durée de vie variable, données)
- partager une variable dans plusieurs morceaux du code (passage d'argument par adresse);
- sélectionner une valeur parmi plusieurs options. :

```
struct Etudiant {
    string nom;
    double moyenne;
    char sexe;  // 'F' ou 'M'
};
void afficherEtudiant(Etudiant e);
void modifierEtudiant(Etudiant *e);
int main()
{
    Etudiant etudiant = {"Diarra",12.5,'F'};
    cout << "---- AVANT MODIFICATION ----" << endl;
    afficherEtudiant(etudiant);
    modifierEtudiant(&etudiant);
    cout << "---- APRES MODIFICATION ----" << endl;
    afficherEtudiant(etudiant);
    return 0;
}
```

```
void afficherEtudiant(Etudiant e){
    if(e.sexe=='F') cout << "Madame ";
    else cout << "Monsieur ";
    cout << e.nom << " votre moyenne est de "
         << e.moyenne << endl;
}
void modifierEtudiant(Etudiant *e){
    cout << "Saisir nom : ";
    cin >> (*e).nom;
    cout << "Saisir moyenne : ";
    cin >> (*e).moyenne;
    cout << "Saisir sexe : ";
    cin >> (*e).sexe;
}
```

```
---- AVANT MODIFICATION ----
Madame Diarra votre moyenne est de 12.5
Saisir nom : Coulibaly
Saisir moyenne : 11
Saisir sexe : M
---- APRES MODIFICATION ----
Monsieur Coulibaly votre moyenne est de 11

Process returned 0 (0x0)   execution time :
Press any key to continue.
```

Allocation statique de l'étudiant

Autre exemple

```
int main()
{
    string reponseA, reponseB, reponseC;
    reponseA = "La peur des jeux de loterie";
    reponseB = "La peur du noir";
    reponseC = "La peur des vendredis treize";
    cout << "Qu'est-ce que la 'kenophobie' ? " << endl; //On pose la question
    cout << "A) " << reponseA << endl; //Et on affiche les trois propositions
    cout << "B) " << reponseB << endl;
    cout << "C) " << reponseC << endl;
    char reponse;
    cout << "Votre reponse (A,B ou C) : ";
    cin >> reponse; //On récupère la réponse de l'utilisateur
    string *reponseUtilisateur(0); //Un pointeur qui pointera sur la réponse choisie
    switch(reponse)
    {
      case 'A':
          reponseUtilisateur = &reponseA; //On déplace le pointeur sur la réponse choisie
        break;
        case 'B':
          reponseUtilisateur = &reponseB;
        break;
        case 'C':
          reponseUtilisateur = &reponseC;
        break;
    }
    //On peut alors utiliser le pointeur pour afficher la réponse choisie
    cout << "Vous avez choisi la reponse : " << *reponseUtilisateur << endl;
    return 0;
}
```

Autre exemple

Allocation dynamique de l'étudiant

```
Saisir nom : Diallo
Saisir moyenne : 17.4
Saisir sexe : F
Madame Diallo votre moyenne est de 17.4

Process returned 0 (0x0)   execution time
Press any key to continue.
```

```
struct Etudiant {
        string nom;
        double moyenne;
        char sexe;  // 'F' ou 'M'
    };
void initialiserEtudiant(Etudiant *e);
void afficherEtudiant(Etudiant e);
int main()
{
    Etudiant *ptrEtudiant;
    ptrEtudiant = new Etudiant;
    initialiserEtudiant(ptrEtudiant);
    afficherEtudiant(*ptrEtudiant);
    delete(ptrEtudiant);
    return 0;
}
```

```
void initialiserEtudiant(Etudiant *e){
        cout << "Saisir nom : ";
        cin >> (*e).nom;
        cout << "Saisir moyenne : ";
        cin >> (*e).moyenne;
        cout << "Saisir sexe : ";
        cin >> (*e).sexe;
}
void afficherEtudiant(Etudiant e){
        if(e.sexe=='F') cout << "Madame ";
        else cout << "Monsieur ";
        cout << e.nom << " votre moyenne est de " << e.moyenne << endl;
}
```

Pointeurs intelligents C++11

Pour faciliter la gestion de l'allocation dynamique de mémoire et éviter **l'oubli** des **delete**, c++ 2011 introduit la notion de « **pointeur intelligent** », smart pointer (**bibliothèque memory**).

Ces pointeurs font leur propre delete au moment opportun au ramasse miette (garbage collecting).
Il existe trois types « pointeurs intelligents » :

- **unique_ptr**

un seul pointeur de type unique_ptr peut pointer sur la même zone mémoire allouée dynamiquement

- **shared_ptr**
- **weak_ptr**

} Plusieurs pointeur de même type peuvent pointer sur la même zone mémoire allouée dynamiquement.

Exemple de problème :

```
int* p1(new int(2));
int* p2(p1);
cout << "Avant delete *p1 = " << *p1 << endl;
delete p2;
cout << "Apres delete *p1 = " << *p1 << endl;
```

```
Avant delete *p1 = 2
Apres delete *p1 = 3673776

Process returned 0 (0x0)
Press any key to continue.
```

Pointeur de type unique_ptr C++11

```cpp
#include <iostream>
#include <memory>
using namespace std;
int main()
{
    unique_ptr<int> px(new int(20));
    cout << *px << endl;
    return 0;
}
```

```
20

Process returned 0 (0x0)
Press any key to continue.
```

Les **unique_ptr** pointent sur une zone mémoire n'ayant qu'un seul pointeur (« un seul propriétaire ») => Ne peut être copié mais peut être « déplacé », « transmis » plus loin (« move semantic »)

```cpp
unique_ptr<int> ptr1(new int(2));
unique_ptr<int> ptr2 = ptr1;
```

```
Message
=== Build: Debug in test (compiler: GNU GCC Compiler) ===
In function 'int main()':
error: use of deleted function 'std::unique_ptr<_Tp, _Dp>::unique_ptr(cons
error: declared here
=== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

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() :

```cpp
ptr.reset();
```

Remet en plus ptr à **nullptr**.