TP1 : Introduction au C++

Préambule

Qu'est-ce que le C++ ?

Le C++ est un langage très populaire qui allie puissance et rapidité. Voici quelques qualités du C++ que nous découvrirons au cours de ces TP :

- Il est très **répandu**. C'est un des langages de programmation les plus utilisés internationalement dans l'industrie et la recherche. Il y a donc beaucoup de documentation sur Internet et on peut facilement trouver de l'aide sur les forums.
- Il est très **rapide** (en temps de calcul) et est donc utilisé pour les applications qui ont besoin de performance (jeux vidéo, outils financiers, simulations numériques ...)
- Il est **portable**, c'est-à-dire qu'un même code source peut être transformé rapidement en exécutable sous Windows, Mac OS et Linux.
- Il existe de nombreuses bibliothèques pour le C++, nous en verrons quelques unes dans ces TP. Les bibliothèques sont des extensions pour le langage car de base, le C++ ne fournit que des outils bas-niveau mais en le combinant avec de bonnes bibliothèques, on peut créer des programmes très puissants.
- Il est **multi-paradigmes**, c'est-à-dire qu'on peut programmer de différentes façons en C++. La plus célèbre est la : **Programmation Orientée Objet (POO)**. Cette technique permet de simplifier l'organisation du code dans les programmes et de rendre facilement certains morceaux de codes réutilisables. Plusieurs TP y seront consacrés.

Quelques liens utiles

- Un cours très bien construit :
  https://openclassrooms.com/fr/courses/1894236-programmez-avec-le-langage-c
- Un polycopié (écrit par K. Santugini)
  http://www.math.u-bordeaux.fr/ ksantugi/downloads/CPlusPlus/PolyCxx.pdf
- Quelques sites web :
  http://stackoverflow.com/, http://www.cplusplus.com, http://cpp.developpez.com

Pour la compilation et l'exécution

Pour compiler un fichier main.cc :

```
g++ -std=c++11 -o run main.cc
```

L'exécutable créé ainsi s'appelle run, on le lancera en tapant

```
./run
```

Pour compiler plusieurs fichiers :

```
g++ -std=c++11 -o run TP1.cc main.cc
```

Quelques conseils

Bien lire la console quand vous avez des erreurs à la compilation. En effet le compilateur vous donne de nombreuses informations pour vous aider à corriger la syntaxe de votre code. Un exemple :

```
main.cc:20:35: error: expected ';' after expression
    cout << "Hello world!" << endl
                                  ^
                                  ;
1 error generated.
```

Le compilateur nous informe que l'erreur est dans le fichier « main.cc » à la ligne 20 et à la colonne 35. Ensuite il nous informe qu'il attend un « ; » à la fin de la ligne :

```
cout << "Hello world!" << endl
```

En effet, chaque ligne de commande en C++ se termine par un point virgule.

N'hésitez pas à **commenter votre code**, cela facilitera la recherche d'erreurs. Pour insérer un commentaire en C++ sur une ligne :

```
// Ceci est un commentaire
```

et sur plusieurs lignes :

```
/* Ceci est un commentaire
sur plusieurs lignes */
```

Pensez à bien **indenter** votre code pour faciliter sa lecture. Avec *Atom* (éditeur qui est conseillé), vous pouvez faire :
Edit > Lines > Auto Indent

Pour ce premier TP, de nombreux programmes vous sont donnés, n'hésitez pas à **modifier et/ou enlever** certaines lignes pour voir le rôle de chaque bout de code.

Objectifs

- Présenter les différents types de données en C++ (int ; double ; . . .)
- Utiliser la librairie *iostream* pour afficher des messages à l'écran : *cout* et interagir avec l'utilisateur : *cin*
- Découvrir les structures de contrôle classiques : boucle for, conditions if et while
- Écrire des fonctions
- Manipuler les vecteurs
- Lire et écrire des fichiers
- Passer des arguments (par valeur ou par référence)
- Structurer son code (.cc, .h, .cpp)
- Découvrir les pointeurs

Exercices

Exercice 1 - Premier programme

1. Créer un fichier main.cc et insérer le code suivant :

```
1  /* Chargement du fichier << iostream >> qui est une bibliothèque
2  d'affichage de messages à l'écran dans une console */
3  #include <iostream>
4
5  // Choisir parmi différentes bibliothèques celles que l'on veut
6  using namespace std;
7
8  /* C'est ici que commence vraiment le programme. Les programmes sont
9  essentiellement constitués de fonctions. Chaque fonction a un rôle et peut
10 appeler d'autres fonctions pour effectuer certaines actions.
11 Tous les programmes possèdent une fonction << main >>. C'est donc la fonction
12 principale. Elle renvoie toujours un entier d'où la présence du << int >>
13 devant le main. */
14 int main()
15 {
16     /* Première ligne composée de 3 éléments qui fait quelque chose de concret :
17     1. cout : affichage d'un message à l'écran
18     2. "Hello world!" : le message à afficher
19     3. endl : retour à la ligne dans la console. */
20     cout << "Hello world!" << endl;
21
22     /* Ce type d'instruction clôt généralement les fonctions. Ici, la fonction
23     main renvoie 0 pour indiquer que tout s'est bien passé (toute valeur
24     différente de 0 aurait indiqué un problème). */
25     return 0;
26 }
```

Le compiler et l'exécuter. Vous devez voir s'afficher dans le terminal :

```
1  Hello world!
```

2. Retirer le point virgule à la fin de la ligne 20. Compiler de nouveau et étudier la réaction du compilateur. Rajouter le point virgule.

3. Commenter la ligne 3. Remarquer que les commandes "cout" et "endl" ne sont plus reconnues. Enlever le commentaire.

4. Commenter la ligne 6. Remarquer que les commandes "cout" et "endl" ne sont plus reconnues. Cependant le compilateur vous propose à la place :

```
1  std::cout
2  std::endl
```

Essayer et conclure sur le rôle de cette ligne.

Exercice 2 - Les types

Une variable est une information stockée en mémoire. Il en existe différents types en fonction de la nature de l'information à stocker : **string** (texte), **int** (entier), **unsigned int** (entier positif ou nul), , **long int** (entier encodé sur 64 bits), **double** (réel encodé sur 64 bits), **float** (réel encodé sur 32 bits) **bool** (true or false). Une variable doit être définie (une valeur doit lui être affectée) avant son utilisation. La valeur d'une variable peut être affichée avec cout. Les variables peuvent être déclarées (à peu près) n'importe où dans le code.

1. Tester le programme suivant.

```
#include <iostream>
//Pour utiliser les fonctions racine (sqrt) et puissance (pow)
#include <cmath>
using namespace std;
int main()
{
        cout.precision(15); // pour afficher une précision à 15 chiffres
        double a(2.125878159992178711); // ou double a = 2.125878159992178711;
        float f(a); // ou float f = a;
        cout << "En double " << a << " versus en float " << f << endl;

        string nomFonction("racine");
        cout << nomFonction << "(" << a << ") = " << sqrt(a) << endl;

        int n(3); // ou int n = 3;
        cout  <<  a  << "^" << n << " = " << pow(a,n) << endl;
        return 0;
}
```

Remarque importante :
Les 3 bouts de code ci-dessous sont équivalents :

```
double a(2.);
```

```
double a = 2.;
```

```
double a;
a = 2.;
```

Par contre, vous ne pouvez pas faire :

```
double a();
a = 2.;
```

2. Il est bien sûr possible d'additionner, de soustraire (etc ...) avec le C++. Rajouter les lignes suivantes au code précédent. Tester le bout de code suivant :

```
double b(2.), c(3.), d(2.1); // (b = 2)
b++; // Ajoute la valeur 1 : cette commande a donné son nom au C++ ! (b = 3)
b += c; // Ajouter la valeur c à b (b = 6)
b -= d; // Retire la valeur d à b (b = 3.9)
cout << "Après de nombreux calculs, b vaut " << b << endl;
```

Essayer le code suivant :

```
int e(2), f(3);

int g = e/f;

cout << "La division de " << e << " par " << f << " vaut " << g << "." << endl;
```

Êtes-vous satisfaits du résultat ? Que se passe t-il si e, f et g sont des doubles ?

3. Il est aussi possible de demander à l'utilisateur de remplir une variable. Pour cela vous aurez besoin d'utiliser "cin" qui contrairement à "cout" permet de faire rentrer des informations. Par exemple pour demander l'âge de l'utilisateur, essayer les lignes suivantes :

```
int age(0);
cin >> age;
```

Coder pour obtenir sur la console ce genre de résultat :

```
Quel age avez-vous ?
23
Vous avez 23 ans !
```

Que se passe t-il si l'utilisateur ne rentre pas un entier ?

4. Les variables peuvent être considérées comme des cases mémoire avec une étiquette portant leurs noms, comme dans la figure ci-dessous.

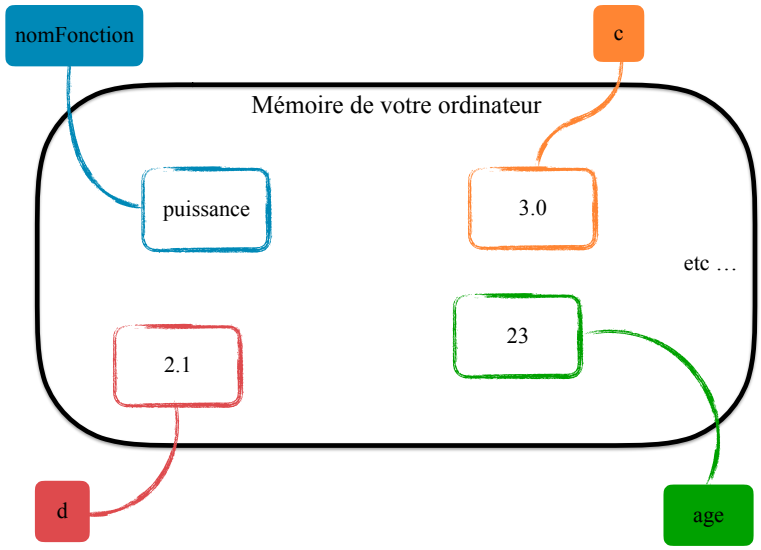

FIGURE 1 – Schéma de l'état de la mémoire après plusieurs déclarations

En C++, il est possible de coller une deuxième étiquette à une case mémoire à l'aide du symbole "&". On obtient alors un deuxième moyen d'accéder à la même case mémoire. Il s'agit d'une **référence**. En rajoutant ces quelques lignes :

```
int& laVariable(age);
cout << "Vous avez " << laVariable << " ans ! (par référence)" << endl;
```

Voilà à quoi ressemble la mémoire :

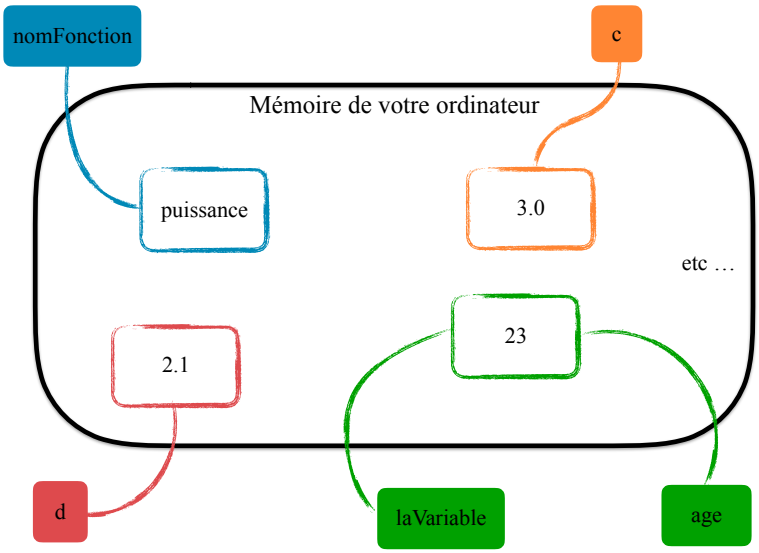

FIGURE 2 – Nouveau schéma de l'état de la mémoire

Bien sûr vous obtenez le même résultat que précédemment. On verra plus tard le rôle de ces références.

Exercice 3 - Et si la variable ne doit pas être modifiée ?

Il y a des variables qui ne sont **jamais** modifiées dans un code. Pour déclarer une constante, on ajoute le mot-clé *const* devant le type :

```
1 const string my_name = "Gertrude";
2 const int my_age = 53;
3 const double my_size = 1.62;
```

1. Réfléchir aux variables que l'on souhaite constantes dans l'implémentation d'un schéma en temps pour résoudre une EDO.
2. Jouer avec le code pour voir comment le *const* fonctionne (compiler, exécuter puis essayer de nouveau de compiler en décommentant la ligne 4) :

```
1 const string my_name = "Gertrude";
2 cout << "Je m'appelle " << my_name << " et j'aimerai pouvoir changer " << endl;
3 cout << "mais il semblerait que cela soit impossible ...." << endl;
4 // my_name = "Alice";
```

Il faut toujours déclarer avec *const* tout ce qui peut l'être. Cela permet :

- d'éviter des erreurs d'inattention lorsque l'on programme,
- de créer un programme plus efficace.

Exercice 4 - Les structures de contrôle

Le programme suivant vous montre la structure de la condition "if" et de la boucle "for" en C++.

```
1  #include <iostream>
2  #include <ctime>
3  #include <cstdlib>
4  using namespace std;
5  int main()
6  {
7    const int nb_essai(10);
8    srand(time(0));
9    const int nombre = (rand() % 100);
10   int proposition(0), est_gagnant(nb_essai+1);
11   cout << "Le nombre caché est un entier compris entre 0 et 100." << endl;
12   cout << "Vous avez " << nb_essai << " essais pour le trouver." << endl;
13
14   for (int essai = 1 ; essai <= nb_essai ; essai++)
15   {
16     cout << "Essai : " << essai << " : votre proposition est : ";
17     cin >> proposition;
18     if (proposition == nombre)
19     {
20       est_gagnant = essai;
21       break;
22     }
23     else if (proposition > nombre)
24     {
25       cout << "Moins" << endl;
26     }
27     else
28     {
29       cout << "Plus" << endl;
30     }
31   }
32   if(est_gagnant <= nb_essai)
33   {
34     if (est_gagnant == 1)
35     {
36       cout << "Vous êtes un devin : vous avez trouvé du premier coup !" << endl;
37     }
38     else
39     {
40       cout << "Bravo! Vous avez trouvé en " << est_gagnant << " essais." << endl;
41     }
42   }
43
44   return 0;
45 }
```

1. Tester le code.
2. Que se passe t-il si vous commentez la ligne 8 ?
3. Quel est le rôle du const à la ligne 7 ? Que se passe t-il si vous mettez un const devant la ligne 10 ?
4. Modifier le code en mettant une boucle while pour permettre de rejouer tant qu'on n'a pas gagné.

Exercice 5 - Les fonctions

La définition des fonctions est toujours la suivante :

```
type nomFonction(arguments)
{
        // Écrire ici ce que la fonction doit faire
}
```

Une fonction est composée de 3 éléments :

- Le premier indique le **type de donnée** qui est renvoyée par la fonction :

  int, double, string, void (= vide, quand elle ne renvoie rien) ...

Pour renvoyer une valeur, nous utilisons la commande "return" comme pour la fonction main().

- Le deuxième est le **nom de la fonction**.
- Entre les parenthèses, on trouve la **liste des paramètres** de la fonction. Il peut y avoir :

  aucun argument comme pour la fonction main() vue ci-dessus
  ou plusieurs comme pour la fonction puissance pow(double, int) (Ex 2, Q 1)

Les accolades délimitent le contenu de la fonction. Toutes les opérations qui seront effectuées se trouvent entre les deux accolades.

1. Faire une fonction *print_ hello* qui affiche "Hello world !" et l'appeler dans la fonction main.

2. Faire une fonction *fibonacci* qui calcule la suite de Fibonacci : $u_{n+1} = u_n + u_{n-1}$. Si u_0 et u_1 sont des entiers, u_n reste un entier pour tout n, le prototype de la fonction sera alors le suivant :

```
int nomFonction(int n, int u0, int u1)
{
        int un(0);
                .
                .
                .
        return un;
}
```

Pour $u_0 = u_1 = 1$, calculer u_{10}, u_{30} et u_{50}. Que se passe t-il ? Essayer avec *long int* au lieu de *int*. Conclure. Peut-on ici utiliser *long unsigned int* ?

Exercice 6 - Passage des arguments (par valeur ou par référence)

1. Par défaut, les arguments d'une fonction sont passés par valeur :

```
#include <iostream>
using namespace std;

void valeur(int b)
{
  b=5;
}

int main()
{
  int a=3;
  cout << "a avant la fonction valeur " << a << endl;
  valeur(a);
  cout << "a après la fonction valeur " << a << endl;

  return 0;
}
```

Compiler et exécuter ce code. Que se passe t-il ?

2. Comme vous avez pu le voir, le nombre imprimé est 3, et non 5 ! La fonction valeur a reçu en argument une copie de a et ne change que la valeur de cette copie. Dans la fonction main, la valeur de a ne change pas. Il est cependant possible de passer les arguments par référence. Elles ont été introduites dans l'exercice 2 à la question 4. Pour les utiliser, il suffit de rajouter le symbole "&" dans le type de l'argument. Tester cette nouvelle fonction :

```
void reference(int& b)
{
  b=5;
}
```

Cette fois-ci, la console imprime 5 à l'écran car le passage a eu lieu par référence.
Remarque : Une référence n'a de sens que s'il y a une variable en mémoire derrière :

```
reference(&(3)); //Cela ne compile pas car cela n'a aucun sens.
```

3. Pour que le code soit plus efficace, on peut aussi passer par référence de gros objets tout en déclarant que leur valeur ne doit pas être modifiée. Pour le moment la notion d'objet peut vous sembler lointaine (TP 3) mais par exemple si vous souhaitez envoyer un email avec une photo de 4Mo, le téléchargement va prendre du temps alors que si vous envoyez l'adresse web de cette image, c'est beaucoup plus rapide ! Quand il s'agit juste de faire passer cet objet et qu'il ne doit pas être modifié, on rajoute le mot-clef const devant le type de l'argument. Cela permet au compilateur de mieux vérifier le code et aide à trouver les bugs. C'est une très bonne habitude à prendre pour "mieux programmer". Tester pour voir l'erreur que renvoie le compilateur :

```
void const_reference(const int& b)
{
  b=5;
}
```

Exercice 7 - Les vecteurs

Dans cet exercice nous allons voir les vecteurs qui permettent de contenir différentes variables de même type. Attention en C++ les vecteurs démarrent à l'indice 0 ! Pour les définitions :

```
vector<type> mon_tableau(n); //définitions d'un tableau de taille n
vector<type> mon_tableau(n, xxx); //et dont tous les éléments valent xxx
vector<type> mon_tableau; //et d'un tableau dont la taille est inconnue
```

1. Exécuter ce premier exemple (et le modifier pour tester de nouvelles choses) :

```
#include <vector>
#include <string>
#include <iostream>
#include <algorithm> // Pour la fonction sort
using namespace std;
int main()
{
  vector<int> x(10); // Un tableau de 10 entiers
  for (int i=0;i<10;i++)
  {
        x[i]=i*(i+1);
  }
  int vec_size = x.size() ; // Retourne la taille du tableau x
  cout << "La taille du vecteur x est : " << vec_size << endl;
  x.resize(20); // Le tableau contient 20 éléments
  cout << "La nouvelle taille du vecteur x est : " << x.size() << endl;

  for (int i=0;i<x.size();i++)
  {
        cout << "x[" << i << "]=" << x[i] << endl;
  }
  return 0;
}
```

Attention l'accès aux valeurs d'un tableau par x[i] ne vérifie pas si on déborde ou non du tableau. Pour un accès moins rapide mais qui vérifie les bornes du tableau, il est possible d'utiliser x.at(i).

2. Jouer avec ce deuxième exemple :

```
#include <vector>
#include <string>
#include <iostream>
#include <algorithm> // Pour la fonction sort
using namespace std;
int main()
{
  vector<string> S; // un tableau vide de string
  S.push_back("Pierre"); //Rajoute "Pierre" en fin de tableau.
  S.push_back("Julien"); //Rajoute "Julien" en fin de tableau.
  S.push_back("Paul"); //Rajoute "Paul" en fin de tableau.
  sort(S.begin(),S.end());

  // Les string sont triés par ordre alphabétique
  for (int i=0;i<S.size();++i)
        cout << "Bonjour " << S[i] << endl;
  return 0;
}
```

Exercice 8 - Et si on veut faire de l'aléatoire ?

Il est possible de générer aléatoirement des *doubles* par exemple en utilisant la librairie *random*. Les 2 objets suivants doivent être définis pour une distribution normale :

```
std::default_random_engine generator(time(0));
std::normal_distribution<double> distribution(0.0,1.0);
```

au début de la fonction *main* par exemple. Et ensuite pour générer un nombre aléatoire suivant la loi normale $\mathcal{N}(\mu, \sigma^2)$, la commande est :

```
double r = mu + sigma*distribution(generator);
```

1. Générer 100 réels selon la loi $\mathcal{N}(1.5, 0.15^2)$.
2. Vérifier que la moyenne de vos 100 réels est proche de 1.5. Et l'écart type ?

Exercice 9 - Écrire dans un fichier et le lire

1. Pour écrire dans un fichier, nous allons utiliser la librairie "fstream" (file stream). Tester le code suivant (penser à regarder le fichier créé !).

```
#include <vector>
#include <iostream>
#include <fstream> // Pour pouvoir écrire et lire des fichiers
#include <string>
using namespace std;
int main()
{
  int vec_size = 10;
  vector<double> const_vec(vec_size,1.1);
  vector<double> it_vec(vec_size);
  for (int i=0;i<vec_size;i++)
  {
       it_vec[i]=1.5*(i+1);
  }
  vector<int> int_vec(vec_size);
  for (int i=0;i<vec_size;i++)
  {
       int_vec[i]=i;
  }
  ofstream mon_flux; // Contruit un objet "ofstream"
  string name_file("mon_fichier.txt"); // Le nom de mon fichier
  mon_flux.open(name_file, ios::out); // Ouvre un fichier appelé name_file
  if(mon_flux) // Vérifie que le fichier est bien ouvert
  {
    for (int i = 0 ; i < vec_size ; i++) // Remplit le fichier
    {
      mon_flux << int_vec[i] << " " << const_vec[i] << " " << it_vec[i] << " " << endl;
    }
  }
  else // Renvoie un message d'erreur si ce n'est pas le cas
  {
    cout << "ERREUR: Impossible d'ouvrir le fichier." << endl;
  }
  mon_flux.close(); // Ferme le fichier
  return 0;
}
```

2. À présent nous allons apprendre à lire un fichier. Ce qu'il faut comprendre c'est qu'on lit le fichier en le parcourant au fur et à mesure donc à chaque fois qu'on lit soit une ligne, soit un mot, soit un caractère, le curseur de lecture se retrouve derrière le dernier élément lu. Tester (et comprendre) le code suivant pour la lecture du fichier que nous venons de créer :

```cpp
#include <iostream>
#include <fstream> // Inclusion de "fstream"
using namespace std;

int main()
{
  //Ouvre un fichier
  ifstream mon_flux("mon_fichier.txt");

  // Premiere facon de lire un fichier (ligne par ligne)
  string ligne;
  getline(mon_flux, ligne);
  // On lit la première ligne : 0 1.1 1.5
  cout << "ligne 1 : " << ligne << endl;
  getline(mon_flux, ligne);
  // On lit la seconde : 1 1.1 3
  cout << "ligne 2 : " << ligne << endl; // etc ...

  // Deuxieme facon de lire : mot par mot (comme cin)
  double nombre1, nombre2;
  mon_flux >> nombre1 >> nombre2;
  // On lit les 2 premiers mots sur la ligne 3 : 2 et 1.1
  cout << "1er mot de la ligne 3 : " << nombre1 << endl;
  cout << "2eme mot de la ligne 3 : " << nombre2 << endl;

  // Troisieme facon de lire : caractere par caractere
  char a, b;
  mon_flux.get(a);
  mon_flux.get(b);
  // On lit le premier caractère sur la ligne 3 après 1.1 : un espace
  // (donc rien ne s'affiche!)
  cout << "premier caractère après premier mot de la ligne 3 : " << a << endl;
  // On lit le caractère après l'espace : 4
  cout << "caractère suivant : " << b << endl;
  mon_flux.close();
}
```

Exercice 10 - Et si certains types n'en étaient pas vraiment ?

1. Un ordinateur ne sait pas gérer du texte. Le type *char* stocke un nombre interprété comme une lettre grâce à la table ASCII (ex : le nombre 48 correspond au caractère *0*, 65 à *A*). Le type *char* ne correspond qu'à 1 seul caractère donc les textes sont des tableaux de *char*. Sans utiliser *string*, nous devrions donc utiliser ce type de code pour construire et afficher des mots :

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
  vector<char> mot(4); //Définit un mot comme un tableau de char
  mot[0]='C'; mot[1]='i'; mot[2]='a'; mot[3]='o'; //Mon mot lettre par lettre
  cout << mot[0] << mot[1] << mot[2] << mot[3] << endl; //Affichage du mot
  mot.resize(5); //Changement de mot = changement de taille
  mot[0]='S'; mot[1]='a'; mot[2]='l'; mot[3]='u'; mot[4]='t'; //Nouveau mot
  cout << mot[0] << mot[1] << mot[2] << mot[3] << mot[4] << endl; //Affichage
  return 0;
}
```

2. C'est là qu'intervient la **programmation orientée objet** (on en reparle au prochain TP !) : on place le tout dans une boîte facile à utiliser. Le même code avec l'objet *string* devient :

```
#include <iostream>
#include <string> // Obligatoire pour utiliser les objets string
using namespace std;
int main()
{
  string mot("Ciao"); //Création d'un objet 'mot' de type string
  cout << mot << endl; //Affichage
  mot = "Salut"; //Changement du contenu
  cout << mot << endl; //Affichage nouveau mot
  return 0;
}
```

Et nous pouvons aller encore plus loin :

```
string mot1("Ciao"), mot2("Salut"), mot3, mot4;
mot3 = "Vous préférez dire " + mot1 + " ou " + mot2 + " ?";
cout << mot3 << endl; //Concaténation

if (mot1 == mot2) //Comparaison de 2 mots
    cout << "Les deux mots sont identiques." << endl;
else
    cout << "Les deux mots sont differents." << endl;

mot4 = mot2.substr(2); //Extraction
cout << mot2 << " sans les 2 premières lettres devient " << mot4 << endl;
```

Réfléchir à comment sont implémentées (à partir de *char*) les méthodes : "+", "=" et "substr".

Exercice 11 - Résolution d'une équation différentielle

On considère l'équation différentielle suivante :

$$\left\{ \begin{array}{rcl} y'(t) & = & f(t, y(t)) \\ y(0) & = & y_0 \end{array} \right.$$

1. Écrire une fonction qui permet de calculer y^{n+1} à partir de y^n avec le schéma d'Euler explicite :

$$y^{n+1} = y^n + \Delta t f(t^n, y^n).$$

Le prototype de cette fonction sera le suivant :

```
// y : y^n entrée, y^(n+1) en sortie --> pour le modifier passer y par référence
// tn : t^n
// dt : pas de temps
// f : fonction f --> on passe la fonction par référence
void Euler(double& y, double tn, double dt, double (&f)(double, double));
```

2. Dans la fonction main, écrire la boucle principale permettant de calculer $y^1, y^2, \cdots, y^N$ à partir de y_0 en appelant la fonction "Euler" à chaque pas de temps.

3. Tester le programme pour les deux fonctions suivantes :

$$\left\{ \begin{array}{rcl} y'(t) & = & y \\ y(0) & = & -2 \end{array} \right.$$

et

$$\left\{ \begin{array}{rcl} y'(t) & = & y^2\, t \\ y(0) & = & -2 \end{array} \right.$$

Comparer avec les solutions exactes qui sont respectivement :

$$y(t) = y_0\, e^t \text{ et } y(t) = 2\frac{y_0}{2 - t^2 y_0}.$$

4. Vérifier l'ordre de convergence de la méthode d'Euler :
a. Calculer l'erreur entre la solution approchée et la solution exacte au temps final pour $\Delta t = 0.001$ et pour $T_{final} = 1$. On la note $e_{\Delta t}$.
b. Faire de même avec $\Delta t/2$. On la note $e_{\Delta t/2}$.
c. Calculer l'ordre de la méthode par :

$$\text{ordreConvEuler} = \log 2 \left(e_{\Delta t} \,/\, e_{\Delta t/2} \right).$$

4. Construire un fichier formé de 2 colonnes correspondant respectivement à t_n et $z(t_n)$. Tracer ensuite la courbe en utilisant gnuplot :

```
plot "ma_solution.txt" title "solution approchee" with linespoints
```

Ajouter une troisième colonne contenant la solution exacte. Tracer les 2 courbes :

```
plot "ma_solution.txt" using 1:2 title "solution approchee" with linespoints
replot "ma_solution.txt" using 1:3 title "solution exacte" with linespoints
```

Exercice 12 - Structurer son code

1. Vous pouvez partir de votre code obtenu à l'exercice précédent ou du code ci-dessous.

```
#include <cmath>
#include <iostream>
using namespace std;

void euler(double& y, double tn, double dt, double (&f)(double, double))
{
  y += dt*f(tn, y);
}

double f_ty2(double t, double y)
{
  return t*y*y;
}

int main()
{
  cout.precision(15);
  double dt(0.001), Tfinal(1.), y0(-2.), y(y0), tn;
  int N = int(round(Tfinal/dt));
  for (int i = 1; i < N+1; i++)
  {
    tn = i*dt;
    euler(y, tn, dt, f_ty2);
  }

  double y_ext_T_final = 2.*y0/(2. - y0*Tfinal*Tfinal);
  cout << "y'=t*y^2 avec y(0)=" << y0 << endl;
  cout << "y(Tf)=" << y << " and y_exacte(Tf)=" << y_ext_T_final << endl;

  return 0;
}
```

Déplacer les fonctions *euler* et *f_ty2* à la fin du programme. Vous obtenez ce type d'erreur (le message exact varie selon le compilateur) :

```
main.cc:21:22: error: use of undeclared identifier 'f_ty2'
```

2. En effet, afin d'améliorer la correction des erreurs, le C++ impose que toute fonction soit déclarée avant son premier appel. Cependant, la définition peut apparaître plus tard. Laisser les **définitions** des fonctions en bas du fichier mais rajouter au début, à la ligne 4 les **déclarations** :

```
void euler(double& y, double tn, double dt, double (&f)(double, double));
double f_ty2(double t, double z);
```

Cette fois ci la compilation fonctionne car la fonction *main* n'a pas besoin de savoir comment la fonction *euler* fonctionne (son algorithme : sa **définition**) pour l'utiliser (y faire appel) ; elle doit par contre savoir ce qui y rentre et ce qui en sort (en connaître son prototype : sa **déclaration**).

3. Nous allons maintenant pouvoir structurer notre code à la manière classique du C++, c'est-à-dire que nous allons séparer les **déclarations/prototypes** et les **définitions** dans deux fichiers différents. Actuellement, tout notre code est dans le fichier main.cc et nous allons donc créer :

- un header (fichier *.h) qui contiendra les prototypes des fonctions ;
- un fichier source (fichier *.cpp) qui contiendra la définition des fonctions.

Créer deux fichiers nommés respectivement : Fonction.h et Fonction.cpp :
a) Insérer dans le fichier Fonction.h les deux prototypes des fonctions *euler* et *f_ ty2* **et rien d'autre ! ! !**
b) Insérer dans le fichier Fonction.cpp les deux définitions des fonctions *euler* et *f_ ty2*. Rajouter en haut de votre fichier l'inclusion du fichier ".h" :

```
#include "Fonction.h"
```

c) Enfin dans le main.cc vous pouvez supprimer les prototypes et les définitions des fonctions, à l'exception bien sur de la fonction *main*. Afin d'avoir accès aux fonctions *euler* et *f_ ty2* vous devez en haut de votre fichier ajouter (comme pour le fichier .cpp) l'inclusion suivante :

```
#include "Fonction.h"
```

Vous êtes prêts pour la compilation en ajoutant le fichier Fonction.cpp :

```
g++ -std=c++11 -o run main.cc Fonction.cpp
```

4. Regarder votre code c'est important de bien comprendre comment les appels aux différents fichiers sont faits. Ne pas hésiter à commenter certaines lignes pour voir les erreurs qui apparaissent. Remarquer que le fichier qui est inclu est le *.h.

Une bonne structure de code permet d'éviter beaucoup beaucoup d'erreurs !
N'hésitez pas à revenir régulièrement à cet exercice !

Exercice 13 - Les pointeurs

1. L'utilisation des pointeurs est une notion complexe donc accrochez vous ! Les pointeurs sont utilisés dans tous les programmes et ils permettent de gérer très finement ce qui se passe dans la mémoire de l'ordinateur. La mémoire d'un ordinateur est constituée de *cases* (plusieurs milliards sur un ordinateur récent). Il faut donc un système pour que l'ordinateur puisse retrouver les cases dont il a besoin. Chacune d'entre elles possède un numéro unique que l'on appelle son adresse. Donc la variable *mon_ entier* définie par la commande suivante :

```
1 int mon_entier = 23;
```

a pour adresse *133655* d'après la figure suivante.

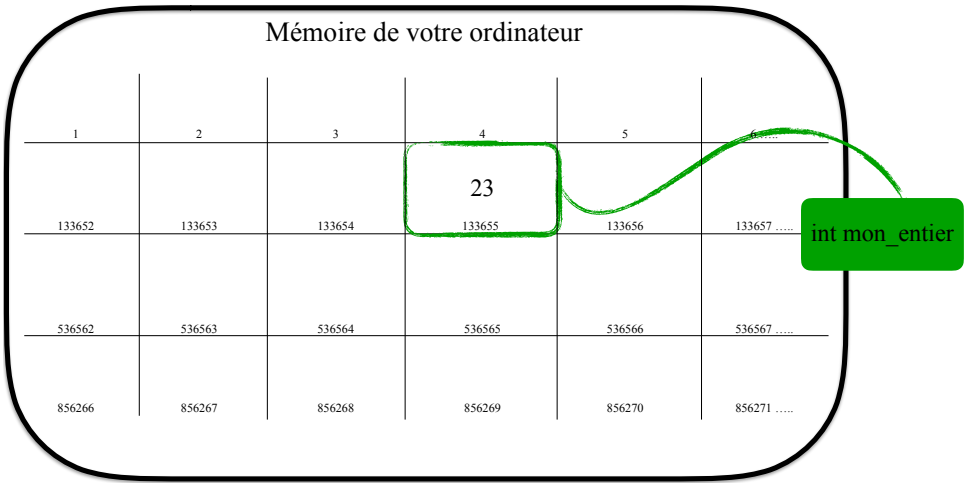

FIGURE 3 – Schéma de la mémoire de votre ordinateur

L'adresse est donc un deuxième moyen d'accéder à une variable. Tester le code suivant.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6     int mon_entier(23);
7     //Affichage de l'adresse de la variable en utilisant &
8     //Elle sera donnée en base 16 (d'où la présence de lettres)
9     cout << "L'adresse est : " << &mon_entier << endl;
10    return 0;
11 }
```

Nous allons maintenant pouvoir définir ce qu'est un pointeur :

Un pointeur est une variable qui contient une adresse mémoire (généralement celle d'une autre variable).

Pour comprendre comment cela fonctionne rajouter les quelques lignes suivantes :

```
1 //Ce code déclare un pointeur qui peut contenir
2 //l'adresse d'une variable de type int.
3 int *pointeurInt;
4 //On peut faire de même pour n'importe quel type :
5 double const *pointeurDoubleConst;
6 vector<int> *pointeurVector; //etc ...
```

Pour le moment, ces pointeurs ne contiennent aucune adresse connue. C'est une situation très dangereuse. Si vous essayez d'utiliser le pointeur, vous ne savez pas quelle case de la mémoire vous manipulez. Pire : elle contient une adresse quelconque (pas nécessairement "0" !), ce qui est une source très fréquente de bug et de faille de sécurité. Il ne faut donc jamais déclarer un pointeur sans lui donner d'adresse. Pour être tranquille, il faut toujours déclarer un pointeur en lui donnant la valeur "nullptr" (correspondant à l'adresse 0). En effet, sur la figure ci-dessus, la première case de la mémoire avait l'adresse 1 car l'adresse 0 n'existe pas. Lorsque vous créez un pointeur contenant l'adresse 0, cela signifie par convention qu'il ne contient l'adresse d'aucune case. Corriger le code précédent :

```
//Ce code déclare un pointeur qui peut contenir
//l'adresse d'une variable de type int.
int *pointeurInt = nullptr;
//On peut faire de même pour n'importe quel type :
double const *pointeurDoubleConst = nullptr;
vector<int> *pointeurVector = nullptr; //etc ...
```

Maintenant qu'on a la variable, il est possible de lui affecter une valeur :

```
1 int mon_entier(23); //Une variable de type int
2 int *ptr = nullptr; //Un pointeur pouvant contenir l'adresse d'un int
3 ptr = &mon_entier; //L'adresse de 'mon_entier' est mise dans le pointeur 'ptr'
4 //On dit alors que le pointeur ptr pointe sur mon_entier
5 cout << "L'adresse de 'mon_entier' est : " << &mon_entier << endl;
6 cout << "La valeur de 'ptr' est : " << ptr << endl;
7 cout << "La valeur est : " << *ptr << endl;
```

Décrire les étapes qui sont effectuées par la machine à la ligne 7.

Avant de voir à quoi cela peut bien servir (non le but n'est pas juste de se compliquer la vie), nous allons faire un petit récapitulatif.

Pour une variable **int nombre** :

- nombre permet d'accéder à la valeur de la variable,
- &nombre permet d'accéder à l'adresse de la variable.

Pour un pointeur **int *pointeur** :

- pointeur permet d'accéder à la valeur du pointeur, c-à-d à l'adresse de la variable pointée,
- *pointeur permet d'accéder à la valeur de la variable pointée.

Donc on a : "nombre égale *pointeur" et "&nombre égale pointeur"... Dans les questions suivantes, nous allons répondre à la question suivante : À quoi cela peut bien servir ?

2. **Une première utilisation : l'allocation dynamique.** Lors de la déclaration d'une variable, le programme effectue deux étapes :

- Il demande à l'ordinateur de lui allouer une zone de la mémoire.
- Il initialise la case avec la valeur fournie. Attention : si rien n'est fourni, il n'y a pas nécessairement de valeur par défaut.

Tout cela est entièrement automatique. De même, lorsque l'on arrive à la fin d'une fonction, le programme rend la mémoire utilisée à l'ordinateur. C'est ce qu'on appelle la libération de la mémoire. C'est à nouveau automatique. Les pointeurs permettent de faire tout ça manuellement.

```
int *pointeur = nullptr;;//Définit un pointeur pouvant contenir l'adresse d'un int
pointeur = new int;//Demande 1 case et renvoie 1 pointeur pointant vers celle-ci
*pointeur = 2;//Accède à la case mémoire pour en modifier la valeur
delete pointeur;//Libère la case mémoire
//Attention le pointeur pointe toujours mais vers une case vide !!!
pointeur = 0;//Indique que le pointeur ne pointe plus vers rien
```

Faire un programme demandant l'âge de l'utilisateur et l'affichant à l'aide d'un pointeur. C'est plus compliqué que la version sans allocation dynamique mais la mémoire est contrôlée. Vous devez définir

qu'une seule variable qui est votre pointeur !
Rappel : Une case peut être réservée dans la mémoire manuellement avec *new*. Dans ce cas, il ne faut pas oublier de libérer l'espace en mémoire, avec *delete*.

3. **Une deuxième utilisation : choisir parmi plusieurs éléments**
a. Tester le QCM suivant (Attention : lors du copier/coller les apostrophes autour de A et de B passeront mal, il faut donc les corriger).

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main()
6  {
7    string repA("Euler"), repB("Runge-Kutta");
8    cout << "Quel schéma en temps souhaitez-vous utiliser ? " << endl;
9    cout << "A) " << repA << endl;
10   cout << "B) " << repB << endl;
11
12   char votre_reponse;
13   cout << "Votre reponse (A ou B) : ";
14   cin >> votre_reponse; //Récupère la réponse de l'utilisateur
15
16   string *reponseUtilisateur = nullptr; //Un pointeur qui pointera sur la réponse
17
18   switch(votre_reponse)
19   {
20   case 'A':
21       reponseUtilisateur = &repA; //Déplace le pointeur sur la réponse choisie
22       break;
23   case 'B':
24       reponseUtilisateur = &repB;
25       break;
26   default:
27       cout << "Ce choix n'est pas valable !" << endl;
28       exit(0); //Le programme s'arrête
29   }
30   //Utilise le pointeur pour afficher la réponse choisie
31   cout << "Vous avez choisi la reponse : " << *reponseUtilisateur << endl;
32   //Le pointeur n'ayant pas été défini par un "new"
33   //il n'est pas nécessaire de libérer la case mémoire
34   return 0;
35 }
```

b. Que se passe t-il si on retire la ligne 28 et que l'on ne répond ni par A ni par B ?
c. L'exemple précédent est intéressant mais n'est pas vraiment utile. En effet, il est possible de juste copier un string sans le passer par un pointeur ! Pour un exemple plus interessant, nous allons reprendre le code de l'exercice 1.
i. Ajouter la fonction suivante dans le code (prototype dans le Fonction.h et définition dans le Fonction.cpp) :

```
1  double f_y(double t, double y)
2  {
3   return y;
4  }
```

ii. Modifier votre fonction main pour pouvoir demander à l'utilisateur de choisir entre résoudre $y' = y$ ou $y' = y^2 t$. Pour cela, construire le pointeur suivant :

```
1 double (*fct)(double, double);
```

et le définir dans un *switch*.

4. **Une troisième utilisation : partager une variable dans plusieurs parties du code.** Cela sera très utile quand nous commencerons la programmation orientée objet (prochain TP).