R6.A.05 : Développement avancé – C++

Plan & Pointeurs / Tableaux / Gestion de la mémoire / Tableau de caractères

J-F. Kamp

Janvier 2025

PLAN du cours R6.A.05

Tous les cours sur 45 min.

- **Semaine 03/2025 :**
  - **Cours1 -** les pointeurs, les tableaux, le tableau de caractères
  - **Cours2 -** classes et objets en C++, structure du code, destructeur
  - **TP1** : pointeurs, tableau de caractères, premières compilations
  - **TP2** : classes et objets en C++

- **Semaine 04 :**
  - **Cours3 -** classes et objets en C++, copy-constructeur, affectation
  - **Cours4 -** Classe string, I/O fichiers et formatage
  - **TP3** : construction de classes pour le formatage des chaînes
  - **Mini-projet**

PLAN du cours R6.A.05

- **Semaine 05 :**
  - **Cours5 -** Fonction amie, surcharge des opérateurs, héritage simple, fonction virtuelle
  - **Mini-projet**

- **Mini-projet noté (1/3 note finale)**

- **Contrôle terminal en semaine 06 (2/3 note finale)**

- **Mais cela peut changer… ChatGPT interdit…**

Références bibliographiques

- Programmer en langage C++, C. Delannoy, Editions Eyrolles, 2004

- Le langage C++, B. Stroustrup, Editions InterEditions

- Langage C, C. Delannoy, Editions Eyrolles, 2002

- WEB :
    - http://www.cplusplus.com/
    - http://www.cppreference.com/

- Moodle : R6.A.05 - clé inscription = r6a05
  Polys au format PDF (pas de papier)

PLAN du Cours1

1. Comparaison Java/C++

2. Les pointeurs
   2.1. les opérateurs * et &
   2.2. opérations sur les pointeurs

3. Arguments d'une fonction
   3.1. passage par pointeur / référence
   3.2. retour de type pointeur / référence

4. Tableaux statiques et dynamiques

5. Gestion manuelle de la mémoire

5. Tableau de caractères

Similitudes et différences avec Java

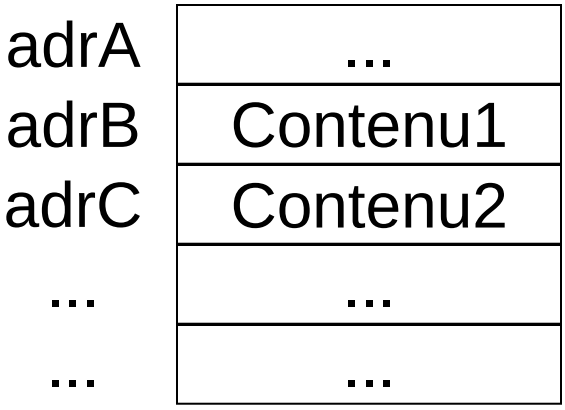
C++ et Java (dans les grandes lignes)

C++	*Java*
• Rapide (temps réel)	• Moins rapide
• Non portable	• Meilleure portabilité
• Gestion mémoire man.	• Ramasse-miettes
• Très (trop) vaste (inclus lang. C)	• Vaste mais API mieux organisée
• Compile « n'importe quoi » (crash PC)	• Evite les erreurs dès la compilation
• Compliqué (Templates, pointeurs, héritage multiple, surcharge opérateurs)	• + facile à apprendre, fantaisies moins acceptées
• Programmation très pointue (mais illisible)	• Programmation plus lisible

Les pointeurs

Pointeur

- **TOUTES** les variables en C++ (primitives et objets) peuvent être manipulées soit par pointeur (*adresse*) soit par valeur



- Un pointeur est une variable qui contient l'adresse mémoire d'une donnée primitive ou objet (référence en Java)
- Un pointeur « pointe » toujours sur un type de donnée (int, float, ...) => il faut préciser ce type dans la déclaration

```
int* a;          // « a » est une variable de type
                 // pointeur (adresse) sur un entier
CCercle* c1; // « c1 » est une variable de type
                 // pointeur sur un objet « cercle »
```

Pointeur

- Par défaut, une variable de type pointeur a un contenu indéfini mais NON NULL (≠ Java !!)

- Comme en Java (cf. référence), la valeur NULL peut être affectée à une variable de type pointeur

```
int* a;          // « a » contient une adresse indéfinie
int b;           // « b » contient une valeur indéfinie

CCercle* c1; // « c1 » contient adresse indéfinie
CCercle c2;  // « c2 » est le contenu de l'objet
             // construit avec le constructeur par
             // défaut de CCercle

a = NULL;    // « a » pointe dans le vide
```

L'opérateur &

A tout moment, il est possible de connaître l'adresse d'un type primitif ou type objet grâce à l'opérateur "&"

```
int* a;          // « a » contient une adresse indéfinie
int b;           // « b » contient une valeur indéfinie

a = &b;          // « a » contient maintenant
                 // l'adresse de « b »

CCercle* c1; // « c1 » contient une adresse indéfinie
CCercle c2;  // « c2 » est le contenu de l'objet CCercle
             // construit avec le constructeur par défaut
             // de CCercle

c1 = &c2;    // « c1 » pointe maintenant
             // sur l'objet « c2 » => c1 et
             // c2 concernent le MEME objet
```

L'opérateur ∗

L'opérateur ∗ permet l'accès au CONTENU de la mémoire à partir de son adresse (opération de "déréférencement"). Attention : ∗ sert aussi à déclarer un pointeur !!

```
int* a;          // « a » contient une adresse indéfinie
int b;           // « b » contient une valeur indéfinie

a = &b;          // « a » contient maintenant
                 // l'ADRESSE de la case mémoire de
                 // contenu « b »
b = 455;
(*a) = 112;      // le CONTENU de la mémoire à
                 // l'adresse « a » prend la valeur 112

cout << "b = " << b;  // Qu'affiche-t-on comme valeur
                      // pour b ?

b = 455;
*(&b) = 112;     // Correct

cout << "b = " << b;  // Qu'affiche-t-on comme valeur
                      // pour b ?
```

Comparaison de pointeurs

Comme en Java (cf. référence), il est possible (et même utile) de comparer l'égalité de pointeurs de même type :

```
int* p1 = NULL;          // p1 null
int* p2 = NULL;          // p2 null

// Allocation dynamique d'un entier primitif.
// L'opérateur « new » renvoie l'adresse de
// la plage mémoire réservée (idem Java).
p1 = new int ( 65 );
p2 = new int ( 65 );
```

`p1 == p2` renvoie vrai ssi p1 et p2 pointent sur la même case mémoire

Erreurs à ne pas commettre

```
int* a;          // « a » contient une adresse indéfinie
int b;           // « b » contient une valeur indéfinie
int c;           // « c » contient une valeur indéfinie

b = (*a) ;       // possible mais dangereux car
                 // l'adresse « a » n'est peut-être pas
                 // valide

(*a) = 5669;     // possible mais dangereux car
                 // l'adresse « a » est indéfinie

b = a;           // FAUX car incompatibilité de type

(*b) = a;        // FAUX car « b » ne contient pas
                 // l'adresse d'une case mémoire

a = &c;          // correct mais le contenu *a est
                 // indéfini puisque « c » est indéfini
```

Arguments d’une fonction

Passage des arguments

- Syntaxe de base

```
type_de_retour nomFct (type var1, type var2, …) {
        ...
        ...
        return ...;
}
```

- Trois manières différentes de transmettre les arguments :
    - par **valeur** : ***type*** est primitif ou objet, **var1** contient la COPIE du contenu d'une zone mémoire
    - par **pointeur** : ***type*** est de type pointeur, **var1** contient l'adresse d'une zone mémoire
    - par **référence** : ***type*** est primitif ou objet suivi du symbole **&**, **var1** est le contenu de la MEME case mémoire que l'argument

Passage par pointeur des arguments

```
// var1 est de type pointeur sur un réel
// var1 recevra l'adresse mémoire d'un réel
void maFct ( float* var1 ) {

        // modifie le CONTENU de la case mémoire
        // pointé par « var1 »
        *var1 = 566.32;

        // !! écrire var1 = 566.32 est totalement FAUX
        // car « var1 » contient une adresse !
}

/*******************************/

int main () {

        float a = 789.23;

        // passage de l'argument par pointeur
        maFct ( &a );

        // le contenu de « a » est modifié par maFct
        cout << "a = " << a << "\n"; // on affiche 566.32
}
```

Passage par référence des arguments

```
void maFct ( float& var1 ) {
        // modifie le CONTENU de la MEME case
        // mémoire que l'argument passé en paramètre
        var1 = 566.32;

        // !! écrire *var1 = 566.32 est totalement FAUX
        // car « var1 » est le contenu de la case mémoire
}
/*******************************/
int main () {
        float a = 789.23;

        // passage de l'argument par référence
        // mais on ne s'en rend pas compte !!
        maFct ( a );

        // le contenu de « a » est modifié par maFct
        cout << "a = " << a << "\n"; // on affiche 566.32
}
```

Tableaux statiques et dynamiques

Tableaux statiques

- Si la taille du tableau est fixée dans la déclaration du tableau, le compilateur connait le nombre de cases successives à réserver en mémoire.

=> allocation statique
=> PAS de mot-clé new pour réserver l'espace en mémoire
=> PAS de delete[] pour libérer l'espace réservé
=> le tableau existe dès sa déclaration
=> le tableau est détruit automatiquement

```
#define N 20 // constante à placer dans le « fichier .h »

double a [N]; // tableau de N réels (0 ... N-1) chaque
              // case de type réel a un contenu
              // indéterminé
a[0] = 10.25;
a[2] = a[0];    // la troisième case contient 10.25
```

Tableaux dynamiques

- Dans la déclaration du tableau, aucune taille n'est fournie (le compilateur ne connait donc pas le nombre de cases à allouer). En cours d'exécution, la taille devient connue et l'allocation est possible.

  => allocation dynamique

  => mot-clé new pour réserver l'espace en mémoire et renvoyer l'**adresse**

  => mot-clé delete[] OBLIGATOIRE pour libérer la mémoire (pas de ramasse-miettes en C++)

  => aucune case de tableau n'existe avant son allocation dynamique

```
void maFct () {
    float* leTab; // le tableau est déclaré mais pas alloué
    int k = 20;

    leTab = new float[k]; // le tableau est alloué
    leTab[9] = 12.36;

    delete[] leTab; // la mémoire doit être libérée
}
```

Tableaux et pointeurs

- Dans tous les cas (int leTab[N] ou int* leTab), le nom du tableau (leTab) est de type pointeur.

- Pointeur sur quoi ? La première case du tableau.
  => leTab == &leTab[0]

Le dépassement de tableau : danger

- En C/C++, aucune protection n'est prévue contre le dépassement de tableau (attention danger !!) :

```
// soit un tableau statique leTab de 11 cases
// => indices autorisés (et garantis) : 0...10
int leTab[11];

// parfaitement possible à la compilation
// grande chance de bug à l'exécution
leTab[52] = 56;
*(leTab-26) = 23;
```

- D'une manière plus générale à partir de n'importe quelle variable de type pointeur, il est possible de « se promener » dans la mémoire (mais risque de plantage évident à l'exécution).

- En Java, l'exception ArrayOutOfBounds prévient immédiatement d'un dépassement de tableau.

Tableaux et pointeurs

Exercices

```
int* var1;          // variable de type pointeur sur un entier
int var2 = 21;      // variable de type entier
int var3[10];       // un tableau de 10 cases

var1 = new int[20]; // var1 pointe sur un tableau de 20

var1 = var3;        // correct, var1 pointe sur 1 autre tableau

var1 = &var2;       // correct, affectation d'une adresse dans
                    // un pointeur

*(var1 + 2) = 566; // FAUX, mais possible...

*(var3 + 55) = -87; // FAUX, mais possible...

var2 = var3[9];          // Sans problème

var2 = *(var1 + 0); // correct, car équivalent à *var1
```

Taille d'un tableau

- Dans le cas du langage C/C++, il n'y a aucune façon simple de connaître la taille (nbre octets) de la mémoire pointée par une variable.

```
int leTab[11];
```

On pourrait croire que :

```
int taille = sizeof ( int* ) / sizeof ( int );
```

MAIS `sizeof ( int* )` n'est pas la taille du tableau mais le nombre d'octets occupés par le type `int*`.

- En Java, on utilise simplement l'attribut `length` pour connaître la taille d'un tableau

Gestion de la mémoire

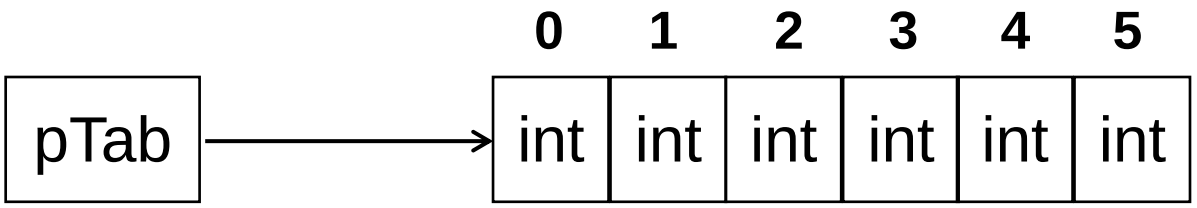
- L'allocation dynamique de mémoire pour créer :
  - un type primitif :
    ```
    int* pInt1 = new int ( 89 );
    ```
  - un tableau de types primitifs :
    ```
    int* pTab = new int[n];
    ```
  - un objet :
    ```
    CCercle* pC1 = new CCercle (20, 0, 30);
    ```
  - un tableau d'objets :
    ```
    CCercle* pCTab = new CCercle [20];
    ```

NEW nécessite la libération A LA MAIN (pas de ramasse-miettes en C++) de cette mémoire dès que celle-ci ne sera plus utilisée (mot-clé delete ou delete[] pour un tableau).

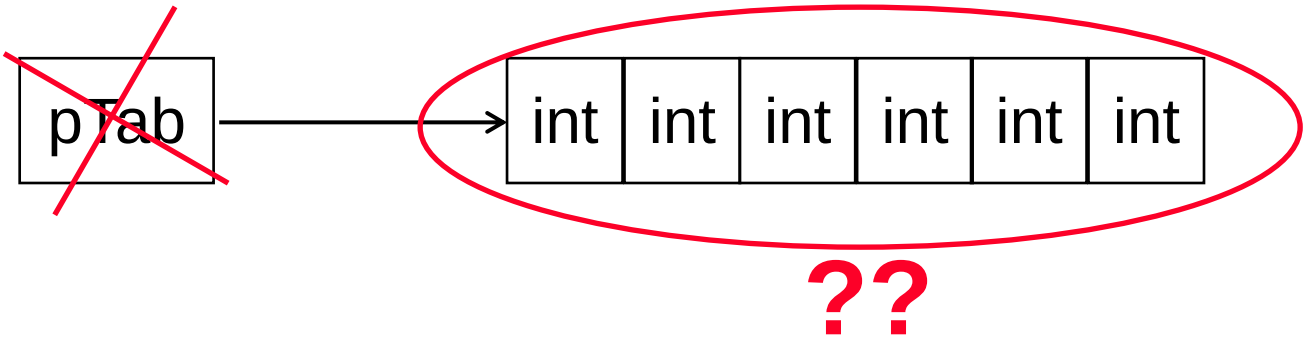
Gestion de la mémoire

- Soit une allocation dynamique de mémoire dans un bloc pour créer un tableau de type primitif int :

```
int* pTab = new int[6];
```



- Juste avant de sortir du bloc, les variables locales sont détruites.



En Java **??** est détruit par le ramasse-miettes MAIS en C++ il faut le détruire à la main => juste avant de sortir du bloc faire !!delete[] pTab;!!

Exemple1 : delete ou pas delete ?

Faut-il détruire la mémoire pointée par pVar1 ?

Si oui, où placer l'instruction delete pVar1; ?

```
// var1 est de type pointeur sur un réel
// var1 recevra l'adresse mémoire d'un réel
void maFct ( float* pVar1 ) {

        // modifie le CONTENU de la case mémoire
        // pointé par « var1 »
        *pVar1 = 566.32;
}

/*******************************/

int main () {

        float a = 789.23;

        // passage de l'argument par pointeur
        maFct ( &a );

        // le contenu de « a » est modifié par maFct
        cout << "a = " << a << "\n"; // on affiche 566.32
}
```

Exemple2 : delete ou pas delete ?

Faut-il détruire la mémoire pointée par pVar1 ?
Si oui, où placer l'instruction delete pVar1; ?
Faut-il détruire la mémoire pointée par pVar2 ?
Si oui, où placer l'instruction delete pVar2; ?

```
float* maFct ( ) {
        // Création dynamique d'une donnée réelle
        // pVar1 est un pointeur sur un réel
        float* pVar1 = new float (566.32);

        return pVar1;
}
/*******************************/
int main () {
        // Variable de type pointeur sur un réel
        float* pVar2 = NULL;

        // Appel de la fonction : retour d'un pointeur
        pVar2 = maFct ( );

        // Affichage de la donnée pointée par pVar2
        cout << *pVar2 << "\n"; // on affiche 566.32
}
```

Tableau de caractères

Tableau de caractères

- En C(C++), une chaîne de caractères est FORCEMENT un tableau dans lequel chaque case contient un seul caractère

- Piège1 : il faut impérativement terminer la chaîne par le caractère nul ‘\0’ => une chaîne de n caractères doit être stockée dans un tableau d’au moins n + 1 cases

```
char tabChar[10] = "Bonjour";
// tabChar est un pointeur sur un tableau automatique

=> tabChar contient { ‘B’, ‘o’, ‘n’, ‘j’, ‘o’, ‘u’, ‘r’, ‘\0’, ?, ? }

char tabChar2[2] = "OK";    // Ne compile pas !!

// Autre possibilité (presque) correcte
char* pChar = "Une chaîne aussi longue que...";

=> pChar pointe sur la première case du tableau alloué AUTOMATIQUEMENT.

cout << pChar << endl;       // Affichage de la chaîne
```

Tableau de caractères

- Piège2 : il FAUT prévoir le caractère nul en fin de chaîne si le tableau de caractères est NON initialisé à la déclaration

```
char tabChar[10];
char i;

for ( i = 0; i < 5; i++ ) { tabChar[i] = ‘A’ + i; }

tabChar[5] = ‘\0’;      // !! ne pas oublier le ‘\0’

=> tabChar contient { ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘\0’, ?, ?, ?, ? }
```

- Piège3 : un tableau de caractères *initialisé avec un pointeur* est NON modifiable

```
char* pChar = "Bonjour";              // !! Warning => NON
const char* pChar = "Bonjour";        // const obligatoire
                                      // non modifiable

pChar[3] = ‘z’;                       // impossible
*( pChar + 3 ) = ‘z’;                 // impossible
pChar++;                              // possible

char tabChar[] = "Bonjour";           // OUI
tabChar[3] = ‘z’;                     // OUI
```

Tableau de caractères

Fonctions utiles pour la manipulation de chaînes de caractères en C (nécessite <string.h>) :

- unsigned int strlen (char* str) : renvoie le nbre de caract. de la chaîne, '\0' NON compris

- char* strcpy (char* strDest, const char* strSrc) : recopie strSrc dans strDest y compris '\0' (et renvoie strDest).
  !! strDest doit être alloué au préalable avec la bonne longueur !!

- Méthode getline (char* str, int taille, char delim = '\n') : lit les caractères sur le flot d'entrée (cin pour clavier), les place dans str et rajoute le caractère '\0' à la fin.
  La saisie s'arrête si :
  - delim a été trouvé
  - ou (taille - 1) caractères ont été lus

  !! la chaîne str doit être allouée au préalable
  Méthode à utiliser avec cin (instance istream)