Programmation avancée en C++: rappel généraux

Emmanuel Franck

Bureau 225, UFR math info
mail: emmanuel.franck@inria.fr

Programme du cours :

- Cours 1: Rappels généraux (pointeurs, tableaux ...).
- Cours 2: Rappels et compléments sur les fonctions
- Cours 3: Classes
- Cours 4: Exemple de classes, classes avancées.
- Cours 5: Cmake et visual studio code (vsc)
- Cours 6: Corrgé CC1 et Templates I.
- Cours 7: Templates II
- Cours 8: Héritage simple.
- Cours 9: Héritage multiple et polymorphisme.
- Cours 10: Polymorphisme et classe abstraite.
- Cours 11: Corrigé CC2 + Polymorphisme et Héritage.
- Cours 12: Librairie STD I.
- Cours 13: Librairie STD II.
- Cours 14: Révision.

- **Fil conducteur**: exemples tirés des mathématiques (algèbre, TAN, Data sciences etc).

Évaluation

- Examen machine en TP: 25/02, 1h30 (1/10)
- Examen machine en TP: 01/04, 1h30 (1/10)
- Examen écrit, 2h (2/5)
- Projet C++ (2/5) A rendre le ??

Intervenant en TP: Guillaume Steimer

1. Premier code et compilation

Généralités

- Les caractéristiques du C++:
  - **Langage compilé**: traduit une fois pour toute par un compilateur puis peut être utilisé sans logiciel additionnel (à l'inverse des langages interprétés comme Python).
  - **Langage typé**: on doit définir le type d'une variable, d'un tableau (entier, booléen etc) au préalable (pas nécessaire en python).
  - **Programmation orienté objet**: basé sur des "objets" (associations de données) et de procédures agissant sur ces données.
- Utilisation C++ :
  - Très efficace et rapide. Avec le C et le fortran, il est très utilisé pour le calcul scientifique et HPC.
  - Permet facilement la définition de cadre abstrait et/ou strict.
  - Il s'agit d'un langage très portable aussi.

Le premier programme

Fichier: first_program.cpp

```
1  #include <cmath>
2  #include <iostream>
3  using namespace std;
4  int main() {
5    double x;
6    x = 0.0;
7    cout<<" >>>> Donnez un nombre "<<endl;
8    cin >> x;
9    cout<<" >>>> Racine carré: "<<sqrt(x)<<endl;
10   return 0;
11 }
```

- **Ligne 1-3**: inclusion de bibliothèques pour l'affichage et les mathématiques.
- **Ligne 4**: fonction de départ d'un programme.
- **Ligne 5**: déclaration du type d'une variable x.
- **Ligne 6**: affectation de la variable x.
- **Ligne 7 et 9**: affichage du message à l'écran avec un calcul.
- **Ligne 8**: flux d'entrée stocké dans la variable *x*.
- **Ligne 10**: fin de la fonction et renvoie un int (code erreur).

Compilation I

- Pour compiler et exécuter ce programme (méthode 1):

```
user $ g++ first_program.cpp -o run
user $ ./run
user $ >>>> Donnez un nombre
user $ 1.5
user $ >>>> Racine carré: 1.22474
```

- Pour compiler et exécuter ce programme (méthode 2):

```
user $ g++ -c first_program.cpp
user $ g++ -o run first_program.o
user $ ./run
user $ >>>> Donnez un nombre
```

- Compilation et création des fichiers ”objet” puis édition de liens à partir des fichiers objets.

Compilation II

- Pour les projets à plusieurs fichiers on utilise la deuxième méthode.
- En effet on va compiler chaque fichier de façon séparé (on parle de fichier ”objet” .o)
- Ils sont ensuite assemblés pour donner un exécutable: on parle d'édition de liens.

```
user $ g++ -c first_program.cpp
```

- Construction du fichier objet first_program.o

```
user $ g++ -o run first_program.o
```

- Edition de liens.
- **Gros projet**: on n'écrit pas les commandes une par une dans le terminal. On utilise un Makefile.

Compilation III

Makefile

```
1  CC = g++
2  EXEC_NAME = run
3  OBJ_FILES = first_program.o
4
5  all : $(EXEC_NAME)
6  clean :
7          rm $(EXEC_NAME) $(OBJ_FILES)
8
9  $(EXEC_NAME) : $(OBJ_FILES)
10         $CC) -o $(EXEC_NAME) $(OBJ_FILES)
11
12 %.o: %.cpp
13         $(CC) -o $@ -c $<
```

- Pour compiler et exécuter ce programme :

```
user $ make
user $ ./run
user $ >>>> Donnez un nombre
user $ 1.5
user $ >>>> Racine carré: 1.22474
```

Compilation IV

- Compilation simple

```
user $ g++ -c first_program.cpp -o first_program.o
user $ g++  first_program.o -o run
```

- Première étape: compilation qui génère un *.o*
- Seconde étape: édition de liens qui génère l'exécutable run.

Compilation IV

- Compilation simple

```
user $ g++ -c first_program.cpp -o first_program.o
user $ g++  first_program.o -o run
```

- Première étape: compilation qui génère un *.o*
- Seconde étape: édition de liens qui génère l'exécutable run.
- Structure de Makefile:

```
1 cible: dépendance
2         commandes
```

Compilation IV

- Compilation simple

```
user $ g++ -c first_program.cpp -o first_program.o
user $ g++  first_program.o -o run
```

- Première étape: compilation qui génère un *.o*
- Seconde étape: édition de liens qui génère l'exécutable run.
- Structure de Makefile:

```
1 cible: dépendance
2         commandes
```

- Définition des variables: compilateur, nom de l'exécutable et fichier qu'on veut construire

```
1 CC = g++
2 EXEC_NAME = run
3 OBJ_FILES = first_program.o
```

Compilation V

- Compilation des .cpp en .o (le *.o* est généré que s'il est le plus ancien):

```
1 %.o: %.cpp
2         $(CC) -o $@ -c $<
```

- Cible: les *.o*, dépendances les *.c*. Commande: construit le *.o* à l'aide du *.cpp*

Compilation V

- Compilation des .cpp en .o (le *.o* est généré que s'il est le plus ancien):

```
1 %.o: %.cpp
2         $(CC) -o $@ -c $<
```

- Cible: les *.o*, dépendances les *.c*. Commande: construit le *.o* à l'aide du *.cpp*

```
user $ g++ -o first_program.o -c first_program.cpp
```

- Analyse de la commande Makefile:
  - **$@**: La cible, ici les *.o*. **$** <: les dépendances, ici les *.cpp*

Compilation V

- Compilation des .cpp en .o (le *.o* est généré que s'il est le plus ancien):

```
1 %.o: %.cpp
2         $(CC) -o $@ -c $<
```

- Cible: les *.o*, dépendances les *.c*. Commande: construit le *.o* à l'aide du *.cpp*

```
user $ g++ -o first_program.o -c first_program.cpp
```

- Analyse de la commande Makefile:
  - **$@**: La cible, ici les *.o*. **$** <: les dépendances, ici les *.cpp*
- Edition de liens:

```
1 $(EXEC_NAME) : $(OBJ_FILES)
2         $(CC) -o $(EXEC_NAME) $(OBJ_FILES)
```

- Commande qui fait l'édition de liens de l'exécutable à partir des fichiers *.o* générés par la commande précédente.
- "**all**": commande appliquée lorsqu'on tape "make".
- "**clean**": lorsqu'on tape "make clean", elle supprime les *.o* et l'exécutable.

2. Bases: type, contrôle et boucles

Type de base

- En C++ on doit définir le type des variables.
- Types usuels:
  - **Entier**: *short int*, *int*, *long int*,
  - **Réel**: *float*, *double*, *long double*,
  - **Bouléen**: *bool*,
  - **Caractère**: *char*.
- On peut souvent définir des versions signées et non signées.
- Pour éviter la modification d'une variable on utilise **const**.

```
1  int x= 1;
2  const int y = 1;
3  x = 2;  y = x;
4  cout<<" >>>> x :"<<x<<"  y: "<<y<<endl;
```

```
user $ make
user $ base.cpp:10:5: error: cannot assign to variable 'y' with
     const-qualified type 'const int' y = x;
user $ base.cpp:8:13: note: variable 'y' declared const here const int y =
     1;
```

Expression constante I

- Variable ”const” doit être initialisée (on lui donne une valeur) à la déclaration.
- L'expression utilisée pour l'initialisation n'as pas besoin de l'être.

```
1  int p;
2  cout<<"donner un entier"<<endl;
3  cin >>p;
4  const int n=p;
5  cout<<"n >>> "<<n<<endl;
```

```
user $ make
user $ 1
user $ n >>> 1
```

- Valeur de *p* inconnue à la compilation. *y* sera fixé à l'exécution mais ne pourra plus changer ensuite.
- valeur d'une variable constante pas forcément connue à la compilation.

Expression constante II

- Amélioration C++ 11: constexpr pour les variables initialisée à la compilation.

```
1  int p;
2  cout<<"donner un entier"<<endl;
3  cin >>p;
4  constexpr int n=2*p;
5  cout<<"n >>> "<<n<<endl;
```

```
user $
     base.cpp:89:17:
     error: ...
user $  constexpr
     int n=p;
```

- ici la variable *p* ne peut pas être connue à la compilation donc pas possible.

```
1  const int p=1;
2  constexpr int n=2*p;
3  cout<<"p >>> "<<p<<endl;
```

```
user $ p >>> 2
```

- Plus généralement constexpr permet de défnir des expressions calculable à la compilation.

Déclaration automatique

- Depuis le C++11 plusieurs possibles de déclaration automatique.
- Cas 1: déclarer une variable à partir d'une autre.

```
1  int p1=1;
2  decltype (p1) q=2;
3  cout<< " print q "<<q<<endl;
```

```
user $ print q 2
```

- Cas 2: déclarer une variable de son initialisation

```
1  auto xa=1;
2  auto ya=2.5;
3  cout<<" type xa "<<typeid(xa).name()<<endl;
4  cout<<" type ya "<<typeid(ya).name()<<endl;
```

```
user $ type xa i
user $ type ya d
```

- Risque d'erreur plus grand. Si vous voulez avoir un double mais initialiser avec 1 et non 1.0 vous aurez un entier.

Opérateurs et conversion de type

- Un opérateur binaire agi sur 2 variables d'un certain type (ils peuvent être différent) et renvoie un résultat
  - **Arithmétique**: +, -, /, *, % (modulo).
  - **Relationnels**: == (égal à), != (différent de), <, >, =>, =<.
  - **Logique**: || (ou), && (et), ! (non).
  - **Affectation**: = permet d'affecter une valeur dans une variable.
  - **Incrémental**: ++ (équivaut à +1), −− (équivaut à -1).
- Les opérateurs (arithmétique, affectation etc) font des conversions de type.

```
1  int x1= 1, x2 =0;
2  double z1 = 1.6, z2 =0.0;
3  z2 = x1;   x2 = z1;
4  x1 = z1 +x2;
5  cout<<" >>>> x2 :"<<x2<<"  z2: "<<z2<<" x1: "<<x1<<endl;
```

```
user $ ./run
user $ >>>> x2 :1  z2: 1 x1: 2
```

- **Problème**: conversion accidentelle de réel en entier. Troncature implicite.

Opérateurs et conversion de type

- Un opérateur binaire agi sur 2 variables d'un certain type (ils peuvent être différent) et renvoie un résultat
  - **Arithmétique**: +, -, /, *, % (modulo).
  - **Relationnels**: == (égal à), != (différent de), <, >, =>, =<.
  - **Logique**: || (ou), && (et), ! (non).
  - **Affectation**: = permet d'affecter une valeur dans une variable.
  - **Incrémental**: ++ (équivaut à +1), −− (équivaut à -1).
- Les opérateurs (arithmétique, affectation etc) font des conversions de type.

```
1  int x1= 1, x2 =0;
2  double z1 = 1.6, z2 =0.0;
3  z2 = x1;   x2 = z1;
4  x1 = z1 +x2;
5  cout<<" >>>> x2 :"<<x2<<"  z2: "<<z2<<" x1: "<<x1<<endl;
```

- On peut expliciter et forcer une conversion:

```
1 z = (double) (x1/x2);
```

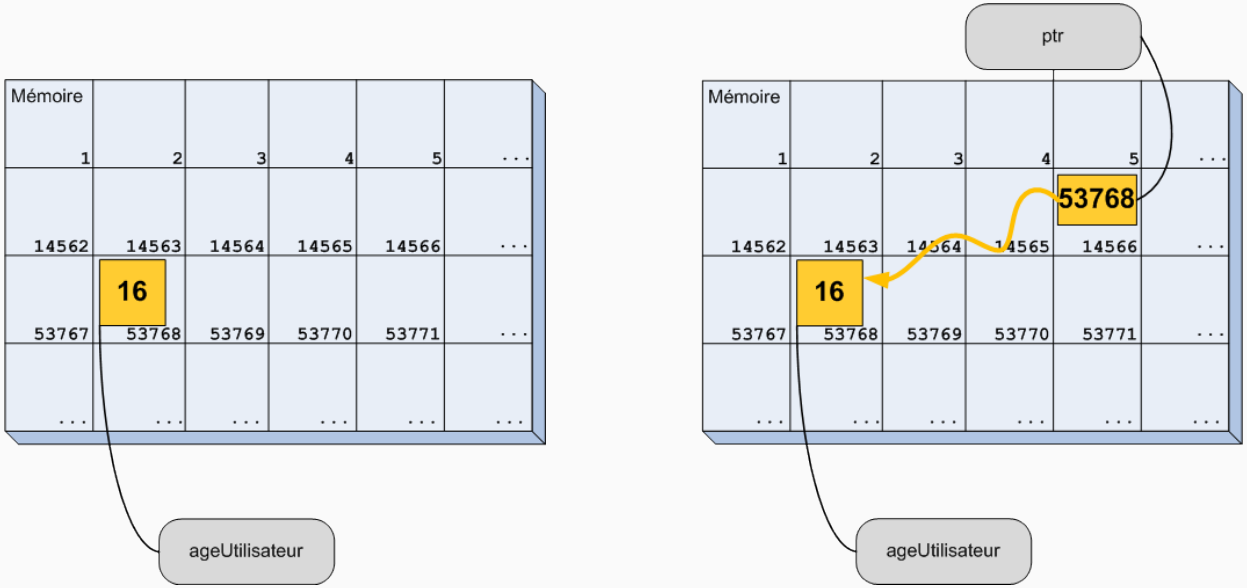
Pointeur

- Pointeur: variable contenant l'adresse mémoire d'une variable. Exemple:

```
1 int * p;  int n =10;
2 p= &n;  *p = 20;
3 cout<<">> p "<<p<<" *p "<<*p<<endl;
```

```
user $ >> p
       :0x7ffeea6a0944
       *p :20
```

- "*" permet d'accéder à la valeur d'une variable associée à un pointeur,
- "" permet d'accéder à l'adresse d'une variable.



Source: OpenClassRoom

Pointeur II

- Avec ”Auto” vous pouvez directement construire une pointeur sur une variable sans vous préoccuper du type.

```
1  int n =10;
2  auto p = &n;
```

- La valeur nulle des pointeurs est nullptr.

```
1  int *p = nullptr
```

- Règles d’affectation/ comparaison:
  - On peut affecter un pointeur dans un autre si, il sont de même type,
  - deux pointeurs sont égaux s’ ils pointent sur la même adresse.
- On peut convertir des pointeurs par une opération de **cast**:

```
1  int *p = nullptr; double *p2 =nullptr;
2  p = static_cast <int *>(p2);
```

- ici on convertit un pointeur sur un double en un pointeur sur un entier.

Pointeur III

- Un pointeur = adresse et type. Parfois le type peut varier. Que faire ?
- On utilise des **pointeurs génériques**

```
1 void *p; int n=10; int *q=&n;
2 p=q;
3 cout<<"adresse "<<p<<" "<<q<<endl;
```

```
user $ >> adresse 0x7ffd8e6eb2b4 0x7ffd8e6eb2b4
```

```
1 void *p; int n=10; int *q=&n;
2 p=q;
3 cout<<"valeur "<<*p<<" "<<*q<<endl;
```

```
user $ main.cpp:19:23: error: void* is not a pointer-to-object type
```

- Peut être utile dans une fonction lorsqu'on connaît pas le type de l'objet.
- On peut le convertir avec un static cast.

Pointeur et constance

- Cas 1

```
1 int n=12; const int * p = &n;
2 p=13;
```

- Ici *p* pointe sur un entier constant. On ne peut pas modifier *n* à travers *p*.

```
user $ main.cpp:17:7: error: assignment of read-only location * p
```

- Cas 2

```
1 int n=12, m=13; int * const p = &n;
2 p=&m;
```

- Ici *p* est constant. Donc *p* ne peut pas pointer sur un autre entier.

```
user $ main.cpp:17:7: error: assignment of read-only location * p
```

Instructions de choix

- Dans de nombreux problèmes il est essentiel de décrire des choix.
- **Exemple**: différentier un calcul selon le signe d'un nombre.

- Instruction **if**:

```
1 double a = 0.0;
2 if (a > 0.0) {
3         cout<<" >>> a est un nombre positif "<<endl; }
4 else if (a < 0.0) {
5         cout<<" >>> a est un nombre négatif "<<endl; }
6 else     {
7         cout<<" >>> a est nul "<<endl; }
```

```
user $ ./run
user $ >>> a est nul
```

Instructions de choix

- Dans de nombreux problèmes il est essentiel de décrire des choix.
- **Exemple**: différentier un calcul selon le signe d'un nombre.
- Autre possibilité: l'instruction **switch**.

```
1 int b = 2;
2   switch(b) {
3   case 0: cout<<" >>> b vaut 0"<<endl;
4     break;
5   case 1: cout<<" >>> b vaut 1"<<endl;
6     break;
7   case 2: cout<<" >>> b vaut 2"<<endl;
8     break;
9  }
```

```
user $ ./run
user $ >>> b vaut 2
```

- **Remarque**: les instructions conditionnelles sont légèrement plus couteuses. A éviter en très grande quantité.

Boucles I

- **Boucle FOR**: permet de réitérer une série d'instructions. Exemple:

```
1 for(int i=0; i<2; i++) {
2         cout<<i<<" fois "<<endl;
3  }
```

```
user $ 0 fois
user $ 1 fois
```

- L'instruction dans la boucle est réalisée tant que "i" est inférieur à 2 (testé en avant l'instruction). "i++" veut dire qu'à la fin du bloc, *i* augmente de 1.
- Exemple 2: série entière pour calculer l'exponentiel.

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

```
1 int n =0, m = 20;
2 long int nfac = 1.0;
3 double xn = 1.0, res =0.0, res0 = 0.0;
4   for(n=1; n<m; n++) {
5     res = res+ xn/nfac;
6     nfac = nfac * n; xn = xn*x;
7     if(abs(res-res0) <0.000001) {  break; }
8     res0=res;
9   }
```

Boucles II

```
1 cout<<" func "<<expo(0.42)<<" ref "<<exp(0.42)<<endl;
2 cout<<" func "<<expo(85.42)<<" ref "<<exp(85.42)<<endl;
```

```
user $ func 1.52196 ref 1.52196
user $ func 1.18692e+10 ref 1.77103e+11
```

- Dans le code précédent le mot-clé "break" permet de quitter la boucle.
- Ici: une fois le critère de convergence réalisée, on arrête la boucle.
- Autres mots-clés: "continue" permet de sauter l'itération courante.
- Exemple: si $i == j$ on fait rien, on passe à l'itération suivante.

$$\sum_i \sum_{j,i\neq j} i+j$$

```
1 double res=0;
2 for (int i = 0; i < 10; i++) {
3   for (int j = 0;j  < 10; j++) {
4     if (i == j) continue;
5       res = res + i+j;  }
6 }
```

Boucles III

- Il existe d'autres types de "boucle".
- **Boucle WHILE**: Boucle avec un critère d'arrêt.

```
1   int n =1, m = 20;
2   long int nfac = 1.0;
3   double xn = 1.0;
4   double res =0.0,res0 = 1.0;
5   while(abs(res-res0) >0.000001) {
6     res0=res;
7     res = res+ xn/nfac;
8     nfac = nfac * n;
9     xn = xn*x;
10    n++;
11  }
```

- La boucle continue tant qu'un critère est respecté.
- Il existe aussi la boucle **do ... while**.

Remarque

- **Remarque**: il est essentiel de toujours affecter une valeur initiale aux variables.

```
1  int n =1, m = 20;
2  long int nfac = 1.0;
3  double xn = 1.0, res =0.0, res0 = 1.0;
4  while(abs(res-res0) >0.000001) {
5    res0=res;
6    res = res+ xn/nfac;
7    nfac = nfac * n;
8    xn = xn*x;
9    n++;
10 }
```

- Si *res* est différent de 0 au début, la somme finale peut être totalement fausse.
- Dans certains cas (ca dépend de la complexité du code, du compilateur) les variables ne sont pas initialisées a zéro.
- Source de bug très fréquente.

3. Bases: fonctions et tableaux

Tableaux I: cas statique

- Les **tableaux** de nombres, caractères etc, sont un élément essentiel de la programmation C/C++.
- **Exemple 1**: les tableaux simples statiques

```
1 double tab[10];
2 for (int i = 0; i <10; i++}{
3   x = x+tab[i];
4 }
```

- [] permet d'accéder à une case du tableau.
- Les tableaux en C/C++ vont de zero à size-1.
- **Exemple 2**: les tableaux doubles statiques

```
1 double tab[10][15];
2 for (int i = 0; i <10; i++}{
3    for (int j = 0; j <15; j++}{
4          x = x+tab[i][j];
5    }
6 }
```

Tableaux II: pointeur et tableau

- **Remarque**: le nom du tableau est en fait un pointeur sur la première case mémoire du tableau.

```
1 double t[10];
2 cout<<"t : "<<t<<" &t[0] :"<<&t[0]<<endl;
```

```
user $  t : 0x7ffee98fe950 &t[0] :0x7ffee98fe950
```

- **Allocation statique**: valide pendant tout le bloc d'instruction (programme/fonction).
- L'allocation dynamique permet d'allouer et libérer la mémoire de façon explicite dans le programme.
- **Opérateurs**: new et delete pour allouer/libérer la mémoire.

```
1 double * tab = new double[10]; double res=0.0;
2 for (int  i=0;i<10;i++){  tab[i] = i;  }
3 for (int  i=0;i<10;i++){  res = res + tab[i];  }
```

Tableaux III: pointeur et tableau

- **Remarque**: le new alloué de la mémoire et affecte dans le pointeur ”tab” l’adresse de la première case mémoire.
- La mémoire est allouée pendant tout le programme mais le pointeur pendant le bloc d’instruction (programme/fonction).
- **Libération mémoire**: delete

```
1 double * tab = new double[10];
2 double res=0.0;
3 for (int  i=0;i<10;i++){  tab[i] = i;  }
4 delete [] tab;
```

Important: il faut toujours libérer la mémoire à la fin d’une fonction ou d’un code une fois plus utilisée.

- **Remarque**: Eviter un trop grand nombre d’allocations mémoire. Mieux vaut faire une grosse allocation au début que plein de petites allocations.
- Passage en argument de fonction:

Tableaux en dimension superieur I

- Cas statique: cas pour les tableaux 1D

```
1  double tab2D_1[2][2];
2   tab2D_1[0][0]=1.0, tab2D_1[0][1]=0.0;
3   tab2D_1[1][0]=0.0, tab2D_1[1][1]=1.0;
4   cout<<"Matrix >> i "<<0<<" j "<<1<<" >>>> "<<tab2D_1[0][1]<<endl;
```

- Cas dynamique:
  - On peut voir un tableau 2D comme un tableau 1D contenant des tableaux 1D.
  - On va appliquer cela dans le cas dynamique.

```
1   double   ** tab2D_2 = NULL;
2   tab2D_2 = new double*[2];
3   for(int i =0; i<2;i++){
4     tab2D_2[i] = new double[2]; }
```

  - On peut le lire comme un pointeur sur des pointeurs de double.
  - On crée un tableau avec le 1er new contenant des pointeurs sur des doubles.
  - Chaque pointeur de double contient un tableau de double.
  - On utilise les tableaux dynamiques de la même façon.

Tableaux en dimension superieur II

- Initialization tableau 1D:

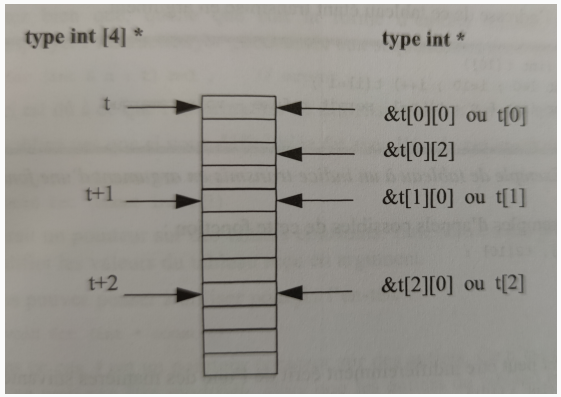
```
1   double tab[3]= {1.0,2.0,3.0};
```

- Initialization tableau 2D:

```
1   double tab[3][2]= { {1.0,2.0}, {1.0,-1.0}, {5.0,2.0}};
```

```
1   double tab[3][2]= { {1.0,2.0, 1.0,-1.0,5.0,2.0};
```

- Rangement en mémoire important dans le second cas.



Source: livre ”programmer en c++ moderne”

Tableaux en dimension superieur III

- **Défauts des tableaux 2D**: plusieurs allocations mémoire(à chaque ”new”).
- Il n’est pas obligatoire que la mémoire allouée soit .
- Un tableau 2D peut être plus coûteux (en temps de calcul) à parcourir/utiliser à cause des allers-retours en mémoire.
- **Possible solution**: Un tableau 1D simulant un tableau 2D:

```
1  int index_ij(int n,int i,int j){
2    int k = i * n +j;
3    return k;
4  }
5  double *tab2D_3 = new double[4];
6
7  tab2D_3[index_ij(2,0,0)]=1.0;
8  tab2D_3[index_ij(2,0,1)]=0.0;
9  tab2D_3[index_ij(2,1,0)]=0.0;
10 tab2D_3[index_ij(2,1,1)]=1.0;
11 cout<<"Ma 3>> i "<<0<<" j "<<1<<" "<<tab2D_3[index_ij(2,0,1)]<<endl;
```

Chaine de caractère

- Le C/C++ permet de manipuler des caractères (char) et des chaines de caractère qui peut être vu comme un tableau de char.
- En mémoire: n caractère ==> n+1 octet (code nul à la fin en plus).
- Chaine constante comme ”bonjour”.

```
1 char * mot=NULL;
2 mot = "bonjour";
3 cout<<" mot >>>"<<mot<<endl;
```

- Lecture/écriture:

```
1 char * phrase= new char[30];
2 cin >> phrase;
3 strcat(phrase,"monsieur");
4 cout<<" Phrase"<<<endl;
```

```
user $  >>> Donner un mot
user $ bonjour
user $ >>> Phrase: bonjour monsieur
```

4. Applications

Exemple: jeu de dé I

- On veut générer des nombres et des processus aléatoires. Librairie: **std**.
- Jeu de dés contre l'ordinateur. Victoire quand le nombre de lancé gagné est supérieur à celui de l'IA.

```
1  int de1, de2, de3;
2    int g1=0,g2=0, i=1;
3    char c;
4    srand(6);
5    while (g1 <= g2) {
6      cout<<" Voulez vous continuer a jouer (y/n) "<<endl;
7      cin >> c;
8      if( c == 'y') {
9          de1 = (rand() % (6 -0 + 1));
10         de2 = (rand() % (6 -0 + 1));
11         cout<<"Lancer :"<<i<<" joueur : "<<de1<<" IA : "<<de2<<endl;
12         if (de1 > de2 ) { g1++; }
13         if (de1 < de2 ) { g2++; }
14     }
15     else { break; }
16     i++;
17   }
18   cout<<" Victoire "<<endl;
```

Exemple: jeu de dé II

- ”rand” génère un nombre aléatoire. Pour avoir un nombre en MIN and MAX:

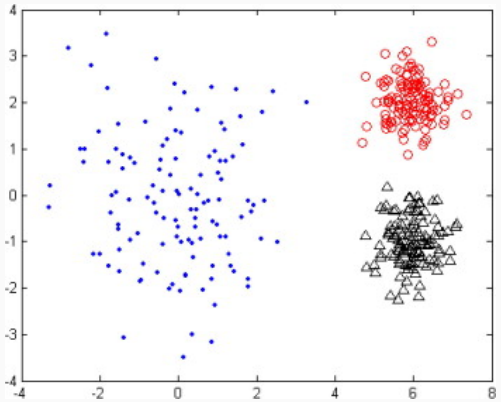
$$rand()\%(\text{MAX} - \text{MIN} + \text{MIN}))$$

```
user $ Voulez vous continuer a jouer (y/n)
user $ y
user $ Lancer :1 joueur 1: 0 joueur 2: 0
user $ voulez vous continuer a jouer (y/n)
user $ y
user $ Lancer :2 joueur 1: 1 joueur 2: 6
user $ voulez vous continuer a jouer (y/n)
user $ y
user $ Lancer :3 joueur 1: 5 joueur 2: 0
user $ voulez vous continuer a jouer (y/n)
user $ y
user $ Lancer :4 joueur 1: 5 joueur 2: 2
user $ Victoire
```

- ”srand(x)” permet de fixer la graine. La liste des tirages aléatoire est la même à chaque exécution. En changeant ”x” on change la liste.

Exemple: Algorithme de K-moyenne et apprentissage I

- **K-moyennes**: algorithme simple d'apprentissage non supervisé plus précisément un algorithme de "clustering".
- L'apprentissage non supervisé va plutôt trouver des patterns dans les données. Notamment, en regroupant les choses qui se ressemblent.
- **But**: partitionner un nuage de points en plusieurs groupes/catégories.
- **Mathématiques**: Etant donné un ensemble de n points $(\mathbf{x}_1, .., \mathbf{x}_n)$ on cherche k ensemble $(S_1, ..S_k)$ tel que la distance entre les points de chaque ensemble et son barycentre soit minimum.



Exemple: Algorithme de K-moyenne et apprentissage II

- Choisir k points qui représentent la position moyenne des partitions $\mathbf{m}_1^0, \ldots, \mathbf{m}_k^0$ initiales (au hasard par exemple).
- Répéter jusqu'à ce qu'il y ait convergence (iter m:
  - affecter chaque observation à la partition la plus proche

$$S_i^m = \left\{ \mathbf{x}_j, \| \mathbf{x}_j - \mathbf{m}_i^0 \| \leq \| \mathbf{x}_j - \mathbf{m}_l^0 \| \right\}$$

  - Mettre à jour la position moyenne des partitions

$$\mathbf{m}_i^m = \frac{1}{\mid S_i^m \mid} \sum \mathbf{x}_j$$

  - tester la convergence $\sum_i^k \sum_{j \in S_i} \| \mathbf{x}_j - \mathbf{m}_i^m \|^2 \leq \epsilon$
- **Application**: Regrouper les 50 plus grandes villes de France en 12 régions.
- Très sensible à l'initialisation.

Exemple: Algorithme de K-moyenne et apprentissage III

- On utilise des tableaux 1D donc:

```
1  int index_ij(int i,int j){
2    int res = i * 2 + j;
3    return res; }
```

- On initialise les centres de partitions avec les 1ères villes.
- Construction des partitions:

```
1  for(int j =0; j < 50;j++){
2    xj=villes[index_ij(j,0)];
3    yj=villes[index_ij(j,1)];
4    minimum=dist(xj,yj,centres[index_ij(0,0)],centres[index_ij(0,1)]);
5    for(int i =1; i < nb_part;i++){
6      res=dist(xj,yj,centres[index_ij(i,0)],centres[index_ij(i,1)]);
7      if(res < minimum){
8        minimum = res;
9        villes_vers_part[j]=i;}
10       } }
```

- On calcule la distance au 1er centre. Puis la distance aux autres centres. Si une autre distance est plus petite elle devient le minimum.

Exemple: Algorithme de K-moyenne et apprentissage IV

- Calcul des positions des centres:

```
1  for(int i =0; i < nb_part;i++){
2     count = 0;
3     centres[index_ij(i,0)]= 0.0; centres[index_ij(i,1)]= 0.0;
4     for(int j =0; j < 50;j++){
5       if(villes_vers_part[j]==i){
6         centres[index_ij(i,0)]+= villes[index_ij(j,0)];
7         centres[index_ij(i,1)]+= villes[index_ij(j,1)];
8         count = count +1;}
9     }
10    centres[index_ij(i,0)]/=count;
11    centres[index_ij(i,1)]/=count;}
```

Exemple: Algorithme de K-moyenne et apprentissage V

- On prend que les villes qui appartiennent à la partition associée. on compte en même temps le nb de villes dans la partition. Calcul de l'erreur:

```
1 for(int i =0; i < nb_part;i++){
2   cix = centres[index_ij(i,0)];  ciy = centres[index_ij(i,1)];
3   for(int j =0; j < 50;j++){
4     if(villes_vers_part[j]==i){
5       xj=villes[index_ij(j,0)]; yj=villes[index_ij(j,1)];
6       erreur+=pow(dist(xj,yj,cix,ciy),2.0); }
7    }
8    erreur = erreur/nb_part; }
```

5. Remarques: organisation/verification

Fichiers CPP/HPP et dossier

- **1er remarque**: il est important de documenter son code.
- **Organisation classique d'un code**:
  - dossier **doc** qui contient la documentation.
  - dossier **src** (pour source) qui contient les fichiers .cpp, .c et c
  - dossier **build** ou en général on compile et exécute le projet.
  - dossiers **includes** et **lib** pour les librairies extérieures.
- Les définitions des classes et les en-têtes de fonctions sont en général dans les fichiers ".hpp". Le corps des fonctions/méthodes dans les fichiers .cpp

Tests unitaires

- Un projet important nécessite:
  - beaucoup de fonctions, classes et différents fichiers,
  - plusieurs développeurs.
- Le recherche de bug (suite à des modifications ou pas) peut être compliqué.
- **Première chose à faire**: les tests unitaires.
- **Principe**: ajouter des petits tests dans chaque portion de code (classes, grosses fonctions etc) qu'on peut régulièrement exécuter.
- Exemple:
  - Un projet est géré sur GIT avec plusieurs développeurs.
  - Chaque modification est "pusher" sur la branche principal que si les tests unitaires marchent.
- **Remarque**: on donnera des exemples par la suite.