

Support de cours C++ - M1

André Garenne - 2022

1 Les bases du C++

- 1.1 Introduction
- 1.2 Logiciels nécessaires
 - 1.2.1 Choix d'un compilateur (**compiler**)
 - 1.2.2 Choix d'un éditeur de code ou d'un IDE
 - 1.2.3 Conventions de présentation dans ce document
- 1.3 Présentation du langage
 - 1.3.1 Avantages et inconvénients du C++
 - 1.3.2 C++ est un langage compilé
 - 1.3.3 Commenter du code source C++
 - 1.3.4 Hello World!
 - 1.3.4.1 Directives du préprocesseur
 - 1.3.4.2 Corps de l'application
 - 1.3.4.3 Générer l'exécutable du **Hello World!**
 - 1.3.4.4 Les arguments de la fonction **main**
 - 1.3.4.5 Manipulation et formatage des types
 - 1.3.5 Options du compilateur
- 1.4 Les variables et types de base en C++
 - 1.4.1 Le type **void**
 - 1.4.2 Le type **bool**
 - 1.4.3 Le type **char**
 - 1.4.4 Le type **int**
 - 1.4.5 Les types codant les réels
 - 1.4.6 Les "effets de bord"
- 1.5 Opérations sur les variables
- 1.6 Les Entrées-sorties
- 1.7 Les structures de contrôle
 - 1.7.1 Les blocs
 - 1.7.2 Les tests : **if**, **else** et **?**
 - 1.7.3 La structure **switch/case**
 - 1.7.4 La boucle **for**
 - 1.7.5 La boucle **while**
 - 1.7.6 La boucle **do**
 - 1.7.7 Comment modifier le déroulement d'une boucle ?
- 1.8 Les variables dimensionnées en C++
 - 1.8.1 L'héritage du C
 - 1.8.2 La classe vector
 - 1.8.3 La classe map
 - 1.8.4 La classe string
- 1.9 Les fonctions
- 1.10 Fonctions de calcul numérique
 - 1.10.1 Les fonctions mathématiques de base
 - 1.10.2 Les générateurs de nombres pseudo-aléatoires
- 1.11 Les exceptions
- 1.12 Gestion de la mémoire avec C++
 - 1.12.1 Des piles et des tas
 - 1.12.2 Des pointeurs et des références
 - 1.12.3 Les pointeurs en C (optionnel)
 - 1.12.4 Fonctions, pointeurs et références

- 1.13 Variables dimensionnées, pointeurs et références
- 1.14 Quelques conversions de type utiles : string, int et double
- 1.15 Les variables composites et les définitions de types
 - 1.15.1 Les struct, un premier pas vers les classes
 - 1.15.2 Les définitions de types ou typedef
- 1.16 Programmation objet en C++
 - 1.16.1 Généralités
 - 1.16.2 Les classes
 - 1.16.2.1 Comment créer une classe ?
 - 1.16.2.2 Comment instancier une classe ?
 - 1.16.2.3 Comment (et pourquoi) séparer prototype et implémentation des classes ?
- 1.17 Ecriture de fichiers texte
- 1.18 Lecture de fichiers texte
- 1.19 Directives du préprocesseur
 - 1.19.1 La directive #include
 - 1.19.2 La directive #define
 - 1.19.3 Macros et directives conditionnelles

2 Visualisation des données

- 2.1 Le logiciel R
 - 2.1.1 Installer et ouvrir une session R
 - 2.1.2 Modifier le répertoire de travail
 - 2.1.3 Charger et tracer les données d'un fichier dans une variable en R
 - 2.1.3.1 Courbes simples
 - 2.1.3.2 Courbes multiples
- 2.2 Le logiciel gnuplot
 - Installer et ouvrir une session gnuplot

3 Simulation du vivant in silico

- 3.1 Généralités
 - 3.1.1 Définitions
 - 3.1.2 Pourquoi réaliser des modèles computationnels ?
 - 3.1.3 A quelle échelle travailler ?
- 3.2 Formaliser un problème de biologie
 - 3.2.1 Les équations différentielles
 - 3.2.2 Les systèmes multi-agents
- 3.3 Méthodes de calcul numérique appliquées aux modèles dynamiques
 - 3.3.1 Principe général
 - 3.3.2 Résolution analytique
 - 3.3.3 Méthode d'Euler explicite (**Forward Euler Method**)
 - 3.3.4 Des données au modèle : validation et exploitation d'un modèle computationnel
 - Utilisation du modèle

1 Les bases du C++

1.1 Introduction

Le C++ est un langage développé par Bjarne Stroustrup à partir de 1983. Son objectif était alors d'étendre le langage C afin de lui apporter la possibilité de travailler avec des classes. C++ est donc un langage objet, compilé, fonctionnel et générique qui est disponible désormais [sur toutes les plateformes](#). Il a également fait l'objet de processus de normalisation ISO et existe en plusieurs versions dont la dernière est actuellement [C++20](#).

Malgré une réputation de complexité (pas complètement usurpée) C et C++ restent avec Java parmi les langages de programmation les plus utilisés comme le montrent les derniers chiffres de l'[index TIOBE](#). L'apprentissage du C n'est pas un préalable nécessaire à celui du C++. Malgré une filiation évidente et beaucoup de points communs, les deux langages sont autonomes et proposent deux approches complémentaires dans leurs domaines d'applications.

1.2 Logiciels nécessaires

Le présent support de cours nécessitera simplement un compilateur C++ et un éditeur de texte avec coloration syntaxique. C'est la ligne de commande qui sera privilégiée. Toutefois afin de produire des sorties graphiques toujours utiles pour visualiser nos résultats, nous sauvegarderons les données produites dans des fichiers et nous utiliserons soit [le logiciel R](#) soit [Python](#) et [Matplotlib](#) qui sont gratuits et multiplateformes pour les lire et les visualiser. Des exemples d'utilisation seront présentés plus loin. Si certains d'entre vous sont familiers de [Gnuplot](#) il peuvent également l'utiliser comme alternative.

Le choix de cette méthode pour produire des graphes a été rendu nécessaire en raison de la diversité d'OS utilisés par les étudiants pour travailler qui pose le problème de la compatibilité l'utilisation d'une API graphique commune. En effet outre ces problèmes techniques de compatibilité, la prise en main d'une telle librairie est en soi assez chronophage et le nombre d'heures de cours n'est pas suffisant.

1.2.1 Choix d'un compilateur (compiler)

Il existe des compilateurs officiels pour les plateformes Linux, Windows et Mac qui sont respectivement [gcc](#), [Microsoft Visual Studio](#) et [XCode](#). A côté d'eux il est possible de trouver des alternatives comme [Code::blocks](#), [Dev-C++](#) ou encore [Borland C++ Compiler](#) par exemple.

Remarque : a priori les programmes suivants sont sains mais je vous conseille quand-même d'avoir des anti-virus et anti-malware à jour.

- **Windows (compilateur ET IDE)**
 - [Code::blocks](#) (me semble finalement la meilleure option et peut inclure MinGW directement)
 - [Devcpp](#) (moins ergonomique je trouve)
 - [MinGW](#) et [Cygwin](#) sont des alternatives possibles mais moins pratiques pour ce cours
- **Mac (compilateur seul)**

Le plus simple est a priori d'installer d'abord XCode (gratuit sous Mac) puis ensuite de saisir dans un terminal la commande suivante :

```
xcode--select --install
```

Pour les plus aventureux sur Mac : https://brew.sh/index_fr et <https://formulae.brew.sh/formula/gcc>. J'ai souvent utilisé **homebrew** qui marche pour le portage de beaucoup de programmes mais je ne l'ai pas testé pour le compilateur **gcc**.

- **Linux (compilateur seul)**

L'enfance de l'Art ... avec [plusieurs variantes possibles](#).

```
sudo apt install g++
```

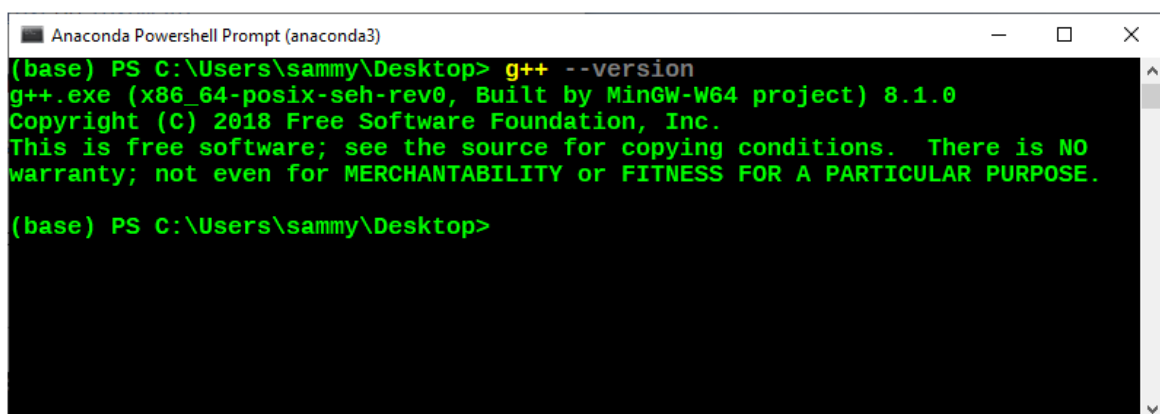
- **Synthèse wikipedia** : https://en.wikipedia.org/wiki/List_of_compilers#C++_compilers

1.2.2 Choix d'un éditeur de code ou d'un IDE

Là encore il n'y a aucune difficulté pour trouver de nombreux outils adaptés et gratuits. Beaucoup d'IDE (**I**ntegrated **D**evelopment **E**nvironment) sont multiplateformes et multi-langage comme [EMACS](#), [VIM](#), [NetBeans](#) ou encore [Eclipse](#). Un simple éditeur de texte avec coloration syntaxique peut également parfaitement convenir ici comme [Atom](#), [Notepad++](#), [SublimeText](#) ou [TextMate](#). La liberté de choix d'un IDE/compilateur est à la discrétion de chacun.

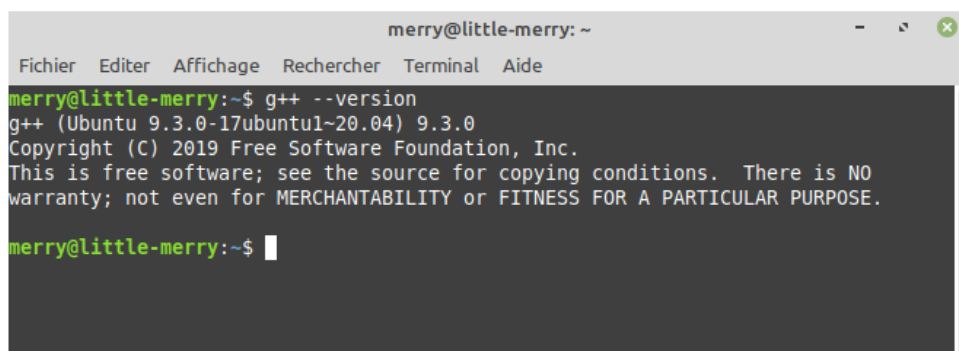
Remarque : pour ce cours les outils utilisés seront une version de C++ compatible C++17.

Sur la figure suivante la commande du compilateur **g++** est utilisée avec la commande **--version** pour avoir la version courante utilisée :



```
Anaconda Powershell Prompt (anaconda3)
(base) PS C:\Users\sammy\Desktop> g++ --version
g++.exe (x86_64-posix-seh-rev0, Built by MinGW-W64 project) 8.1.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

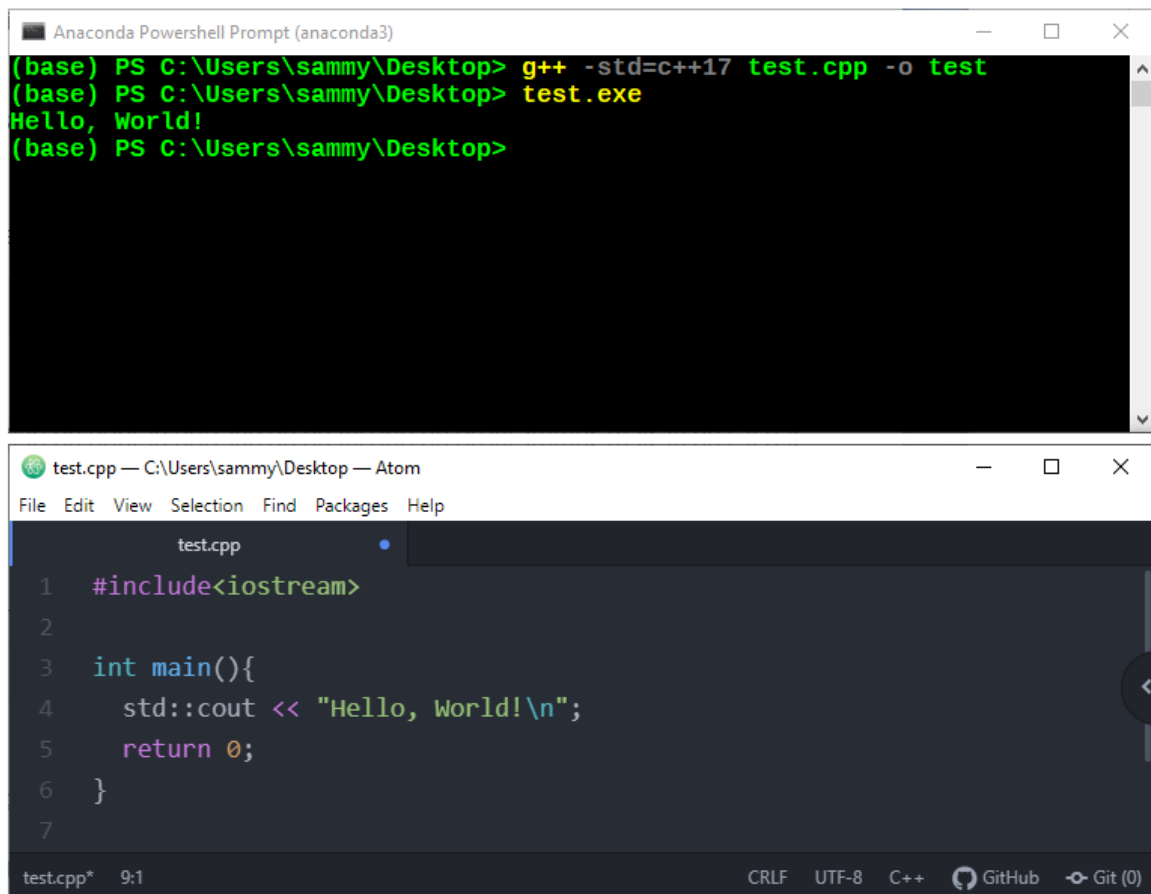
(base) PS C:\Users\sammy\Desktop>
```



```
merry@little-merry: ~
Fichier  Editer  Affichage  Rechercher  Terminal  Aide
merry@little-merry:~$ g++ --version
g++ (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

merry@little-merry:~$
```

La figure ci-dessous montre un exemple d'utilisation du compilateur MinGW (installé avec Code::blocks) sous Windows :



La fenêtre du haut contient la commande de compilation et d'exécution en mode C++17 puis l'exécution du code. Le code source est dans la fenêtre du bas et édité avec l'IDE Atom.

1.2.3 Conventions de présentation dans ce document

Le code C++ sera présenté de la façon suivante :

```
int main(){  
    // des trucs à faire ...  
    return 0;  
}
```

Et le traitement dans la console de commande Windows de la façon suivante :

```
g++ -std=c++17 test.cpp -o test  
test.exe # le .exe final est optionnel et sera omis par la suite
```

Et dans le cas d'une console **bash** sous Linux :

```
g++ -std=c++17 test.cpp -o test  
test
```

Par la suite les exemples seront présentés dans la console de commande Windows.

Remarque : le choix du nom **main.cpp** pour le programme est souvent utilisé par défaut mais c'est une simple convention. En revanche la fonction principale, point d'entrée de l'exécutable doit être nommée main.

1.3 Présentation du langage

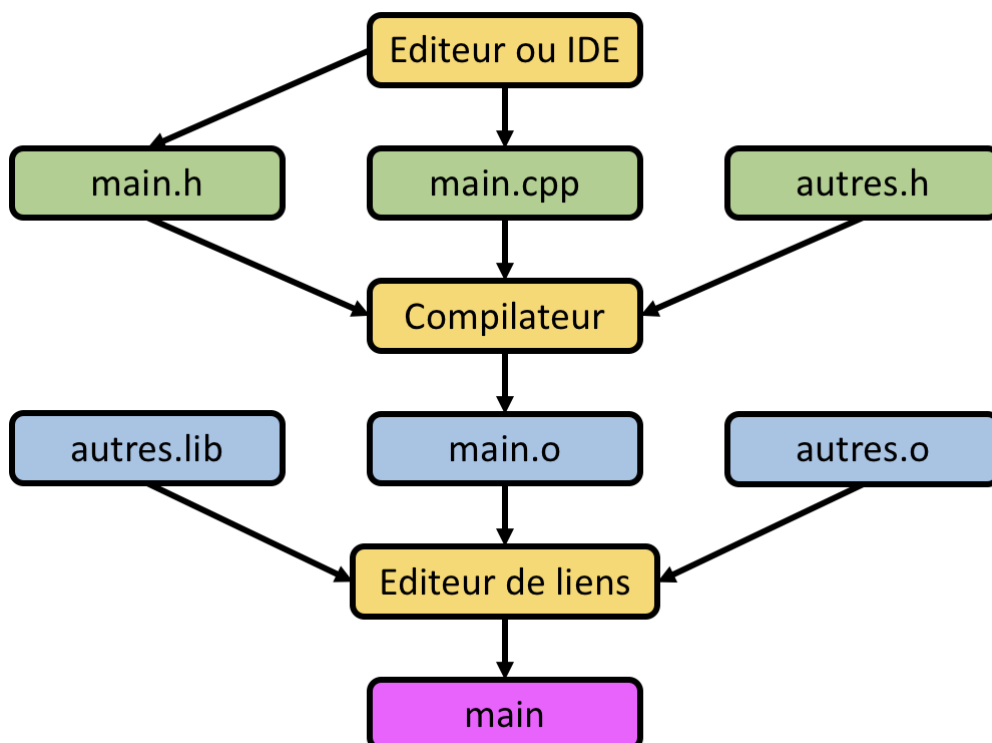
1.3.1 Avantages et inconvénients du C++

Le C++ offre un très grand nombre de fonctionnalités qui lui sont propres et qui incluent celles du C, une bonne portabilité des sources (voire excellente si l'on s'en tient à certains standards), un typage fort et une [grande vitesse d'exécution](#) pour le calcul intensif. Il permet de travailler en mode procédural, supporte la programmation orientée objet et reste [bare-metal](#). Enfin C et C++ demeurent les langages de loin les plus utilisés pour développer les OS et les **drivers**. En ce qui concerne la vitesse d'exécution (et d'une façon générale) C et C++ sont [extrêmement bien classés](#) ce qui n'en fait pas pour autant toujours une panacée. Lorsque l'on doit développer un projet il importe de prendre en compte de nombreux facteurs parmi lesquels la ré-utilisabilité, les ressources disponibles, l'existant (pas la peine de ré-inventer la roue).

Les inconvénients de C++ concernent surtout la clarté du code qu'il est très simple de rendre illisible ¹. Toutefois ce reproche est applicable à d'autres langages comme Java et n'est pas réellement une spécificité du C++. L'installation et l'utilisation de bibliothèques spécialisées peuvent aussi parfois s'avérer délicates. C'est sans doute l'un des points qui peut rebuter le plus d'utilisateurs. Par exemple générer des graphes de données scientifiques 2D/3D est nettement plus simple en utilisant Python ou R qu'en installant une bibliothèque spécialisée exploitée en C++. Par contre de l'animation 3D interactive fait en général l'objet de bibliothèques développées en C ou C++, de même qu'une grande partie du code qui est à la base de R et de Python. La gestion de la mémoire en C++ peut être vue selon les utilisateurs à la fois comme un avantage ou un inconvénient. En effet cette gestion est manuelle et ne fait pas appel à des processus automatisés d'allocation/libération de type **garbage collector** comme c'est le cas par exemple en Java. Il s'en suit qu'il faut être plus rigoureux sur ce point avec C ou C++ mais en retour le développeur a un meilleur contrôle sur la mémoire qui, quand elle est gérée proprement, peut efficacement se substituer aux traitements automatiques en y apportant plus de souplesse.

1.3.2 C++ est un langage compilé

Cela signifie que le code source est soumis à plusieurs étapes avant d'obtenir un code exécutable en langage machine (code natif) comme on le voit sur la figure suivante :



L'**éditeur** ou IDE permet à l'humain de produire le code source sous la forme de fichiers de type texte avec des fichiers de type **XXX.cpp** (corps du code source C++) et **XXX.h** (entête du code source avec un h pour **header**). Ces fichiers sont ensuite assemblés et compilés pour fournir un fichier objet **main.o**. Les fichiers objets contiennent déjà du code machine mais c'est un intermédiaire et il n'est pas exécutable en l'état.

Remarque : les sources des programmes en C++ contiennent en général au moins deux types de fichiers. Ceux qui contiennent les instructions à proprement parler dont le suffixe est `.cpp` et ceux qui contiennent les déclarations (l'interface) et dont le suffixe est `.h` (ou `.hpp`). Le **h** est pour **header** ou entête.

Le stade suivant fait donc appel à l'éditeur de liens (**linker**) qui assemble toutes les bibliothèques et fichiers objets nécessaires pour les réunir en un exécutable en code machine (adapté à l'architecture de la machine cible) ou en une librairie dynamique :

- dans le cas d'un lien statique avec une bibliothèque, son contenu est ajouté directement à l'exécutable
- dans le cas d'un lien dynamique ce contenu est lié à l'exécutable mais non contenu dedans

A l'issue de l'édition de liens le fichier contenant la fonction **main()**² est produit et exécutable directement. Ces différentes étapes sont illustrées dans le paragraphe suivant.

1.3.3. Commenter du code source C++

Les commentaires C++ peuvent être réalisés en fin de ligne à l'aide de la double barre oblique `//`. Ils peuvent aussi être réalisés avec la séquence `/*` **pour ouvrir la saisie du commentaire et se clore avec la séquence inverse** `*/`.

```
int a = 12; // commentaire en fin de ligne
/*
ceci est
un long long ...
...
...
... long commentaire
*/
```

1.3.4 Hello World!

Dans tous les cours sur un langage ... ou presque, il est d'usage de commencer par la création d'un exécutable minimaliste mais visuellement contrôlable : le **Hello World!**.

```
#include <iostream> // directive d'inclusion du pré-processeur
// déclaration de la fonction principale main
int main(){ // qui est le point d' "entrée" de l'exécutable
    std::cout << "Hello, world!\n"; // affichage sur la sortie standard
    return 0; // sortie du programme
}
```

Cet exemple simple va être détaillé dans les paragraphes suivants car il illustre très bien la structure de base et le fonctionnement d'un petit exécutable. Pour le moment nous allons le saisir avec notre éditeur (il n'est pas nécessaire de saisir les commentaires) et le sauvegarder dans un fichier nommé **hw.cpp** que nous placerons sur le bureau.

1.3.4.1 Directives du préprocesseur

Pour commencer, le fichier **iostream.h** de la bibliothèque standard (**Standard Template Library** ou STL) est importé à l'aide de la commande **include**. Cette opération est réalisée dans l'entête du programme et plus précisément dans la partie du code source que l'on nomme "directives du pré-processeur" (le préprocesseur gère le code en interprétant les fichiers source à l'aide des directives commençant par le symbole **#**). Initialement le fichier des flux d'entrée/sortie (**iostream**) devait être invoqué à l'aide de son fichier d'entête comme c'est généralement le cas (**iostream.h**). Mais avec la standardisation de 1998 **iostream** fait partie de l'espace de nommage (**namespace**) standard **std** et à ce titre nous utilisons les symboles **<** et **>** pour le délimiter (et non pas les **"**)³ et dans ce cas, le suffixe **.h** n'est plus nécessaire. Dans le paragraphe [1.19](#) nous reviendrons plus en détail sur les macro et les directives du pré-processeur.

1.3.4.2 Corps de l'application

La fonction **main()** est obligatoire pour la compilation d'un exécutable. C'est elle qui est en effet appelée quand le programme est exécuté. A la suite de la déclaration du nom de la fonction le bloc de code de cette fonction commence avec l'accolade ouvrante **{**. La ligne suivante appelle la fonction **cout** de la bibliothèque **std** et à l'aide de l'opérateur **<<** affiche le contenu suivant stocké dans la chaîne de caractères "Hello, world!\n". Le **\n** est un caractère d'échappement qui permet un retour à la ligne.

```
std::cout << "Hello, world!\n";
```

La commande **cout** peut donc être interprétée de la façon suivante : **std::cout** reçoit la chaîne de caractères "Hello, world!\n" via l'opérateur **<<** et redirige le flux vers la sortie standard (ici l'écran).

Remarque : comme nous le verrons par la suite la fonction de sortie standard **cout** s'adapte aux contenus à afficher.

La commande s'achève avec un **;** (en C++, en dehors des directives de compilation, les commandes s'achèvent toujours avec ce symbole). Le code s'achève avec un **return 0;** qui renvoie la valeur 0 en sortant de la fonction **main** (sortie du programme) et le bloc de code s'achève en dessous avec l'accolade fermante **}**.

1.3.4.3 Générer l'exécutable du Hello World!

Dans une console **bash** il faut se placer dans le dossier qui contient le fichier **.cpp** à compiler puis saisir la commande de compilation avec **g++**. Par exemple si nous avons placé le fichier **hw.cpp** sur le bureau :

```
g++ -std=c++17 test.cpp -o test
```

Comme nous le voyons ici une seule commande va pouvoir réaliser à la fois la compilation et l'édition de lien. C'est la commande **g++** suivie du nom du fichier à compiler (ici **hw.cpp**) suivi de l'argument **-o** pour préciser un nom d'exécutable puis le nom de l'exécutable lui-même (ici **hw**). Quelques options de compilation utiles seront précisées dans le paragraphe [1.3.5](#).

Remarque : en l'absence de spécification d'un nom pour le fichier de sortie, l'exécutable est stocké dans un fichier nommé par défaut **a.exe** (et **a.out** sous les OS type Linux).

Le programme peut alors être exécuté directement à l'aide de son nom :


```
test.exe
Hello, world!
```

1.3.4.4 Les arguments de la fonction main

De très nombreux programmes sont disponibles en ligne de commande sous la forme d'exécutables. Il s'agit en général de [programmes écrits en C \(ou C++\)](#). Par exemple le code de la commande **ls** contient près de 5000 lignes de code C. Pour qu'une fonction **main** puisse gérer des arguments passés après la commande il est d'usage de l'écrire de cette façon :

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

int main(int argc, char** argv){ // récupère plusieurs arguments passés dans
    argc, argv
    cout << argc << endl; // affiche le nombre d'arguments total (avec
    l'exécutable)
    for (unsigned int i=0;i<argc;i++){ // parcourt tous les éléments par indice
        cout << argv[i] << endl; // affiche chaque élément
    }
    return 0;
}
```

Le premier argument de la signature est **argc** qui est un entier qui contient le nombre de mots de la commande. Le second **char** argv** est un pointeur sur un pointeur de **char** ce qui revient finalement à une liste de mots qui peuvent être utilisés dans la commande comme des "options". Par exemple le programme ci-dessus renvoie :

```
g++ -std=c++17 test.cpp -o test # compilation de l'exécutable
test.exe hello you # exécution du programme avec 2 arguments : hello et you
3
test.exe
hello
you
```

Cette signature pour la fonction **main()** permet donc d'exécuter avec des arguments une fonction en mode ligne de commande. Il peut s'agir d'un nom de fichier par exemple, ou encore d'instruction pour contrôler le déroulement du programme.

1.3.4.5 Manipulation et formatage des types

A l'aide de la bibliothèque [iomanip](#) il est possible de contrôler les paramètres de format :

```
#include <iostream>
#include <iomanip> // ouvre à des fonctions supplémentaires
using namespace std;
int main(){
    cout << "Hello, world!\n";
    cout << "Hello, world!" << endl; // autre saut de ligne
    // caractères répétés
    cout << setfill ('*') << setw (5) << endl;
    cout << 1./6 << endl;
    // nombre de chiffres après la virgule fixés
    cout << setprecision(3);
    cout << 1./6 << endl;
```

```
    return 0;
}
```

A l'exécution nous obtenons :

```
Hello, world!
Hello, world!

0.166667
0.167
```

1.3.5 Options du compilateur

Il existe plusieurs centaines d'options dans gcc et il serait vain de les énumérer dans la mesure où elles sont par ailleurs bien documentées. La commande suivante permet de les afficher :

```
g++ --help
```

Parmi celles qui nous seront les plus utiles il faut mentionner :

1. **-g** qui génère des informations de débogage
2. **-Wall** qui génère tous les avertissements
3. **-o** qui permet de spécifier un nom de fichier de sortie
4. **-Ofast** meilleur niveau d'optimisation standard (vitesse d'exécution)
5. **-std=gnu++XX** spécifie d'utiliser le standard C++XX (XX année 11, 14, 17, 20)

Les options sont ensuite à choisir en fonction des applications et des contraintes de développement. Par exemple la compilation du code suivant :

```
#include <iostream>
int main(){
    std::cout << "Hello, world!\n";
    int a;
    return 0;
}
```

avec la commande suivante :

```
g++ -Wall -Ofast -std=gnu++14 -g test.cpp -o hw
```

aboutit à :

```
test.cpp:5:6: warning: unused variable 'a' [-Wunused-variable]
    int a;
        ^
1 warning generated.
```

Le fait d'avoir déclaré une variable **a** de type **int** sans jamais l'utiliser déclenche un **warning** durant la compilation mais pas d'erreur.

1.4 Les variables et types de base en C++

Les types qui seront présentés dans les paragraphes suivants sont standardisés selon le cas en C ou en C++ (**void**, **bool**, **int** etc.). Ils permettent notamment de catégoriser une variable. Une variable est un identifiant (nom) qui peut contenir lettres, chiffres (mais pas commencer par un chiffre) et **underscore** "_" mais pas d'espace, pas d'opérateurs ni de symbole de ponctuation. Cet identifiant est associé à un contenu et aussi implicitement à un type.

Il est possible d'avoir en C++ des informations sur le type d'une variable (il faut néanmoins savoir que le nom renvoyé est fonction du compilateur). Pour ce faire il faut inclure la bibliothèque **typeinfo** (**#include <typeinfo>**) et dans le code utiliser la fonction **typeid(var).name()**. Celle-ci renverra le type de la variable var sous la forme d'un pointeur de type **const char**.

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

int main(void){
    double doublevar=0.33;
    const char* res=typeid(doublevar).name();
    cout << "doublevar : " << doublevar << endl;
    cout << "type doublevar : " << typeid(doublevar).name() << endl;
    cout << "res : " << res << endl;
    cout << "type res : " << typeid(res).name() << endl;
    return 0;
}
```

Donne à l'exécution :

```
doublevar : 0.33
type doublevar : d
res : d
type res : PKc
```

Le typage en C ou en C++ n'est pas dynamique comme avec Python ou R par exemple et une variable doit être définie puis affectée, même si ces deux opérations peuvent être réalisées dans une même instruction et souvent la syntaxe suivante sera retrouvée :

type *variable* = **expression**

Une **expression** est tout ce qui est susceptible de renvoyer une valeur (un calcul, un test, un appel de fonction ...) et qui va constituer le contenu de la variable.

Le **type** a été évoqué plus haut et définit la nature de la variable

L'expression est tout d'abord évaluée puis grâce à l'opérateur d'affectation **=** sa valeur est attribuée à *variable*.

Le nom des variables est laissé à la discrétion du développeur mais il faut bien sûr éviter les mots-clés du langage (par exemple **for**), les caractères accentués, donner des noms de taille raisonnable tout en étant suffisamment explicite et essayer d'homogénéiser la méthode. Il vaut mieux également éviter de mélanger plusieurs langues dans le même code. Par ailleurs il existe des conventions de nommage auxquelles il est bon de se référer dès que l'on aborde un projet conséquent. Dans ce cours nous privilégierons l'utilisation du [lowerCamelCase](#).

1.4.1 Le type void

Il s'agit d'un type ... vide qui permet de spécifier l'absence de type pour un argument de fonction par exemple ou pour une valeur qu'elle retourne. Son principal intérêt réside dans la création de procédures (fonctions ne renvoyant rien) mais il ne sera pas très souvent utilisé dans la suite de ce cours.

Remarque : il ne faut pas confondre le type **void** (absence de type) avec **nullptr** qui indique un type : pointeur nul.

1.4.2 Le type bool

Les variables booléennes en C++ prennent la valeur **false** ou **true**. Les entiers peuvent également être interprétés comme des booléens (0 signifie **false** et les autres valeurs signifient **true** par défaut). Les booléens seront abordés plus en détail dans le paragraphe [1.8](#).

1.4.3 Le type char

C'est le type initial du codage ASCII (**American Standard Code for Information Interchange** ⁴) pour encoder un caractère et il est généralement codé sur un octet ou **byte** (8 bits). Le code ASCII standard est lui codé sur sept bits. Par exemple le caractère dont la valeur est 65 représente le 'A'. Dans l'exemple ci-dessous nous créons deux variables de type char puis nous leurs affectons la même valeur (65) de façon différente :

```
// code_01.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    char a,b; // déclaration de deux variables char
    a='A'; // affectation à l'aide du caractère entre simple quote '
    b=65; // affectation à l'aide de la valeur ASCII du caractère
    cout << "a: " << a << endl; // endl code un saut de ligne
    cout << "b: " << b << endl;
    return 0;
}
```

Après compilation et exécution nous obtenons :

```
a: A
b: A
```

Remarque : nous avons ici ajouté un espace de nommage par défaut qui permet de n'avoir à saisir que le nom de la fonction (**using namespace std;**). Par exemple désormais il suffira de taper **cout** et non plus **std::cout** comme c'était le cas dans le programme précédent **hw.cpp**.

Le type **char** est historiquement celui qui permet de gérer des chaînes de caractères en C. Nous verrons ultérieurement comment gérer ces chaînes de caractères en utilisant les classes C++ de la **stl**.

1.4.4 Le type int

Le type **int** est codé en fonction de l'architecture de la machine et donc au format du processeur utilisé. Sur une machine 16 bits il est codé sur deux octets et respectivement sur quatre et huit sur les machines 32 et 64 bits. La fonction **sizeof()** permet de récupérer la taille d'une variable.

```
// code_02.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    cout << "char est code sur " << sizeof(char) << " octet(s) " << endl;
    cout << "short int est code sur " << sizeof(short int) << " octet(s) " <<
endl;
    cout << "int est code sur " << sizeof(int) << " octet(s) " << endl;
    cout << "long int est code sur " << sizeof(long int) << " octet(s) " <<
endl;
    return 0;
}
```

Après compilation et exécution le programme affiche la taille des différentes variables :

```
char est code sur 1 octet(s)
short int est code sur 2 octet(s)
int est code sur 4 octet(s)
long int est code sur 8 octet(s)
```

1.4.5 Les types codant les réels

Les réels sont codés à l'aide des types **float** et **double** avec des précisions variables qui sont fonctions du nombre d'octets d'encodage (le séparateur décimal est ici le "." et pas la ",").

```
// code_03.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    cout << "float: est code sur " << sizeof(float) << " octet(s) " << endl;
    cout << "double: est code sur " << sizeof(double) << " octet(s) " << endl;
    cout << "long double: est code sur " << sizeof(long double) << " octet(s) "
<< endl;
    return 0;
}
```

Après compilation et exécution :

```
float est code sur 4 octet(s)
double est code sur 8 octet(s)
long double est code sur 16 octet(s)
```

Les gammes de valeurs encodées selon les types ⁵ sont indiquées dans le tableau :

Type	Nombre d'octets (byte)	Gamme de valeurs
char	1	-127 to 127 or 0 to 255
unsigned char	1	0 to 255
signed char	1	-127 to 127
int	4	-2147483648 to 2147483647
unsigned	4	0 to 4294967295
signed	4	-2147483648 to 2147483647
short	2	-32768 to 32767
unsigned short int	2	0 to 65,535
signed short int	2	-32768 to 32767
long int	4	-2,147,483,648 to 2,147,483,647
signed long int	4	same as long int
unsigned long int	4	0 to 4,294,967,295
float	4	+/- 3.4e +/- 38 (7 digits)
double	8	+/- 1.7e +/- 308 (15 digits)
long double	8	+/- 1.7e +/- 308 (15 digits)
wchar t	2 ou 4	1 wide character

Remarque : il est possible d'ajouter le modificateur **const** en déclarant une variable et d'en faire ainsi une constante : **const int maConstante = 12;** Cela signifie bien-sûr que le code ne devra pas tenter de la modifier car ceci génèrerait alors une erreur à la compilation. Le principal avantage de déclarer une constante, en dehors de la cohérence apportée au code, est de permettre au compilateur de produire un code plus optimisé.

1.4.6 Les "effets de bord"

Les "[effets de bord](#)" (mauvaise traduction de l'anglais *side-effect*) se produisent en général quand des instructions peuvent modifier l'état interne du processeur sans le déclarer explicitement. En pratique cela peut se produire par exemple :

- quand une fonction modifie le contenu d'une variable indépendamment de la valeur retournée
- quand un opérateur modifie la valeur d'un de ses opérandes
- quand des directives du préprocesseur sont mal gérées ...

Le code suivant illustre ces principes :

```
#include <iostream>
#define SOMME(a) (a+a)

int x;
void reset_x(){
    x=0;
```

```

}

using namespace std;
int main(void){
    // effet de bord 1
    int val=3;
    cout << SOMME(val++) << endl; //équivalent à cout << (val+++val++) << endl;
    cout << val << endl;

    // effet de bord 2
    x=12; // variable déclarée en externe à main
    cout << x << endl;
    reset_x();
    cout << x << endl;
    return 0;
}

```

A l'exécution cela donne :

```

7 // l'opérateur + additionne 3, incrémente puis 4 => 7 puis incrémente => 5
5
12 // la valeur de x
0 // est directement remise à zéro

```

1.5 Opérations sur les variables

La création des variables en C++ nécessite une étape de déclaration suivie d'une étape d'affectation. Les deux peuvent se faire directement dans la même commande comme cela a été évoqué plus haut et comme on le voit dans l'encadré du code suivant :

```

// code_04.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){ // création, calcul et affichages de variables numériques
    char a1; // déclaration
    a1 = 12; // affectation
    a1='A'; //nouvelle affectation
    cout << "a1: " << a1 << " et (int) a1: " << (int) a1 << endl;
    int a2 = 15; // déclaration et affectation
    double a3 = 0.1; // déclaration et affectation
    cout << "a3*a1: " << a3*a1 << endl;
    int a4=++a2; // préincrément de a2 et affectation à a4 de a2+1
    a3*=100; // équivalent à a3=a3*100
    cout << "a3: " << a3 << endl;
    cout << "a4: " << a4 << endl;
    return 0;
}

```

Après compilation et exécution :

```

a1: A et (int) a1: 65
a3*a1: 6.5
a3: 10
a4: 16

```

Les opérations sur les variables incluent notamment les opérateurs `+`, `-`, `*`, `/`, `%`, `&&`, `||`, et `!`. Le symbole `%` représente la fonction modulo (reste de la division euclidienne entière) et le `!` la négation logique. Il existe par ailleurs les opérateurs binaires de comparaison : `==`, `>=`, `<=`, `>`, `<` et `!=` qui permettent de comparer des variables ou des valeurs. Ces opérateurs renvoient selon les cas la valeur vraie (1) ou fausse (0).

Enfin `&&` et `||` codent respectivement pour les **AND** :

AND (&&)	true	false
true	true	false
false	false	false

et les **OR** logiques :

OR ()	true	false
true	true	true
false	true	false

Le script suivant montre le résultat de l'utilisation d'opérateurs de comparaisons qui est renvoyé sous la forme de valeurs entières 1 (**true**) ou 0 (**false**) :

```
// code_05.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    cout << "2 est superieur à 1 : " << (2>1) << endl;
    cout << "1 est superieur à 2 : " << (1>2) << endl;
    cout << "10 est egal à 10 : " << (10==10) << endl;
    return 0;
}
```

Après compilation et exécution nous obtenons :

```
2 est superieur à 1 :1
1 est superieur à 2 :0
10 est egal à 10 : 1
```

Les résultats des comparaisons sont bien des booléens (**bool**) mais pour l'affichage ils sont convertis en entiers et donc apparaissent sous la forme 0 ou 1.

Remarque : il faut ajouter des parenthèses autour du test ici car si on les omet, l'ordre de priorité des opérateurs renvoie une erreur de compilation.

1.6 Les Entrées-sorties

Nous avons déjà abordé l'utilisation de la commande **cout** qui dirige un flux vers la sortie par défaut (en général l'écran). Il serait intéressant de pouvoir en retour utilise la console comme une interface de saisie pour un utilisateur du programme. La commande **cin** permet de réaliser cette opération comme nous le montre l'exemple suivant :


```

#include <iostream>
#include <typeinfo>
using namespace std;
int main(void){
    char nom[50];
    double note;
    cout << "Saisir un nom : ";
    cin >> nom;
    cout << "Saisir une note : ";
    cin >> note;
    cout << "Donc " << nom << " a eu la note de " << note << endl;
    return 0;
}

```

Après compilation et exécution le programme donne le résultat suivant :

```

Saisir un nom : Leatherface
Saisir une note : 19
Donc Leatherface a eu la note de 19

```

Le programme attend qu'une valeur soit saisie et passée dans une variable **nom** (tableau de char de taille 50) avec **cin** puis la même opération se répète pour la variable **note**. Le programme affiche ensuite de façon formatée le contenu des variables qu'il vient de saisir.

1.7 Les structures de contrôle

1.7.1 Les blocs

De très nombreuses structures en C++ font appel à la notion de bloc qui est un héritage de la programmation procédurale et qui désigne un groupe d'instructions. Par exemple la fonction **main** que nous avons utilisée est suivie d'un bloc. Celui-ci est délimité par les accolades **{** et **}**. Il n'est pas nécessaire de placer un ";" à la fin d'un bloc car il est considéré comme une instruction unique quel que soit le nombre de lignes qui le composent. Comme nous allons le voir ensuite, le bloc est la brique de base des structures de contrôle. Mais il y a un point important à souligner ici. Quand une variable est créée dans un bloc, par défaut elle n'existe et n'est adressable que dans ce bloc :

```

// code_06.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    { // bloc 1
        int a=12; // a n'existe QUE dans le bloc 1 et pas dans main
    }
    cout << a << endl;
    return 0;
}

```

A la compilation une erreur est détectée :

```
g++ code_06.cpp -o code_06
code_06.cpp:8:10: error: use of undeclared identifier 'a'
    cout << a << endl;
           ^
1 error generated.
```

En effet la variable **a** ayant été déclarée dans un bloc différent de l'appel à **cout** elle est inconnue à cet endroit. Pour que cela fonctionne il aurait fallu déclarer **int a**; avant le début du bloc, lui affecter une valeur dans le bloc et il devenait alors accessible au **cout**. L'exemple suivant illustre plus avant cette notion de portée des variables :

```
// code_07.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    int b; // b existe dans main
    { // bloc 1
        int a=12; // a n'existe que dans le bloc 1
        { // bloc 2
            b=a; a ET b existent dans le bloc 2
        }
    }
    cout << b << endl;
    return 0;
}
```

A la compilation il n'y a plus d'erreur et l'exécution donne :

```
12
```

Comme on le voit les variables créées dans un bloc sont accessibles aux blocs qui leurs sont imbriqués mais pas aux blocs d'ordre supérieur.

1.7.2 Les tests : if, else et ?

La forme générale d'un test est :

```
if (condition){ Si ...
    Opérations
} else { Sinon ... (remarque : else est une clause optionnelle)
    Autres opérations
}
```

La **condition** est en général un test de comparaison du type de ceux abordés dans [1.5](#). Si le résultat exprimé par la **condition** est vrai (donc différent de 0) alors l'instruction (ou le bloc) placée immédiatement après est exécutée. Dans le cas contraire elle n'est pas traitée et le programme continue à sa suite. L'instruction peut être une commande seule ou un bloc de commandes. Exemple :

```
// code_08.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    int a=1;int b=2;int c=3;
```

```

if (a>b){
    cout << "a>b" << endl;
} else { // instruction "sinon"
    cout << "a<b" << endl;
}
if (b>c || c>a){ // OR logique
    if (b>c) cout << "b>c" << endl; // tests imbriqués
    if (c>a) cout << "c>a" << endl;
}
if (a<c && b>a) cout << "a>c et b>a" << endl; // AND logique
return 0;
}

```

L'exemple ci-dessus nous permet d'illustrer quelques caractéristiques importantes des tests. (i) Les tests peuvent être imbriqués. (ii) Il est possible d'utiliser la condition **sinon** grâce à la commande **else**. (iii) Les conditions des tests peuvent être combinées à l'aide des opérateurs OR (| |) ou AND (&&). Le résultat à l'exécution est donné ci-dessous :

```

a<b
c>a
a>c et b>a

```

Il existe également un opérateur ternaire ? qui peut réaliser des tests avec une syntaxe compacte :

```
(condition) ? (valeur{si vrai}) : (valeur{si faux});
```

Si la condition est vraie la première valeur est renvoyée et la seconde dans le cas contraire. Par exemple pour récupérer la valeur maximum de **a** et **b** et l'affecter à **c** nous pouvons utiliser le code suivant :

```

// code_09.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    int a=10; int b=20; int c; // déclaration de trois variables int
    c=(a>b)?a:b; // est-ce que a>b ? si oui alors a sinon b est renvoyé dans c
    cout << c << endl;
    return 0;
}

```

```

code_09
20

```

1.7.3 La structure switch/case

Si les tests **if** permettent de gérer à peu près tous les cas de figure, il est parfois plus simple d'utiliser une forme d'exécution conditionnelle dans laquelle les conditions sont énumérées sous la forme de cas à traiter indépendamment (ou pas). La structure utilisée dans ce cas sera appelée **switch/case** et se présente sous la forme générale suivante :

```

switch (argument){
    case cas_1:{bloc d'operations_1}
    case cas_2:{bloc d'operations_2}
    ...
    case cas_n:{bloc d'operations_n}
    default :{bloc à exécuter par défaut}
}

```

L'argument à passer à **switch** doit être une valeur entière. Chaque cas est alors traité et lorsque la valeur de **argument** est égal à **cas_n** alors c'est le bloc d'opérations correspondant qui est traité. Toutefois quand ce cas se présente, tous les autres cas suivants sont traités également par défaut. Pour éviter ceci il faut employer le mot-clé **break** (qui permet de sortir de la structure immédiatement enveloppante) à la fin de chaque bloc d'opérations. Si aucun cas n'a été identifié il est enfin possible d'utiliser la condition **default** qui sera alors appliquée. Exemple :

```

// code_10.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    int a=4; // on fait varier a de 1 à 3
    cout << "cas ou a = " << a << endl;
    switch(a){
        case 1:{
            cout << "trouve le cas a=1" << endl;
            // ici le break est supprimé;
        }
        case 2:{
            cout << "trouve le cas a=2" << endl;
            break;
        }
        default:{
            cout << "rien trouve" << endl;
        }
    }
    return 0;
}

```

Pour tester ce programme nous allons changer la valeur de **a** dans le code, de un à trois inclus et après chaque modification de **a** nous sauvegardons, nous recompilerons et nous exécutons :

```

code_10
cas ou a = 1
trouve le cas a=1
trouve le cas a=2
...
g++ code_10.cpp -o code_10
code_10
cas ou a = 2
trouve le cas a=2
...
g++ code_10.cpp -o code_10
code_10
cas ou a = 3
rien trouve

```

- Comme nous le voyons ici, quand nous fixons **a=1** le cas correct est bien traité mais comme le **break** a été omis le programme exécute aussi le cas suivant ou **a=2** (même si **a** est bien égal à 1). A l'issue du cas **a=2** comme le **break** est présent le programme s'arrête.
- Quand nous fixons **a=2** l'exécution donne bien un traitement correct car le **break** vient empêcher de continuer.
- Quand nous fixons **a=3** aucun cas n'ayant été identifié le programme exécute le bloc par défaut

1.7.4 La boucle for

Lorsque l'on doit répéter une opération un nombre de fois connu à l'avance, le plus simple est de confier à une boucle **for** le soin de le réaliser. Elle s'utilise de la façon suivante :

```
for (initialisation, condition, incrémentation){
    bloc d'opérations
}
```

Une boucle **for** utilise une variable compteur, l'**initialise** puis l'**incrmente** et répète l'**opération** (ou le **bloc d'opérations**) tant que la **condition** renvoie la valeur vrai. Par exemple pour calculer la somme du carré des 10 premiers entiers nous pouvons utiliser le code suivant :

```
// code_11.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    int somme=0;
    for (int i=1;i<=10;i++){
        somme+=i*i;
    }
    cout << somme << endl;
    return 0;
}
```

La structure de boucle **for (int i=1;i<=10;i++)** crée localement une variable entière **i** compteur de boucle, l'initialise à 1, puis répète la boucle de 1 en 1 (**i++**) tant que **i ≤ 10**. Quand **i** atteint 11 le programme passe à la suite. Après compilation et exécution nous obtenons un résultat de 385 :

385

1.7.5 La boucle while

Quand le nombre d'itérations n'est pas connu à l'avance la boucle **while** est une alternative à la boucle **for**. Elle se présente de la façon suivante :

```
while (condition){
    bloc d'opérations
}
```

La boucle se répète tant que **condition** renvoie la valeur vrai. Supposons par exemple que l'on veuille connaître la valeur **n** pour la **somme** des **n** premiers entiers qui ne dépasse pas 1000. Le code suivant permet de répondre à cette question :

$$somme = \sum_{i=1}^n i$$

```
// code_12.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    int somme=0;
    int n=0;
    while (somme<=1000){ // tant que la somme est <= 1000
        somme+=++n; // pré-incrémentation de n puis incrémentation de somme
    }
    // l'étape somme > 1000 a été franchie donc il faut rectifier
    // la valeur de somme en enlevant la dernière valeur
    somme-=n--; // décrémentation de somme et post-décrémentation de n
    cout << "n = " << n << " pour une somme de " << somme << endl;
    return 0;
}
```

Après compilation et exécution nous obtenons un résultat de 990 pour **n=44** :

```
n = 44 pour une somme de 990
```

Nous verrons plus loin comment utiliser les commandes de rupture des boucles (**break** et **continue**) pour améliorer leur contrôle.

1.7.6 La boucle do

Le principe est très proche de la boucle **while** simple à ceci près que le bloc d'opérations est toujours réalisé au moins une fois car la condition de sortie se situe après :

```
do{
    bloc d'opérations
}while(condition)
```

Là encore tant que la **condition** est vraie la boucle continue. Pour illustrer cette structure de contrôle nous allons nous inspirer de l'exemple précédent à ceci près que désormais le programme doit donner la valeur **n** telle que le produit **prod** des **n** premiers entiers soit inférieur ou égal à 1000 :

$$prod = \prod_{i=1}^n i$$

```
// code_13.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    int prod=1;
    int n=0;
    do {
        prod*=++n; // pré-incrémentation de n puis produit de prod
    } while (prod<=1000);
    // l'étape somme > 1000 a été franchie donc il faut corriger
    prod/=n--; // division de produit et post-décrémentation de n
    cout << "n = " << n << " pour un produit de " << prod << endl;
    return 0;
}
```

L'exécution renvoie bien la valeur **n=6** avec un produit de 720.

```
n = 6 pour un produit de 720
```

1.7.7 Comment modifier le déroulement d'une boucle ?

Il est possible de modifier le déroulement d'une boucle de deux manières : soit en l'interrompant avec **break** soit en la poursuivant directement vers la prochaine accolade fermante (donc en retournant au début) avec **continue** :

```
// code_14.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    int compteur=0;
    while(true){ // boucle infinie
        compteur++;
        cout << "compteur: " << compteur << endl;
        if (compteur<3) continue; // retourne au début de la boucle
        compteur+=10;
        if (compteur>40) break; // sort de la boucle
    }
    return 0;
}
```

Comme on le voit dans le code ci-dessus une boucle infinie est mise en place. Un compteur est incrémenté en début de boucle et sa valeur affichée. Un premier test applique la commande **continue** tant que ce compteur est inférieur à trois ce qui signifie que la portion de code située à la suite de ce test n'est pas exécutée tant que le test est vrai. Au delà de deux (donc à partir de **compteur = 3**) le compteur est incrémenté en plus de 10 à chaque pas, donc de 11 en tout par tour de boucle, d'où l'affichage de la valeur suivante de 14. Enfin un test vérifie que le compteur est **>40** et sort de la boucle quand la condition est réalisée ce qui donne la sortie écran suivante :

```
compteur: 1
compteur: 2
compteur: 3
compteur: 14
compteur: 25
compteur: 36
```

1.8 Les variables dimensionnées en C++

L'objectif de ce cours est l'application du langage C++ à la simulation de systèmes biologiques à des fins de modélisation. Il va donc falloir souvent traiter de l'évolution temporelle de signaux et donc de séries de valeurs dans l'espace et dans le temps. C'est à ce stade que l'utilisation de variables dimensionnées prend tout son sens. Une variable dimensionnée est un identificateur unique qui permet de pointer sur une toute une série de valeurs à la fois. En R ce sont par exemple les vector, les matrix, les data.frame ou encore les list. En Python ce sont les list ou les dict etc.

1.8.1 L'héritage du C

Avant l'apparition du paradigme objet en C++, le C pouvait déjà travailler sur des variables dimensionnées. Comme l'objet de ce cours est d'utiliser C++ au mieux, nous nous contenterons de montrer rapidement ici comment C gère les tableaux et les chaînes de caractères à travers quelques exemples simples. Considérons le code suivant :

```
// code_19.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    int a[3]; // déclaration d'un tableau statique d'entiers
    a[0]=1;a[1]=2;a[2]=3; // puis affectations
    char b[]={'a','b','c',0}; // déclaration et affectation
    cout << "la variable a contient : " << a[0] << "," << a[1] << " et " << a[2]
<< endl;
    cout << "la variable b contient : " << b[0] << " puis " << b[1] << " et " <<
b[2] << endl;
    b[1]='B';
    cout << "la variable b contient maintenant : " << b[0] << " puis " << b[1]
<< " et " << b[2] << endl;
    cout << "donc b contient : " << b << endl;
    return 0;
}
```

Dans le code ci-dessus nous avons créé deux variables dimensionnées. La première (a) est créée en spécifiant sa taille (3) puis en réalisant les affectations par indice. La seconde (b) correspond à une chaîne de caractères en C et ne nécessite pas que l'on précise la taille entre [] car l'affectation est réalisée dans la même commande que la déclaration. Dans les deux cas, ces variables sont allouées statiquement, c'est à dire que, même si le contenu peut en être modifié, ce n'est pas le cas pour la longueur (ou dimension). Après compilation et exécution nous obtenons :

```
la variable a contient : 1,2 et 3
la variable b contient : a puis b et c
la variable b contient maintenant : a puis B et c
donc b contient : aBc
```

Remarque : un tableau est en général aussi un pointeur vers l'adresse mémoire de son premier élément. Dans le cas ci-dessus par exemple a[0] peut être remplacé par *a et a[1] par *(a+1).

Il est aussi possible de travailler avec des tableaux dynamiques en C en passant par les pointeurs et des allocations dynamiques mais nous n'irons pas plus loin car l'objectif, comme cela a été rappelé plus haut est d'utiliser les capacités de C++ pour la gestion des variables dimensionnées et des chaînes de caractères ce que nous aborderons dans les paragraphes suivants.

1.8.2 La classe vector

Pour gérer un tableau dynamique de types à une dimension sans passer explicitement par des pointeurs, le C++ et la **stl** ont apporté la classe **vector**. Nous reviendrons dans le paragraphe [1.16](#) sur les notions de classes et d'objets mais pour le moment nous verrons qu'il s'agit d'un type particulier appelé conteneur car il contient des ensembles d'autres types comme par exemple des

int, des **double** ou d'autres objets prédéfinis ou non. La création et l'utilisation d'un **vector** en C++ peut être illustrée comme suit :

```
// code_20.cpp
#include <iostream>
#include <vector>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    vector<int> v1(3); // création d'un vector de taille 3
    cout << "taille de v1 : " << v1.size() << endl;
    // ajout de 10 éléments à ce vecteur
    for (int i=0;i<10;i++) v1.push_back(i);
    cout << "taille de v1 : " << v1.size() << endl;
    // stocke les carrés des 13 premiers entiers dans v1
    for (int i=0;i<v1.size();i++) v1.at(i)=(i+1)*(i+1);
    // affiche le contenu de v1
    cout << "contenu de v1 : ";
    for (int i=0;i<v1.size();i++) cout << v1.at(i) << " ";
    cout << endl;
    v1.clear(); // efface le contenu de v1
    cout << "taille de v1 : " << v1.size() << endl;
    return 0;
}
```

Le code ci-dessus montre quelques-unes des facilités d'utilisation de la classe **vector**. Un vecteur d'entier v1 (donc un objet de la classe **vector**) va successivement être créé, allongé, rempli avec des valeurs spécifiques, affiché puis vidé. Pour ce faire nous utiliserons des méthodes spécifiques à l'objet v1.

Remarque : avec le standard C++11 il est également possible d'initialiser un **vector** à la construction en utilisant la commande **vector v1{1,2,3}**; Dans ce cas v1 contient d'emblée ces trois valeurs.

Les méthodes abordées ici sont les suivantes :

- l'opérateur [*indice*] qui n'est pas une méthode à proprement parler mais un opérateur équivalent à la méthode **at(indice)**.
- la méthode **size()** qui renvoie le nombre d'éléments contenus dans **v1**
- la méthode **clear()** qui efface le contenu de **v1**

Nous voyons au passage qu'une méthode s'applique à un objet à l'aide d'un "." par exemple **v1.clear()**. C'est un principe général et c'est aussi le cas pour les attributs des objets. Dans le cas où l'objet est un pointeur nous utilisons l'opérateur -> mais tout ceci sera revu en détail dans la partie [1.16](#)

```
taille de v1 : 3
taille de v1 : 13
contenu de v1 : 1 4 9 16 25 36 49 64 81 100 121 144 169
taille de v1 : 0
```

Ces opérations de base suffisent à l'essentiel de l'utilisation que nous aurons des vecteurs mais il existe bien des fonctionnalités supplémentaires que l'on peut utiliser avec cette classe. Pour approfondir ces informations il suffit d'aller [ici](#). Le site <http://www.cplusplus.com> est du reste un excellent site de référence à consulter au besoin.

1.8.3 La classe map

Si la classe **vector** permet de stocker des séries d'éléments et d'y accéder au moyen de leur position (ou indice ou index) il est parfois utile de pouvoir accéder aux éléments d'un conteneur à l'aide d'une clé arbitraire et pas seulement d'une valeur indicée automatiquement. Les **map** permettent d'utiliser cette fonctionnalité et sont assez proche des structures de type dictionnaire en Python.

```
// code_21.cpp
#include <iostream>
#include <map>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    // création d'une map m1 de clé: int et de valeur: int
    map<int, int> m1;
    // ajout de 2 éléments
    m1[123456789]=1;
    // .at(clé) = valeur n'est pas utilisable pour une création mais seulement
    // pour une modification ou une consultation
    m1[987654321]=2;
    m1[91]=789;
    // affichage des éléments de m1
    map<int,int>::iterator pos; // création d'un itérateur
    // boucle pour parcourir toutes les clés
    for (pos = m1.begin(); pos != m1.end(); ++pos){
        // affiche les couples (clé)-(valeur)
        cout << "m1[" << pos->first << "] = " << pos->second << endl;
    }
    m1[123456789]+=10; // un élément est modifié
    cout << "m1[123456789] = " << m1[123456789] << endl;
    // affiche la taille actuelle de m1
    cout << "taille de m1: " << m1.size() << endl;
    // cherche si une clé est présente (affiche 1 si oui et 0 sinon)
    cout << "cle[123456789]: " << m1.count(123456789) << endl;
    // renvoie la valeur associée à la clé et 0 sinon
    cout << "cle[123456789]: " << m1.find(123456789)->second << endl;
    m1.clear(); // efface m1
    // affiche la nouvelle taille de m1
    cout << "taille de m1: " << m1.size() << endl;
    return 0;
}
```

Le code ci-dessus montre comment créer, modifier, parcourir et effacer des **map**. Le mode de fonctionnement des **map** est proche de celui des **vectors** par certains côtés mais le fait que l'adressage interne de ses éléments passe par une clé et non pas un index entier croissant complique un peu le parcours dans l'objet. Ceci se fait en deux étapes :

- création d'un itérateur adapté à la **map**, ici c'est la variable **pos**. Cet itérateur **pos** est en fait un pointeur sur une paire **first** et **second**. Nous utilisons donc l'opérateur **->** pour obtenir le contenu de ces deux membres.
- parcours de toutes les valeurs de l'itérateur à l'aide d'une boucle entre les bornes de la **map** depuis **m1.begin()** jusqu'à **m1.end()**

Comme **vector** et **string** que nous verrons ensuite, **map** est une classe et ici **m1** est donc un objet c'est-à-dire une instance de cette classe. C'est pourquoi là aussi les méthodes sont appelées sur l'objet **m1** avec le "." de la même façon qu'avec le **vector**.

```

m1[91] = 789
m1[123456789] = 1
m1[987654321] = 2
m1[123456789] = 11
taille de m1: 3
clé[123456789]: 11
taille de m1: 0

```

1.8.4 La classe string

Un autre type de classe conteneur est très fréquemment utilisé en C++ pour traiter les chaînes de caractères : la classe **string**. Elle est préférée au tableau de char du C ⁶ et possède de nombreuses fonctions de traitement analogues à celles de la classe **vector** comme nous allons le voir.

```

// code_22.cpp
#include <iostream>
#include <string>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    string chaine1 = "Hello";
    string chaine2 = "world!";
    string chaine3 = chaine1 + " " + chaine2;
    chaine1.clear();
    chaine2.at(0)='T';
    chaine2.at(5)=' ';
    chaine2.push_back('!');
    cout << "chaine1: " << chaine1 << ", taille: " << chaine1.size() << endl;
    cout << "chaine2: " << chaine2 << ", taille: " << chaine2.size() << endl;
    cout << "chaine3: " << chaine3 << ", taille: " << chaine3.size() << endl;
    cout << "chaine3[0]: " << chaine3.at(0) << endl;
    cout << "chaine3.substr(2,3): " << chaine3.substr(2,3) << endl;
    return 0;
}

```

Le code ci-dessus commence par la création de deux variables de type **string** (chaine1 et chaine2) contenant chacune une chaîne de caractères à l'initialisation. La variable chaine3 est créée par concaténation (surcharge de l'opérateur +) de chaine1 et chaine2. Le contenu de chaine1 est ensuite effacé, et celui de chaine2 modifié à la fois dans son contenu et dans sa taille. La dernière commande montre l'utilisation de la méthode **substr(a,b)** qui renvoie une sous-chaîne depuis l'indice de position a et de longueur b.

```

chaine1: , taille: 0
chaine2: Torld !, taille: 7
chaine3: Hello world!, taille: 12
chaine3[0]: H
chaine3.substr(2,3): llo

```

Remarque : pour créer des **vector** de **string** il suffit de spécifier le type contenu à la construction ou à l'initialisation mais dans ce cas le choix des opérateurs diffère de ce que nous avons vu dans [1.13.2](#) car les { } et les () se comportent alors de la même façon :

```

vector v1{3, "abc"}; // {"abc","abc","abc"}
vector v1(3, "abc"); // {"abc","abc","abc"}

```

1.9 Les fonctions

En C++ les fonctions suivent la structure suivante :

```
type nom (arguments){
    Corps de la fonction...
    retour instance de type
}
```

type est le type de variable renvoyé par la fonction (par exemple **double**). Celui-ci est renvoyé depuis le corps de la fonction par la commande **return** qui réalise en même temps la sortie de la fonction par le programme. Les **arguments** constituent la "signature" d'une fonction et sont les séquences **type - nom** qu'elle va utiliser. Le corps de la fonction réalise les traitements requis. Pour appeler une fonction il suffit ensuite d'utiliser son nom, obligatoirement suivi des **()** contenant les éventuels arguments requis. Le code suivant illustre ces différents éléments :

```
// code_15.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int fonction_1(int value){// fonction_1 déclarée et définie avant le main
    return value*2;
}
int fonction_2(int value); // prototype de fonction_2 déclaré avant le main
int main(){
    int a=10;
    cout << "fonction_1(" << a << ") : " << fonction_1(a) << endl;
    cout << "fonction_2(" << a << ") : " << fonction_2(a) << endl;
    return 0;
}
int fonction_2(int value){ // fonction_2 définie après le main
    return value*3;
}
```

Ce qui donne à l'exécution :

```
fonction_1(10) : 20
fonction_2(10) : 30
```

Une des règles de la création de fonctions est que leur prototype (ou déclaration) doit apparaître avant le **main**, faute de quoi lors de la compilation celui-ci ne saura pas comment les utiliser. La définition de la fonction (implémentation du corps de la fonction) peut être placée directement avant (cas de la **fonction_1**) ou être placée après le main dès lors que le prototype est placé avant (cas de la **fonction_2**).

Remarque : les règles de nommage des fonctions sont proches de celles des variables et peuvent également suivre des conventions particulières selon les développeurs.

Il est possible d'introduire plusieurs fonctions ayant le même nom **à condition que leur signature (donc leurs arguments) diffère** (c'est le principe de la "surcharge"). Des différences portant uniquement sur le type de retour ne suffisent pas à surcharger des fonctions.

Remarque : il n'est pas possible de déclarer des fonctions directement dans le **main** car c'est lui-même une fonction.

Les paramètres sont passés par défaut par valeur en C++. Pour pouvoir modifier le contenu d'une variable dans une fonction le plus simple est de passer une référence (cf. [1.12.2](#)) sur cette variable :

```
// code_15_1.cpp
#include <iostream>
#include <vector>
using namespace std;

void fa1(int value){
    value=value+20;
}

void fa2(int& value){
    value=value+20;
}

void fv1(vector<int> value){
    value.at(0)=value.at(0)+20;
}

void fv2(vector<int>& value){
    value.at(0)=value.at(0)+20;
}

int main(){
    int a=10;
    vector<int> v{1,2,3};
    cout << "a : " << a << "\t et v.at(0) : " << v.at(0) << endl;
    fa1(a);fv1(v);
    cout << "a : " << a << "\t et v.at(0) : " << v.at(0) << endl;
    fa2(a);fv2(v);
    cout << "a : " << a << "\t et v.at(0) : " << v.at(0) << endl;
    return 0;
}
```

Le résultat est le suivant :

```
a : 10    et v.at(0) : 1
a : 10    et v.at(0) : 1
a : 30    et v.at(0) : 21
```

Remarque : pensez à utiliser l'option **-std=c++11** qui est nécessaire pour pouvoir initialiser directement un vecteur avec des valeurs, ici **{1, 2, 3}**.

1.10 Fonctions de calcul numérique

1.10.1 Les fonctions mathématiques de base

La bibliothèque **math** donne l'accès à de nombreuses fonctions mathématiques très utiles :

```
#include <iostream>
#include <math.h>
using namespace std;
int main(){
```

```

double var1=10.3;
double var2=9;
double var3=-10.5;
cout << sqrt(var2) << endl;
cout << pow(var2,2) << endl;
cout << floor(var1) << endl;
cout << ceil(var1) << endl;
cout << fabs(var3) << endl;
cout << exp(1) << endl;
cout << log(exp(1)) << endl;
cout << log10(100) << endl;
cout << sin(var1) << endl;
cout << asin(1) << endl;
return 0;
}

```

A l'exécution nous obtenons :

```

3
81
10
11
10.5
2.71828
1
2
-0.767686
1.5708

```

1.10.2 Les générateurs de nombres pseudo-aléatoires

Il est souvent utile, particulièrement dans le domaine de la simulation numérique, d'avoir accès à des générateurs de nombres (pseudo)aléatoires ou PRNG. Nous allons voir ici comment les utiliser pour générer des valeurs selon deux lois de probabilités communes : la loi normale et la loi uniforme.

La bibliothèque C++ permettant de générer des nombres aléatoires est nommée **random** et fait partie de la **stl** comme **iostream**. Comme l'exploration de cette bibliothèque serait très longue nous allons donner ici un moyen simple de la faire fonctionner :

```

// code_18.cpp
#include <iostream>
#include <random>
using namespace std; // utilisation de l'espace de nom de std

// création du PRNG
default_random_engine dre;

// générateur suivant une loi normale
normal_distribution <double> nd;

// générateur suivant une loi uniforme
uniform_real_distribution <double> urd;
double getGaussian(){
    return nd(dre);
}

```

```
// fonction personnalisée -> loi normale
double getUniform(){ // fonction personnalisée -> loi uniforme
    return urd(dre);
}

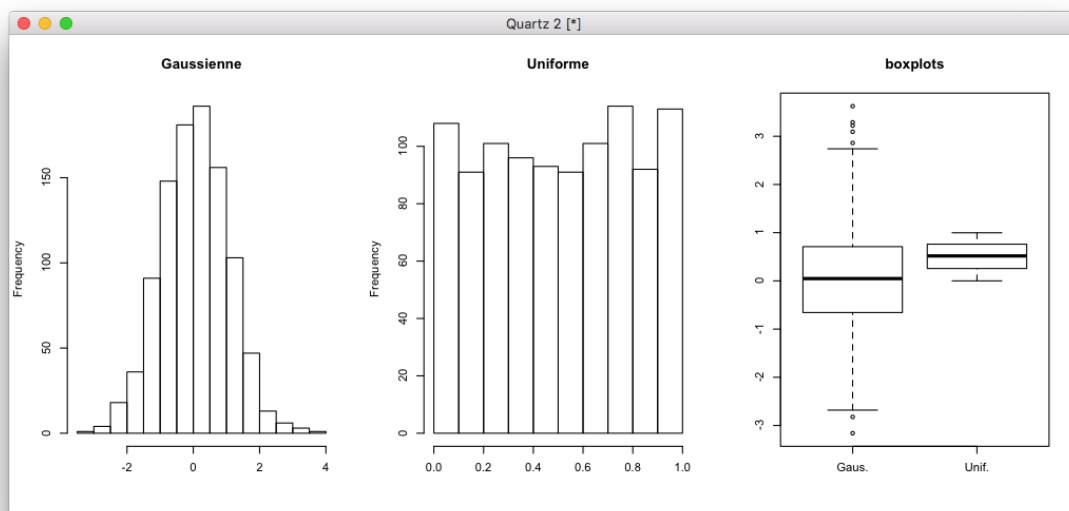
int main(){
    for (int i=0;i<1000;i++){
        cout << getGaussian() << " " << getUniform() << endl;
    }
    return 0;
}
```

Le code ci-dessus crée une variable **dre** de type **default_random_engine** qui est le moteur de génération des nombres aléatoires. Il faut ensuite lui adjoindre une variable qui l'utilisera pour générer des nombres répartis selon une loi normale et qui sera de type **normal_distribution** et une autre qui générera, toujours avec ne des nombres répartis selon une loi uniforme : **uniform_real_distribution**. Ces variables sont créées avant la fonction main et elles seront donc utilisables dans main mais pour simplifier leur utilisation nous avons créé, toujours avant la fonction main, deux fonctions simples à utiliser qui génère à chaque appel un nombre aléatoire suivant une loi normale : **getGaussian()** ou selon une loi uniforme : **getUniform()**. Nous réalisons ensuite une boucle qui génère 1000 couples de valeurs aléatoires de chaque distribution, placées côte à côte de façon à former deux colonnes.

```
code_18 > testDist.txt
```

Le code exécuté ici dans la console compile et exécute le code mais en redirigeant les sorties de l'écran vers un fichier, à l'aide du symbole > que l'on nomme **testDist.txt**.

Nous verrons plus loin [2](#) comment visualiser les données avec R mais la figure suivante donne un aperçu de ce que donne le résultat des tirages aléatoires :



Les résultats de cette figure sont cohérents avec les fonctions utilisées dans le code précédent et par la suite nous emploierons un code de ce type pour générer nos valeurs aléatoires.

Remarque : est le type de valeur renvoyé par défaut quand on crée un générateur donc il est possible de juste préciser <> et ici le résultat sera le même.

1.11 Les exceptions

En C++ comme dans beaucoup d'autres langages il est possible de gérer le déroulement du code en interceptant, en générant et en traitant les exceptions ou erreurs levées par le code. Certaines sources ne peuvent pas être compilés si un code sensible ou vulnérable n'est pas préalablement encadré par un **try-catch**. Il existe toute [une série d'exceptions ou d'erreurs](#) de haut-niveau gérées par C++. La structure d'un **try-catch** est en général construit comme suit :

```
try{
    bloc d'opérations
} catch {
    gestion de l'exception
}
```

La commande **throw** lance un exception qui pourra également être interceptée. Le code suivant montre deux exemples d'exception générées par le système et leur interception :

```
// code_14_1.cpp
#include <iostream>
#include <stdexcept>
#include <fstream>
#include <vector>
using namespace std;

int main (void) {
    vector<int> vec1(1);
    try {
        vec1.at(20)=100; // ok avec at mais pas avec []
    } catch (const exception& e) {
        cerr << "exception sur : " << e.what() << '\n'; // ou cerr
    }
    string str1="salut";
    double val1=0;
    string::size_type st;
    try {
        val1=stod(str1,&st);
    } catch (const exception& e) {
        cerr << "exception sur : " << e.what() << '\n'; // ou cerr
    }
    return 0;
}
```

Après compilation et exécution nous obtenons :

```
exception sur : vector
exception sur : stod: no conversion
```

Lorsque le vecteur **vec1** est appelé à une position n'existant pas, une exception de type **out of range** est levée. En effet il est créé avec une taille de 1 et la position 20 est appelée. Toutefois cette exception n'est pas levée si l'on utilise l'opérateur **[]**. Lorsque la conversion de chaîne de caractère tente d'obtenir la valeur **double** de "salut" une exception est également levée pour signaler que ce n'est pas possible en pratique.

Remarque : d'après Bjarne Stroustrup les exceptions ne concernent pas les erreurs de bas niveau "The Design and Evolution of C++" (Addison Wesley, 1994), "low-level events, such as arithmetic overflows and divide by zero, are assumed to be handled by a dedicated lower-level mechanism rather than by exceptions. This enables C++ to match the behaviour of other languages when it comes to arithmetic. It also avoids the problems that occur on heavily pipelined architectures where events such as divide by zero are asynchronous."

1.12 Gestion de la mémoire avec C++

1.12.1 Des piles et des tas

La mémoire est gérée "manuellement" en C++ mais en pratique le développeur n'a pas toujours à se soucier d'allouer et de désallouer la mémoire nécessaire à l'exécution des programmes. Dans les exemples qui précèdent, à aucun moment nous n'avons eu à réaliser ce genre de manipulation explicitement et c'est le langage qui s'en est chargé pour nous. Il est néanmoins parfois nécessaire de réaliser manuellement des opérations de gestion de la mémoire. C'est notamment le cas avec le langage C et l'utilisation de variables dimensionnées dynamiques. C++ a grandement simplifié ces procédures puisqu'il est désormais possible de travailler avec des tableaux dynamiques de grande taille sans passer par des allocations et désallocations explicites de mémoire. Lors des premiers cours en C on apprend en général que la mémoire est divisée en deux parties distinctes : la pile (**stack**) et le tas (**heap**). Pour connaître la taille en ko de la pile par défaut (qui est beaucoup plus petite que le tas en général) nous pouvons utiliser dans la console la

[commande suivante](#) :

```
ulimit -s  
8192
```

Remarque : pour les OS Windows on utilisera plutôt le **Windows System Resource Manager** dans la mesure où il ne reconnaît pas cette commande.

En raison de leurs tailles respectives, il est conseillé de réserver la pile aux variables de petite taille. La pile est surtout utilisée pour les variables locales et les appels de fonctions et **en théorie** il vaut mieux réserver le tas pour les segments de données plus lourds. **En pratique** pour les variables dynamiques et de grande taille nous utiliserons plutôt ici des classes conteneurs spécialisées en C++ dont la gestion mémoire est optimisée.

1.12.2 Des pointeurs et des références

Quand une variable est déclarée C++ réserve une portion de mémoire à son identificateur, correspondant à la taille prévisible occupée par cette variable. Par exemple écrire **char* a** réserve un emplacement d'un octet à la variable **a**. Ecrire ensuite **a=12** place la valeur 12 à cet emplacement. L'adresse de cet emplacement peut être trouvée grâce à l'opérateur de référence : **&**. Une référence ne peut pas être modifiée pour une variable et sa durée de vie est celle de la variable. Pour pouvoir travailler ensuite directement sur cet emplacement en mémoire nous utilisons l'opérateur d'indirection (ou déréférencement) ***** qui désigne un pointeur sur cette adresse :

```
// code_16.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    char a; // définition d'un char
    a=12; // affectation de la valeur 12
    char* add = &a; // création d'un pointeur contenant l'adresse de a
    // &add affiche sur quelle adresse "pointe" le pointeur : celle de a
    cout << "adresse de a: " << &add << endl;
    return 0;
}
```

Le code compilé et exécuté affiche l'adresse de la variable a :

```
adresse de a: 0x61fe10
```

Remarque : l'adresse de a est exprimée en hexadécimal (commence par 0x) ce qui est une convention depuis le langage C.

Un pointeur est donc une variable dont la particularité est donc de contenir une adresse mémoire d'un type donné. Cette adresse est en général une adresse virtuelle attribuée par le processus en cours. Elle permet de "pointer" sur un contenu et donc aussi bien de le lire que de le modifier. Comme nous l'avons vu, quand une variable est créée à l'aide d'un opérateur d'affectation, un emplacement lui est réservé en mémoire par le programme via l'OS afin que d'autres processus ne risquent pas d'en modifier le contenu. Par exemple lorsque l'on écrit :

```
int variable_1=12;
```

Le programme attribue une adresse de format **int** automatiquement à variable_1 et stocke le contenu affecté à cette variable (ici 12) à cette adresse. On peut visualiser la situation comme suit :

nom	adresse	contenu (décimal)
variable_1	0x61fe14	12

Un **pointeur** sur la variable_1 est alors **toute variable qui contient cette adresse 0x61fe14**. Dans l'exemple ci-dessous nous créons deux pointeurs vers une même variable.

Remarque : traditionnellement en informatique la capacité mémoire [RAM \(Random Access Memory\)](#) s'exprime en puissances de deux : par exemple 1 ko = 1024 octets = 2^{10} octets mais [l'alignement sur un système décimal](#) a conduit à un mésusage des termes ko, Mo, Go, To ... En théorie nous devrions donc maintenant nommer les capacités mémoire exprimées en puissances de deux avec les préfixes Kio (ou Ko pour le distinguer de ko), Mio (mebioctet), Gio (gibioctet), Tio (tebioctet) et réserver les ko, Mo, Go, To ... aux puissances de dix mais en pratique cette nouvelle nomenclature est très peu prise en compte. Pour résumer avec la nouvelle nomenclature : 1 ko = 1000 octets = 10^3 octets et 1 Kio = 1024 octets = 2^{10} octets - 1 Mo = 1000 000 octets = 10^6 octets et 1 Mio = 1 048 876 octets = 2^{20} octets - 1 Go = 1000 000 000 octets = 10^9 octets et 1 Gio = 1 073 741 824 octets = 2^{30} octets etc.

Le code ci-dessous montre comment créer deux pointeurs différents pointant vers la même variable :

```
// déclaration et affectation de variable_1
int variable_1 = 12;
// création d'un pointeur de type int sur variable_1
int* pointeur_1 = &variable_1;
// création et initialisation d'un pointeur de type int non affecté
int* pointeur_2 = NULL;
// affectation de pointeur_2 à l'adresse de variable_1
pointeur_2 = &variable_1;
```

- **pointeur_1** contient l'adresse de **variable_1** et "sait" que cette adresse contient une variable de type **int**.
- **pointeur_2** est également un pointeur de type **int** mais ne pointe sur rien (**NULL**) lors de son initialisation. Il faudra donc ensuite l'affecter à une adresse valide (**pointeur_2 = &variable_1**) pour pouvoir l'utiliser.

Le code suivant résume la création et l'utilisation de pointeurs sur une variable de type **int** :

```
// code_17.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    // déclaration et affectation de variable_1
    int variable_1=12;
    // création d'un pointeur sur variable_1
    int* pointeur_1=&variable_1;
    // création et initialisation d'un pointeur non affecté
    int* pointeur_2=NULL;
    // affectation de pointeur_2 à variable_1
    pointeur_2=&variable_1;
    // adresse hexa de variable_1
    cout << "1) &variable_1 : " << &variable_1 << endl;
    // contenu de variable_1 à l'aide d'un pointeur sur l'adresse
    cout << "2) *(&variable_1) : " << *(&variable_1) << endl;
    // adresse de pointeur_1
    cout << "3) &pointeur_1 : " << &pointeur_1 << endl;
    // adresse de pointeur_2
    cout << "4) &pointeur_2 : " << &pointeur_2 << endl;
    // incrémente le contenu du pointeur_1
    *pointeur_1+=1;
    // affiche le contenu de variable_1
    cout << "5) [*pointeur_1+=1;] => variable_1 : " << variable_1 << endl;
    // incrémente le contenu du pointeur_2
    *pointeur_2+=1;
    // affiche le contenu de variable_1
    cout << "6) [*pointeur_2+=1;] => variable_1 : " << variable_1 << endl;
    // incrémente le contenu du contenu de l'adresse du
    // pointeur_1 (donc pointeur_1)
    *(*(&pointeur_1))+=1;
    // affiche le contenu de variable_1
    cout << "7) [*(*(&pointeur_1))+=1;] => variable_1 : " << variable_1 << endl;
    return 0;
}
```

une fois compilé et exécuté le programme affiche les sept étapes suivantes :

```

1) &variable_1 : 0x61fe1c
2) *(&variable_1) : 12
3) &pointeur_1 : 0x61fe10
4) &pointeur_2 : 0x61fe08
5) [*pointeur_1+=1;] => variable_1 : 13
6) [*pointeur_2+=1;] => variable_1 : 14
7) [*(*&pointeur_1)+=1;] => variable_1 : 15

```

L'évolution des sept étapes ci-dessus peut également être visualisée dans le tableau suivant :

nom	adresse	contenu	étape
variable_1	0x61fe1c	12	1 et 2
pointeur_1	0x61fe10	0x61fe1c	3
pointeur_2	0x61fe08	0x61fe1c	4
variable_1	0x61fe1c	13	5
variable_1	0x61fe1c	14	6
variable_1	0x61fe1c	15	7

Les notions importantes à comprendre ici sont :

- la notion d'adresse d'une variable
- la notion d'opérateur d'indirection
- la notion de pointeur sur l'adresse d'une variable
- comment créer, afficher et manipuler adresses et pointeurs en C++ sur des **int**

1.12.3 Les pointeurs en C (optionnel)

Remarque : Ce paragraphe est optionnel pour le cours. En effet c'est le C++ qui est essentiellement abordé ici et pas directement le langage C. Néanmoins pour ceux que cela pourra intéresser, j'ai jugé bon de l'ajouter à titre de complément d'information.

Un pointeur est une variable dont la particularité est de contenir une adresse mémoire. Cette adresse est en général une adresse virtuelle attribuée par le processus. Elle permet de "pointer" sur un contenu et donc aussi bien de le lire que de le modifier. Plus concrètement, quand une variable est créée à l'aide d'un opérateur d'affectation, un emplacement lui est réservé en mémoire par le programme (ou processus) via l'OS afin que d'autres processus ne risquent pas d'en modifier le contenu. Par exemple lorsque l'on écrit :

```
int variable_1=12;
```

Le programme attribue une adresse automatiquement à **variable_1** et stocke le contenu affecté à cette variable (ici 12) à cette adresse qui pour des raisons historiques est en général exprimée en hexadécimal. On peut visualiser la situation comme suit :

nom	adresse	contenu
variable_1	0x61fe1c	12

Un pointeur sur **variable_1** est alors toute variable qui contient cette adresse **0x7fff5f199a8c**. En C la manière la plus simple de récupérer l'adresse d'une variable est d'utiliser l'opérateur d'adresse **&** et pour créer directement une variable de type pointeur il faudra utiliser l'opérateur d'indirection ***** Par exemple le code suivant crée une variable et deux pointeurs sur cette variable :

```
int variable_1 = 12; // déclaration et affectation de variable_1
int* pointeur_1 = &variable_1; // création d'un pointeur de type int sur
variable_1
int* pointeur_2 = NULL; // création et initialisation d'un pointeur de type int
vide
pointeur_2 = &variable_1; // affectation de pointeur_2 à l'adresse de variable_1
```

pointeur_1 contient l'adresse de **variable_1** et "sait" que cette adresse contient une variable de type **int**.

pointeur_2 est également un pointeur de type **int** mais ne pointe sur rien (**NULL**) lors de son initialisation. Pour pouvoir l'utiliser il faudra donc ensuite l'affecter à une adresse valide avec :

```
pointeur_2 = &variable_1;
```

Pour illustrer ces notions nous pouvons utiliser le code suivant qui donne un aperçu de la création et de l'utilisation de pointeurs sur une variable de type **int** :

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    // déclaration et affectation de variable_1
    int variable_1=12;
    // création d'un pointeur sur variable_1
    int* pointeur_1=&variable_1;
    // création et initialisation d'un pointeur non affecté
    int* pointeur_2=NULL;
    // affectation de pointeur_2 à variable_1
    pointeur_2=&variable_1;
    // adresse hexa de variable_1
    printf("1 - &variable_1 : %p\n",&variable_1);
    // contenu d'un pointeur sur l'adresse de variable_1
    printf("2 - *(&variable_1) : %d\n",*(&variable_1));
    // adresse de pointeur_1
    printf("3 - &pointeur_1 : %p\n",&pointeur_1);
    // adresse de pointeur_2
    printf("4 - &pointeur_2 : %p\n",&pointeur_2);
    // incrémente le contenu du pointeur_1
    *pointeur_1+=1;
    // contenu de variable_1
    printf("5 - [*pointeur_1+=1;] => variable_1 : %d\n",variable_1);
    // incrémente le contenu du pointeur_2
    *pointeur_2+=1;
    // contenu de variable_1
    printf("6 - [*pointeur_2+=1;] => variable_1 : %d\n",variable_1);
    // incrémente le contenu du contenu de l'adresse
    // du pointeur_1 (donc pointeur_1)
    *(*(&pointeur_1))+=1;
    // contenu de variable_1
    printf("7 - [*( *&pointeur_1)+=1;] => variable_1 : %d\n",variable_1);
```

```
    return 0;
}
```

Une fois compilé et exécuté le programme affiche les 7 étapes suivantes :

```
1 - &variable_1 : 000000000061FE1C
2 - *(&variable_1) : 12
3 - &pointeur_1 : 000000000061FE10
4 - &pointeur_2 : 000000000061FE08
5 - [*pointeur_1+=1;] => variable_1 : 13
6 - [*pointeur_2+=1;] => variable_1 : 14
7 - [*(*(&pointeur_1))+=1;] => variable_1 : 15
```

En résumé nous avons abordé ici brièvement en C :

- la notion d'adresse d'une variable
- la notion d'opérateur d'indirection
- la notion de pointeur sur l'adresse d'une variable
- comment créer, afficher et manipuler adresses et pointeurs en C sur des **int**

1.12.4 Fonctions, pointeurs et références

Les variables passées à des fonctions sont en général dupliquées en début de fonction et leurs valeurs copiées dans la variable locale dont le nom est utilisé en paramètre. Pour cette raison les modifications effectuées sur les variables dans le corps de la fonction disparaissent en même temps que ces variables quand le programme sort de la fonction. Si l'on veut modifier dans la fonction elle-même la valeur de la variable passée en argument, le plus simple est de lui transmettre l'adresse de cette variable. Comme nous l'avons vu ceci implique l'utilisation du symbole **&** :

```
// code_17_1.cpp
#include <iostream>
#include <vector>
using namespace std; // utilisation de l'espace de nom de std
void f1(int val){val=val*3;}
void f2(int& val){val=val*3;}
void f3(vector<int> vec){vec[0]=vec[0]*3;}
void f4(vector<int>& vec){vec[0]=vec[0]*3;}
```

Le début du programme contient les appels aux bibliothèques spécialisées nécessaires puis la définition des fonctions. f1 et f2 seront utilisées pour étudier les variables simples, f3 et f4 pour étudier les variables dimensionnées (cf. [1.8](#)). Ces fonctions ne renvoient qu'un type **void** (donc rien en pratique). Elles sont identiques 2 à 2 à ceci près que le **&** est utilisé dans la signature de f2 et de f4. Cela signifie que les argument passés ne seront pas dupliqués mais passés par leur référence (leur adresse) et la conséquence de ceci est que les modifications réalisées sur les variables passées en argument dans le corps de la fonction vont être effectives et perdurer après l'appel à la fonction. Nous ajoutons à la suite le code source de la fonction **main()** :

```
// test_17_1.cpp suite et fin ...
int main(){
    int valeur_1=10;
    cout << "valeur_1 initiale : " << valeur_1 << endl;
    f1(valeur_1);
    cout << "valeur_1 apres f1 : " << valeur_1 << endl; // inchangé
```

```

    f2(valeur_1);
    cout << "valeur_1 apres f2 : " << valeur_1 << endl; // changé
    vector<int> vecteur_1={10};
    cout << "vecteur_1[0] initial : " << vecteur_1[0] << endl;
    f3(vecteur_1);
    cout << "vecteur_1[0] apres f3 : " << vecteur_1[0] << endl; // inchangé
    f4(vecteur_1);
    cout << "vecteur_1[0] apres f4 : " << vecteur_1[0] << endl; // changé
    return 0;
}

```

Après compilation et exécution nous voyons bien la différence entre les arguments passés par référence (& dans f1 et f2) et ceux qui sont passés par valeur (f1 et f3).

```

valeur_1 initiale : 10
valeur_1 après f1 : 10 // inchangé
valeur_1 après f2 : 30 // changé
vecteur_1[0] initial : 10
vecteur_1[0] après f3 : 10 // inchangé
vecteur_1[0] après f4 : 30 // changé

```

Il est également possible de passer les valeurs en utilisant les pointeurs. Dans le programme **code_17_2.cpp** nous créons tout d'abord deux fonctions f1 et f2. La première prend un entier par valeur et la seconde un pointeur sur un entier et les deux réalisent la même opération :

```

// code_17_2.cpp
#include <iostream>
#include <vector>
using namespace std; // utilisation de l'espace de nom de std

void f1(int val){val=val*3;}
void f2(int* val){*val=*val*3;}

```

Pour tester ces deux fonctions nous utilisons un pointeur sur un entier **valeur_1** qui pointe sur **val**. **valeur_1** contient donc l'adresse de **val** et logiquement ***valeur_1** contient la valeur de **val**. Les fonctions sont ensuite appelées respectivement sur le contenu du pointeur et sur le pointeur lui-même.

```

// code_17_2.cpp suite et fin ...
int main(){
    int val=10;
    int* valeur_1=&val;
    cout << "valeur_1 initiale : " << *valeur_1 << endl;
    f1(*valeur_1);
    cout << "valeur_1 après f1 : " << *valeur_1 << endl;
    f2(valeur_1);
    cout << "valeur_1 après f2 : " << *valeur_1 << " et val : " << val << endl;
    return 0;
}

```

Après compilation et exécution nous voyons bien que le passage de l'adresse du pointeur avec f2 génère une modification du pointeur **valeur_1** et donc aussi de la variable **val** :

```

valeur_1 initiale : 10
valeur_1 après f1 : 10
valeur_1 après f2 : 30 et val : 30

```

1.13 Variables dimensionnées, pointeurs et références

Nous ne développerons pas trop longuement ce chapitre. En effet chaque classe ou type de variable (et même les fonctions) peut être adressé par des pointeurs. Néanmoins les classes de variables dimensionnées (**vector**, **map** ...) sont déjà optimisées en C++ et les pointeurs sur les variables dimensionnées étaient surtout utiles en C ou dans des versions antérieures. Dans le programme **code_22_1.cpp** nous créons un pointeur sur un vecteur d'entier pVec. Celui-ci est construit à l'aide du mot clé **new**. Comme ce pointeur sur un objet est créé (en général sur le tas) avec le mot clé **new** il faudra qu'il soit également effacé explicitement à l'aide de l'instruction **delete** afin de libérer la mémoire à la fin de main.

```

// code_22_1.cpp
#include <iostream>
#include <vector>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    vector<int>* pVec = new vector<int>();
    pVec->push_back(10);pVec->push_back(20);
    for (int i=0;i<pVec->size();i++){
        cout << "pVec(" << i << ")=" << (*pVec)[i] << endl;
    }
    delete pVec;
    return 0;
}

```

Comme nous le voyons dans le code ci-dessus un pointeur sur une variable créée avec **new** implique quelques modifications syntaxique pour être utilisé :

- l'appel aux attributs et méthodes de l'objet ne se fait plus avec le "." mais avec la flèche ->
- pour accéder à la valeur d'indice i de pVec nous avons deux possibilités (1) soit comme ici avec un pointeur sur le vecteur (***pVec**) suivi de l'opérateur d'indice **[]**, (2) soit en utilisant la méthode **at()** de la façon suivante : **pVec->at(i)**. Le résultat sera similaire.

Après compilation et exécution le résultat est le suivant :

```

pVec(0)=10
pVec(1)=20

```

Comme nous le voyons les pointeurs sur les variables dimensionnées ne posent pas de problème particulier à condition de gérer les **new** et les **delete** dans un rapport de 1 pour 1.

1.14 Quelques conversions de type utiles : string, int et double

Il existe des méthodes incluses dans la **std** permettant des conversions de types qui sont souvent utiles notamment lorsque l'on travaille avec les entrées sorties (clavier, fichier ...). Le cas se pose souvent de conversion **string** → numérique et réciproquement :


```
// code_23.cpp
#include <iostream>
#include <string>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    cout << "conversion chaine vers valeur numerique : " << endl;
    string string1="123.456";
    string::size_type st;
    // st equivalent à un entier non signé
    // ici st contient le nombre de caractères de la chaine
    long double double1 = stold(string1,&st);
    int int1 = stoi(string1,&st);
    cout << "De string1 on a double1: " << double1 << " et int1: " << int1 <<
endl;
    cout << "conversion valeur numerique vers chaine : " << endl;
    string string2 = to_string(int1);
    string string3 = to_string(double1);
    cout << "De int 1 et double1 on a string2: " << string2 << " et string3: "
<< string3 << endl;
    return 0;
}
```

Le code ci-dessus montre comment extraire des valeurs numériques double1 et int1 à partir d'une chaîne contenant une valeur string1 interprétable. Puis dans un deuxième temps comment créer deux chaînes string2 et string3 à partir de ces deux valeurs numériques ⁷.

```
conversion chaine vers valeur numerique :
De string1 on a double1: 123.456 et int1: 123
conversion valeur numerique vers chaine :
De int 1 et double1 on a string2: 123 et string3: 123.456000
```

Il existe de nombreuses méthodes de conversions, notamment celles qui font appel à des flux mais l'essentiel des besoins de ce cours peuvent être réalisés à l'aide des méthodes présentées ci-dessus.

1.15 Les variables composites et les définitions de types

1.15.1 Les struct, un premier pas vers les classes

Historiquement les premières variables composites en C sont les **struct**. Ces structures permettent d'associer plusieurs types prédéfinis et de les rendre adressables directement par le biais des membres d'une seule variable. Considérons l'exemple suivant :

```
// code_24.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
struct cellule{
    char type[20];
    double diametre;
};
int main(){
    cellule c1={"neurone",15.5};
    cout << "c1.type : " << c1.type << " et c1.diametre : " << c1.diametre <<
endl;
    return 0;
}
```

Le code ci-dessus crée une structure de données mixte nommée **cellule**. Cette **struct** comporte deux champs : un tableau de caractères (type) et un **double** (diametre). Ainsi il est possible de créer des variables contenant des ensembles hétérogènes de valeurs comme c'est le cas ici pour la variable c1 qui est créée avec les opérateurs de bloc **{ }** auxquels on passe les valeurs des différents champs. Pour accéder à ces champs c'est la notation "." qui est utilisée de façon similaire à la syntaxe des attributs et méthodes pour les objets.

```
c1.type :neurone et c1.diametre : 15.5
```

Nous n'allons toutefois pas ici développer plus avant cet aspect **struct** (ni les **unions**, **enum** ...) qui sont des héritages du langage C. Nous allons plutôt nous attacher à l'approche objet qui étend et améliore ces notions.

1.15.2 Les définitions de types ou typedef

Comme **struct**, **typedef** est un héritage du C utilisable également en C++. C'est une forme d'alias ou de raccourci qui permet d'inventer un type de donnée en vue de simplifier le code qui l'utilise. Potentiellement tous les types de données peuvent être concernés et nous allons illustrer ici deux cas simples de raccourcis **typedef** :

```
// code_24_1.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
typedef unsigned int uInt; //
typedef uInt * intPtr;
int main(){
    uInt a=10; // création d'un unsigned int avec la variable a
    intPtr b=&a; // création d'un pointeur sur un uInt avec b
    a=a+10; // modification du contenu de a
    cout << "b = " << *b << endl; // répercutée à l'adresse de a dans b
    return 0;
}
```

Après compilation et exécution nous obtenons :

```
b = 20
```

Le **typedef** permet donc de gagner en lisibilité à condition d'avoir des alias cohérents et explicites. Il peut être très utile notamment pour créer des alias de types combinés à l'aide des conteneurs de la **stl** comme **vector**, **map** et des types simples du langage ...

1.16 Programmation objet en C++

1.16.1 Généralités

C++ a été conçu par son inventeur Bjarne Stroustrup comme une version aboutie de son "**C plus classes**". Il l'a initialement développé pour améliorer la gestion de calculs distribués en partant de C et en s'inspirant d'autres langages comme [Simula](#) qui est considéré comme le premier langage objet. L'évolution s'est ensuite faite assez logiquement pour aboutir au C++ qui en 1983 devenait un langage à part entière. Quelques années plus tard Python, Java, Delphi et bien d'autres ont été construits sur cette approche objet.

1.16.2 Les classes

La notion de **classe**⁸ peut être définie ici comme un ensemble de règles permettant la construction d'**objets**. L'**objet** est décrit dans ce même contexte comme une instance d'une **classe**.

Pour rendre ces notions un peu plus concrètes on peut comparer une **classe** à une usine permettant de fabriquer des voitures par exemple et une voiture produite dans cette usine à un **objet**. Si l'on se replace dans un contexte de développement logiciel, un objet peut être un bouton, une fenêtre, un **sprite** etc.

L'objectif initial de la programmation orientée objet (**POO**) est de développer des assemblées d'objets qui collaborent en vue de répondre à une problématique donnée. Ces objets sont en général "encapsulés" et communiquent entre eux et avec le reste de leur environnement à travers une interface. Cette notion d'encapsulation qui fait que la complexité du code de fonctionnement de l'objet est "masquée" améliore aussi la réutilisabilité et à un certain degré la lisibilité du code. Si les objets sont correctement développés ils pourront en effet être réutilisés en l'état comme des briques logiciel fiables par la suite⁹.

Chaque objet se voit pourvu de méthodes et d'attributs. Pour reprendre l'exemple de la voiture, celle-ci pourrait par exemple se voir associés les attributs **couleur**, **cylindrée**, **année de fabrication**, **marque**. et les méthodes **démarrer()**, **gauche()**, **droite()**, **avancer()**. Fondamentalement les méthodes sont des fonctions (d'où l'ajout des parenthèses) et les attributs des variables.

1.16.2.1 Comment créer une classe ?

En C++ une classe se crée à l'aide du mot-clé **class**. Idéalement il faut séparer les prototypes (entêtes et déclaration) des classes de leur implémentation. Nous verrons un peu plus loin comment procéder. Pour le moment nous allons modéliser la situation suivante : nous voulons créer une classe **cellule** qui possède deux attributs (numero et diamètre), deux méthodes de consultation pour ces attributs (**getter**) et une méthode de modification (**setter**) pour l'attribut diamètre.

```
class cellule{ // nom de la classe
private:
    int numero; double diametre;
public:
    cellule(){ // constructeur principal (aucun type de retour)
        numero=0; diametre=0;
```

```

    }
    cellule(int numero, double diametre){ // constructeur supplémentaire
(surchargé)
        this->numero=numero; this->diametre=diametre;
    }
    ~cellule(){ // destructeur (appelé quand l'objet est détruit)
} // utile en cas d'allocation dynamique (inutile ici)
    int getNumero()const{ // renvoie le numéro de l'objet
        return numero; // méthode constante (ne modifie pas l'objet)
    }
    double getDiametre()const{ // renvoie le diametre de l'objet
        return diametre; // méthode constante (ne modifie pas l'objet)
    }
    void setDiametre(double diametre){ // modifie le diametre de l'objet
        this->diametre=diametre;
    }
};

```

Une première remarque est que dans le code ci-dessus nous avons choisi l'utilisation de la convention de nommage [lowerCamelCase](#).

La classe comporte un bloc `{ }` et s'achève par un `;"`. Dans ce premier cas prototype et implémentation de la classe sont fusionnés. Les attributs sont déclarés immédiatement après le début de la classe. Par défaut il sont privés (donc non accessible directement depuis le **main()**).

L'implémentation de la classe commence par la surcharge du constructeur (cette méthode particulière ne retourne rien et est prend le nom de la classe). En POO une méthode (en général une fonction ou un opérateur) peut être surchargée c'est-à dire que son nom peut être utilisé pour désigner plusieurs fonctionnalités différentes. La seule contrainte est que leurs arguments diffèrent (des différences sur le type de retour ne suffisent pas comme nous l'avons vu dans le paragraphe [1.10](#)). Par défaut le compilateur génère automatiquement un constructeur sans argument. Nous l'avons ici surchargé en spécifiant que les valeurs des deux attributs devaient être égaux à 0.

Nous avons ensuite ajouté un autre constructeur qui prend en compte deux arguments qui vont être utilisés pour initialiser les valeurs des deux attributs. Comme nous avons mis des noms identiques aux arguments et aux attributs, nous utilisons le pointeur **this->** pour spécifier l'attribut de l'objet courant et le distinguer ainsi de la variable argument.

Un destructeur est ensuite ajouté. Cette méthode (qui comme le constructeur ne retourne rien comme valeur) prend aussi le nom de la classe mais précédé du symbole tilde `~`. Il est utile pour libérer la mémoire en cas d'allocation dynamiques dans la classe mais n'est ajouté ici que par souci d'exhaustivité.

Les deux méthodes qui suivent (**getNumero()** et **getDiametre()**) sont des **getter**. Elles ne modifient pas les valeurs des attributs de l'objet et sont destinée à récupérer leur valeur. Nous pouvons donc ici déclarer ces méthodes constantes avec le mot-clé **const**. L'intérêt de cette qualification est que (i) l'on sait que l'objet n'est pas modifié par cette fonction et que (ii) cela peut améliorer les performances du compilateur qui sait qu'il n'aura pas à modifier les attributs de l'objet à cet endroit.

La méthode suivante (**setDiametre()**) est un **setter**. Elle permet de modifier le contenu d'un attribut et c'est pourquoi elle n'est pas déclarée constante. Pour les mêmes raisons que précédemment nous utilisons ici le pointeur **this->** pour modifier le contenu de l'attribut.

L'intérêt de déclarer **getters** et **setters** en laissant privé l'accès aux attributs est que l'on contrôle mieux la façon dont ils sont modifiés. Il est en effet possible d'utiliser des exceptions pour gérer ces modifications ce qui sécurise le fonctionnement de l'objet et limite les risques d'erreur. Cela crée une interface de communication entre le code interne d'un objet et l'extérieur un peu à la manière d'un agent dans un système multi-agents. L'utilisateur d'une classe n'a donc pas à se soucier de l'implémentation interne des objets mais simplement à échanger avec lui.

1.16.2.2 Comment instancier une classe ?

Le code ci-dessous reprend la création de classe précédente et ajoute une fonction main pour l'exploiter :

```
// code_25.cpp
#include <iostream>
using namespace std; // utilisation de l'espace de nom de std
class cellule{ // nom de la classe
private:
    int numero; double diametre;
public:
    cellule(){ // constructeur principal (aucun type de retour)
        numero=0; diametre=0;
    }
    cellule(int numero, double diametre){ // constructeur supplémentaire
(surchargé)
        this->numero=numero; this->diametre=diametre;
    }
    ~cellule(){ // destructeur (appelé quand l'objet est détruit)
    } // utile en cas d'allocation dynamiquen inutile ici
    int getNumero()const{ // renvoie le numéro de l'objet
        return numero; // méthode constante (ne modifie pas l'objet)
    }
    double getDiametre()const{ // renvoie le diametre de l'objet
        return diametre; // méthode constante (ne modifie pas l'objet)
    }
    void setDiametre(double diametre){ // modifie le diametre de l'objet
        this->diametre=diametre;
    }
};
int main(){
    cellule neurone_1(1,15.5);
    cout << "neurone_1.getDiametre() : " << neurone_1.getDiametre() << endl;
    neurone_1.setDiametre(21.5);
    cout << "neurone_1.setDiametre(21.5)" << endl;
    cout << "neurone_1.getDiametre() : " << neurone_1.getDiametre() << endl;
    return 0;
}
```

Dans la fonction main un objet de la classe cellule est créé avec des valeurs d'initialisation et le contenu de son attribut diametre affiché. A l'aide du **setter**, ce même attribut est modifié puis ré-affiché pour montrer que l'objet a été changé. Après compilation et exécution nous obtenons :

```
code_25
neurone_1.getDiametre() : 15.5
neurone_1.setDiametre(21.5)
neurone_1.getDiametre() : 21.5
```

1.16.2.3 Comment (et pourquoi) séparer prototype et implémentation des classes ?

La réponse à la question "pourquoi" est dans ce cas la suivante : le prototype d'une classe doit être déclaré avant son utilisation dans une fonction main. De plus, il est fréquent qu'une même interface (prototype) puisse impliquer des implémentations différentes. Pour ne pas devoir ré-écrire tout le reste du code il est bien que seule l'implémentation de la classe change. De cette façon les classes et fonctions qui l'utilisent n'ont pas en théorie à être modifiées.

La réponse à la question "comment" est relativement intuitive. Nous allons créer un fichier contenant les prototypes (**.h** pour **header**), un fichier contenant l'implémentation (**.cpp**) et un fichier qui contient l'implémentation de la fonction main.

Le fichier d'entête : cellule.h:

```
// dans cellule.h
#ifndef CELLULE_H
#define CELLULE_H
class cellule{
public:
    cellule();
    cellule(int numero, double diametre);
    ~cellule();
    int getNumero()const;
    double getDiametre()const;
    void setDiametre(double diametre);
private:
    int numero;
    double diametre;
};
#endif
```

Les directives du préprocesseur **#ifndef**, **#define** et **#endif** seront abordées plus en détail dans le chapitre [1.19](#) mais il faut juste savoir ici qu'elles servent notamment à éviter les conflits liés aux importations multiples.

Le fichier d'implémentation : cellule.cpp:

```
// dans cellule.cpp
#include "cellule.h"
#include <iostream>
cellule::cellule(){
    numero=0;
    diametre=0;
}
cellule::cellule(int numero, double diametre){
    this->numero=numero;
    this->diametre=diametre;
}
cellule::~~cellule(){
}
int cellule::getNumero()const{
    return numero;
}
double cellule::getDiametre()const{
    return diametre;
}
```

```

}
void cellule::setDiametre(double diametre){
    this->diametre=diametre;
}

```

Le fichier principal : **code_26.cpp**:

```

// code_26.cpp
#include <iostream>
#include "cellule.h" // utilisation de la classe cellule
using namespace std; // utilisation de l'espace de nom de std

int main(){
    cellule neurone_1(1,15.5);
    cout << "neurone_1.getDiametre() : " << neurone_1.getDiametre() << endl;
    neurone_1.setDiametre(21.5);
    cout << "neurone_1.setDiametre(21.5)" << endl;
    cout << "neurone_1.getDiametre() : " << neurone_1.getDiametre() << endl;
    return 0;
}

```

Il faut ensuite placer ces trois fichiers dans le même dossier et saisir les commandes suivantes :

```

g++ -c cellule.cpp
g++ -c code_26.cpp
g++ -o code_26 cellule.o code_26.o
code_26
neurone_1.getDiametre() : 15.5
neurone_1.setDiametre(21.5)
neurone_1.getDiametre() : 21.5

```

A ce stade il est bon de commencer à se documenter sur l'utilitaire [make](#) qui constitue une aide précieuse pour tout ce qui concerne l'organisation de la compilation et de l'édition de liens pour produire des exécutables. Toutefois cela sort du cadre de ce cours et nous utiliserons les commandes vues au-dessus pour travailler:

- La commande **g++ -c cellule.cpp** génère le fichier objet **cellule.o**.
- La commande **g++ -c code_26.cpp** génère le fichier objet **code_26.o**.
- La commande **g++ -o code_26 cellule.o code_26.o** lie les fichiers objets et produit un exécutable sous la forme du fichier nommé **code_26**.

1.17 Ecriture de fichiers texte

L'objectif de ce cours étant une double initiation au langage C++ et à la modélisation de processus biologiques, il sera nécessaire de savoir sauvegarder des données calculées. Nous allons utiliser ici des commandes d'écriture de fichiers texte car ils sont simple à ré-utiliser ensuite pour des analyses **a posteriori**. La bibliothèque la plus simple à utiliser en C++ pour les écritures de fichier est **ofstream**. Par défaut l'écriture d'un fichier efface le contenu précédent. Si l'on veut ajouter des données à un fichier existant déjà sans en effacer le contenu il faut ajouter des paramètres lors de la construction du fichier à l'aide du constructeur **ofstream** de la façon suivante :

```
// ajoute les éléments à la suite
ofstream fichier("test.txt", ios::out | ios::app);
// efface le contenu du fichier et repart de 0
ofstream fichier("test.txt", ios::out | ios::trunc);
```

Le code ci-dessous utilise la fonction d'écriture par défaut (effacement du fichier et recommencer à 0). Dans un premier temps un objet **ofstream** est créé avec comme paramètre le nom du fichier de sortie (ici **test_27.txt**). Cette fonction renvoie un code d'erreur **null** en cas de problème et c'est pour cela que l'on commence par un test **if** avant de commencer l'écriture. En cas d'échec à l'ouverture, un message est envoyé sur le flux d'erreur standard (**cerr**) . Une chaîne de caractères est alors créée contenant les données à écrire en incluant des caractères de retour à la ligne **\n** pour aligner les données sur des colonnes (la classe **string** est le type de données utilisées pour les flux vers les fichiers tels qu'ils sont utilisés ici). Le contenu de data est ensuite envoyé sur la sortie fichier à l'aide de l'opérateur **<<** puis on le ferme avec la méthode **close()**.

```
// code_27.cpp
#include <iostream>
#include <string>
#include <fstream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    // ouverture en écriture avec effacement du fichier ouvert
    ofstream fichier("test_27.txt");
    if(fichier){
        string data = "x y\n"; data += "0 0\n";
        data += "1 1\n"; data += "2 4\n";
        data += "3 9\n"; data += "4 16\n";
        fichier << data;
        fichier.close();
    } else cerr << "Impossible d'ouvrir le fichier !" << endl;
    return 0;
}
```

Après compilation et exécution nous obtenons la création d'un fichier que nous pouvons visualiser avec la commande **more** :

```
x y
0 0
1 1
2 4
3 9
4 16
```

Remarque : il est très utile comme on le voit ici de maîtriser au mieux les modes de conversion des formats valeur numérique vers les chaînes de caractères et réciproquement car les simulations produisent des valeurs numériques qu'il faut ensuite pouvoir transformer en chaînes pour les écrire. Il existe certaines méthodes spécialisées utilisant d'autres types de flux mais leur mise en œuvre est plus complexe.

1.18 Lecture de fichiers texte

Nous avons vu comment sauvegarder des données dans un fichier texte. Nous allons maintenant voir comment exécuter le processus inverse en lisant des données depuis un fichier pour les stocker dans une variable en C++.


```
// code_28.cpp
#include <iostream>
#include <string>
#include <fstream>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    // ouverture en lecture
    ifstream fichier("test_27.txt"); // , ios::in
    string data; // pour stocker le contenu du fichier
    string line;
    if(fichier){
        while(getline(fichier, line)){
            data+=line+"\n"; // \n est ajouté car le saut de ligne est omis
        }
        fichier.close();
    } else cerr << "Impossible d'ouvrir le fichier !" << endl;
    cout << data;
    return 0;
}
```

Le code ci-dessus montre comment récupérer le contenu d'un fichier texte pour le placer dans une variable de type **string**. Nous allons utiliser un flux d'entrée cette fois-ci : **ifstream**. Une fois que le fichier est ouvert il est lu jusqu'à la dernière ligne avec la commande **getline()** dans une boucle **while**. Quand la dernière ligne est atteinte la boucle s'arrête. Dans la boucle une variable **data** récupère chaque ligne stockée dans une autre variable **line**. Toutes les lignes du fichier sont donc concaténées dans la variable **data** quand la boucle s'achève. Comme le saut de ligne est exclu par la fonction **getline()** nous en ajoutons un à chaque ligne.

```
x y
0 0
1 1
2 4
3 9
4 16
```

Il faut ensuite envisager un traitement dédié de la chaîne de caractères récupérée dans **data** pour en extraire les valeurs numériques et les stocker dans des variables dimensionnées adéquates mais cela n'est pas en l'état étudié dans ce cours.

Il est également possible de récupérer directement des données numériques depuis un flux d'entrée dans un fichier. Si nous plaçons par exemple les données suivantes : 10, 33, 12, 122, 1 et 4 dans une colonne (ou une ligne) d'un fichier **test_29.txt** :

```
// code_29.cpp
#include <iostream>
#include <fstream>
#include <vector>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    ifstream fichier("test_29.txt");
    // un vecteur est créé en passant un flux d'entrée avec un itérateur
    // de type int appliqué sur les données contenues dans fichier
    vector<int> data(istream_iterator<int>(fichier), istream_iterator<int>{});
    // boucle pour afficher le carré de chaque valeur
    for (int i=0;i<data.size();i++){
```

```

        cout << data[i] << "\\t" << data[i]*data[i] << endl;
    }
    fichier.close();
    return 0;
}

```

A l'exécution nous voyons que le résultat est directement exploitable numériquement :

```

10 100
33 1089
12 144
122 14884
1 1
4 16

```

Remarque : il faut ajouter l'argument **-std=gnu++11** à la compilation car la structure invoquée ici n'existait pas en l'état auparavant.

1.19 Directives du préprocesseur

Comme nous l'avons vu à plusieurs reprises, le début (et parfois la fin) des fichiers sources en C++ contient souvent des directives au préprocesseur. C'est l'ensemble des commandes qui commencent par #.

1.19.1 La directive #include

Comme nous l'avons déjà mentionné cette directive demande au compilateur d'ajouter une bibliothèque au code qui suit. La directive **#include** se contente en général d'ajouter directement le code concerné à la compilation. Toutefois le compilateur est assez intelligent pour voir que si les commandes de la bibliothèque ne sont pas utilisées dans le code source, elle n'a pas besoin d'être importée et elle est donc laissée de côté. De même les commentaires ne sont pas repris dans la construction de l'exécutable. Par exemple le code ci-dessous :

```

// code_30.cpp
int main(){
    return 0;
}

```

et le code ci-dessous :

```

// code_30.cpp
#include <iostream>
#include <string>
#include <vector>
int main(){
    return 0;
}
/*
Ah ! Non ! C'est un peu court, jeune homme !
On pouvait dire... oh ! Dieu ! ... bien des choses en somme...

[...] intégralité de "La tirade des nez" (acte 1, scène 4)

Je me les sers moi-même, avec assez de verve,
Mais je ne permets pas qu'un autre me les serve.

```

*/

génèrent tous les deux à la compilation un exécutable de même taille (en l'occurrence 4248 octets). En revanche le code suivant auquel nous ajoutons une ligne:

```
// code_31.cpp
#include <iostream>
#include <string>
#include <vector>
int main(){
    std::vector<int> vec;
    return 0;
}
```

produit un exécutable de 8656 octets. En effet comme la bibliothèque devient nécessaire à la compilation en raison d'un appel à la classe **vector** elle est de fait incluse et augmente la taille de l'exécutable. De même si l'on spécifie **std::vector<std::string> vec;**, le compilateur ajoute cette fois-ci explicitement la bibliothèque string car elle devient nécessaire et la taille de l'exécutable passe alors à 9156 octets.

1.19.2 La directive #define

Cette commande permet de définir une constante du préprocesseur.

```
#define MA_CONSTANTE 123.456
```

Son principal avantage par rapport à une constante (cf. [1.4.5](#)) est qu'elle ne prend pas de place en mémoire. En pratique à chaque fois que le compilateur tombe sur la séquence **MA_CONSTANTE** il la remplace par **123.456**. C'est un outil très pratique mais il faut alors se méfier des effets de bord ¹⁰ que cela peut générer ¹¹ et donc être prudent dans le choix de ses noms de constante.

Remarque : la directive #undef permet d'enlever la constante définie.

1.19.3 Macros et directives conditionnelles

Il est possible de créer des macros dans les directives de compilation et donc de réaliser des tests comme cela a déjà été brièvement évoqué dans le paragraphe [1.16.2](#). Ces macros permettent notamment de réaliser des compilations conditionnelles et de lever certains conflits de noms. Le principe de ces test est qu'il permettent notamment de vérifier qu'une constante a déjà été créée ou non dans l'espace de travail courant. La structure générale en est la suivante :

```
#if CONDITION1
// code à inclure dans la compilation si CONDITION1 réalisée
#elif CONDITION2
// code à inclure dans la compilation si CONDITION2 réalisée
#elif CONDITION3
// etc ...
#endif
```

Où CONDITION1, CONDITION2 et CONDITION3 sont des expressions renvoyant une valeur vraie ou fausse. Très souvent ce sont les commandes #ifdefAAA ou #ifndefAAA qui sont employées et qui signifient en pratique et respectivement **est-ce que AAA existe ?** et **est-ce que AAA n'existe pas ?** Reprenons ici l'exemple développé dans le paragraphe [1.16.2](#) :

```
// dans cellule.h
#ifndef CELLULE_H
#define CELLULE_H
class cellule{
public:
    // ...
private:
    // ...
};
#endif
```

Cela doit être lu de la façon suivante : si la constante **CELLULE_H** n'existe pas (**#ifndef CELLULE_H**) alors définis-la (**#define CELLULE_H**) et prend en compte tout le code qui suit jusqu'au prochain **#endif** pour la compilation.

2 Visualisation des données

C++ offre la possibilité de travailler avec de très nombreuses librairies graphiques gratuites ou payantes que soit en 2D ou en 3D. La combinaison installation/prise en main est toutefois rarement triviale et varie beaucoup selon les besoins et les plateformes à quelques exceptions près. A titre d'informations en voici quelques-unes : [wxwidgets](#), [gnuplot](#), [oxyplot](#), [qwt](#), [vtk](#), [plplot](#), [koolplot](#), [wxPlot](#), [ChartDirector](#), [Boost](#), [PlotLab](#) etc.

Comme l'objectif de ce cours est la prise en main du langage C++ et son application à la modélisation en biologie nous allons opter pour une méthode plus rationnelle. C++ va nous servir à produire et à sauvegarder les données des simulations et nous procéderons à l'analyse avec le logiciel **R** dont l'interface graphique et les outils d'analyse statistique sont très développés et très simples à utiliser. Nous présenterons également en suivant succinctement l'utilisation de **gnuplot** qui possède de puissantes fonctions de visualisation 2D-3D.

2.1 Le logiciel R

2.1.1 Installer et ouvrir une session R

Le logiciel R est [téléchargeable](#) et installable gratuitement sur les principaux OS. C'est une référence dans le domaine des statistiques et de la modélisation et de plus en plus de librairies s'orientent désormais vers les thématiques du [machine learning](#) et de la [représentation des données scientifiques](#).

Pour ouvrir une session R il suffit de l'ouvrir à partir du menu ou de taper R à l'invite de commande (ici sous Windows) :

```
R.exe

R version 4.0.4 (2021-02-15) -- "Lost Library Book"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R est un logiciel libre livré sans AUCUNE GARANTIE.
Vous pouvez le redistribuer sous certaines conditions.
Tapez 'license()' ou 'licence()' pour plus de détails.

R est un projet collaboratif avec de nombreux contributeurs.
Tapez 'contributors()' pour plus d'information et
'citation()' pour la façon de le citer dans les publications.

Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide
en ligne ou 'help.start()' pour obtenir l'aide au format HTML.
Tapez 'q()' pour quitter R.

>
```

2.1.2 Modifier le répertoire de travail

Une fois en ligne de commande il faut ajuster le répertoire de travail de R de façon à se situer de préférence directement dans le répertoire où sont les données à analyser. Deux commandes sont alors utiles : **getwd()** et **setwd()**. La première commande renvoie le répertoire courant et la seconde permet de le modifier. Supposons que nos fichiers de données se trouvent sur le bureau :

```
> getwd()
[1] "C:/Users/sammy"
> setwd("./Desktop/")
> getwd()
[1] "C:/Users/sammy/Desktop"
```

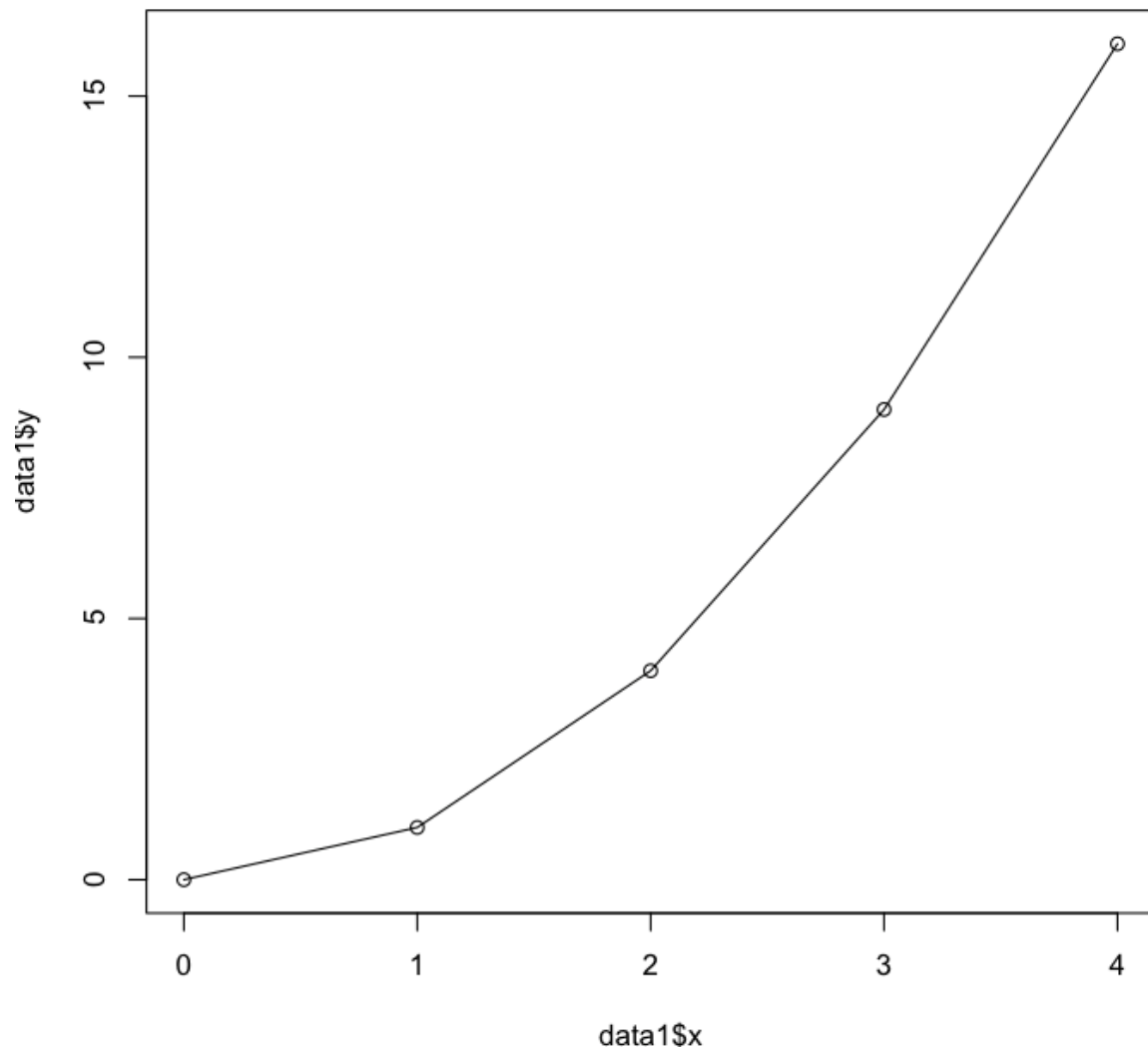
2.1.3 Charger et tracer les données d'un fichier dans une variable en R

2.1.3.1 Courbes simples

Les variables en R sont typées dynamiquement et la commande de base pour lire des données depuis un fichier est **read.table()**. Si nous voulons par exemple étudier le fichier **test_27.txt** généré par le programme **code_27.cpp** dans le paragraphe [1.17](#) nous pouvons taper les commandes suivantes :

```
#plot_27.r
data1=read.table("test_27.txt",header=TRUE)
plot(data1x,data1y,type="l")
points(data1x,data1y)
```

La première commande lit les données du fichier **test_27.txt** et prend en compte le fait que la première ligne ne contient pas des données mais des noms de colonnes (`header=TRUE`). La seconde ligne avec **plot()** trace la courbe avec les valeurs de la première colonne en abscisse et les valeurs de la seconde colonne en ordonnées. La troisième ligne avec **points()** ajoute en plus des points pour chaque valeur. Le résultat est présenté sur la figure suivante :



2.1.3.2 Courbes multiples

Il est souvent utile d'afficher le tracé de plusieurs courbes simultanément pour les comparer par exemple. Il existe une commande R pour le faire qui est la commande **matplot()**. Mais avant nous allons écrire un programme C++ qui génère 3 courbes différentes avec le temps en abscisse, y1, y2 et y3 en ordonnée et stocke les valeurs dans le fichier de données **test_32.txt** sous la forme de 4 colonnes avec leur nom :

```
// code_32.cpp
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <math.h>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    // création des variables pour stocker les données
    vector<double> time,y1,y2,y3;
    // calcul des valeurs en fonction du temps
    for (double t=0;t<10;t+=0.01){
        time.push_back(t);
        y1.push_back(cos(t));
        y2.push_back(sin(t*2)*2);
        y3.push_back(cos(t*2.5)*sin(t*2)*1.5);
    }
    // nombre de points
    int np=time.size();
```

```

// création de la variable data de type string pour écriture
string data="time y1 y2 y3\n";
for (int i=0;i<np;i++){
    data+=to_string(time[i])+" "+to_string(y1[i])+" "+to_string(y2[i])+"
"+to_string(y3[i])+"\n";
}
// ouverture en écriture du fichier
ofstream fichier("test_32.txt");
if(fichier){
    fichier << data;
    fichier.close();
} else cerr << "Impossible d'ouvrir le fichier !" << endl;
return 0;
}

```

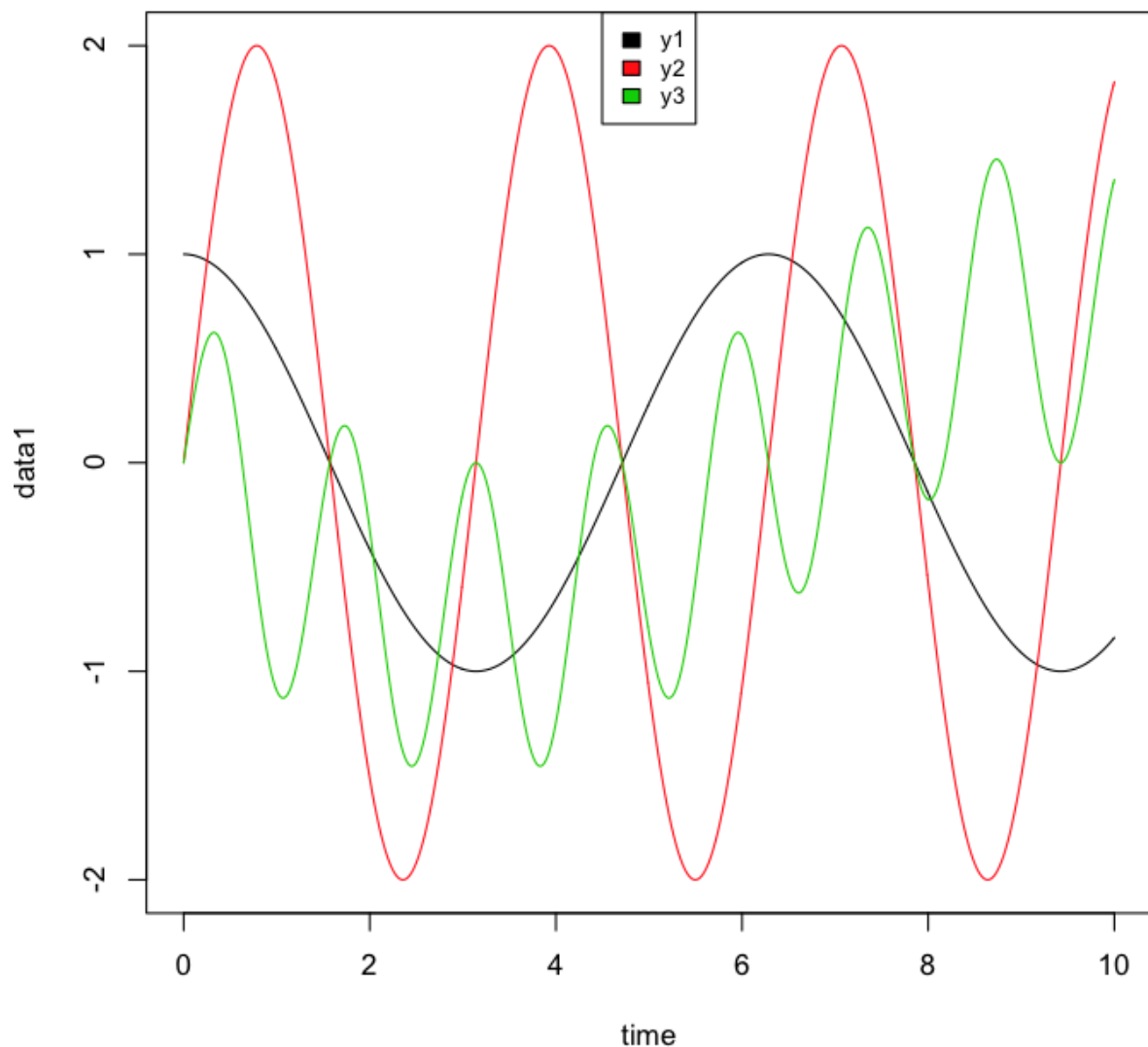
Remarque : nous avons du aussi inclure la bibliothèque **math.h** pour pouvoir calculer les fonctions sinus et cosinus.

```

#plot_32.r
data1=read.table("test_32.txt",header=TRUE)
time=data1$time
y1=data1$y1
y2=data1$y2
y3=data1$y3
data1=data1[,-1]
matplot(time,data1,type="l",lty=1)
legend("top", colnames(data1),col=seq_len(3),cex=0.8,fill=seq_len(3))1)

```

Les commandes R permettent de tracer les courbes de la figure suivante.

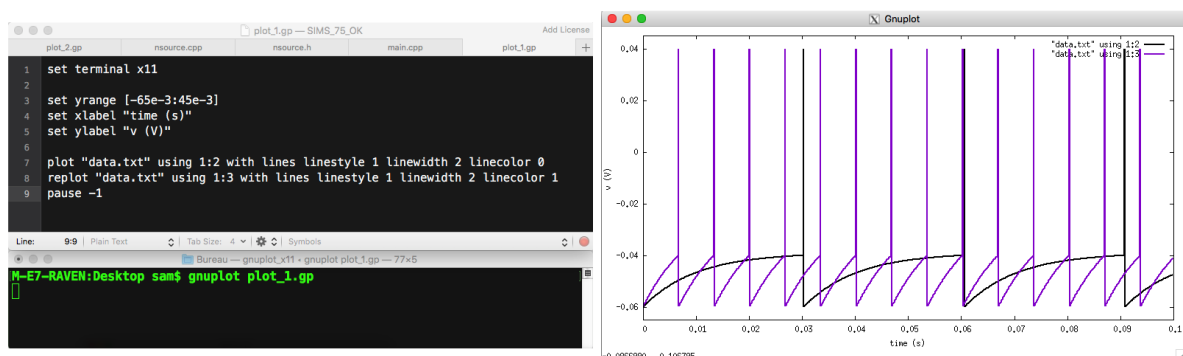


D'autres commandes seront ponctuellement évoquées pour certains tracés spécifiques mais en comprenant bien le fonctionnement de **plot()** et **matplot()** l'essentiel des courbes pourront être produites.

2.2 Le logiciel gnuplot

Installer et ouvrir une session gnuplot

Le logiciel [gnuplot](https://www.gnuplot.info/) s'installe sur la plupart des plateformes (Windows, linux, OSX ...) et est gratuit. Une fois le logiciel installé nous pouvons travailler comme pour R soit avec l'aide de scripts soit directement dans la console :



Gnuplot sera considéré comme une alternative à R selon sa disponibilité.

3 Simulation du vivant in silico

Après avoir vu comment utiliser le langage C++ pour réaliser des calculs et R pour produire des graphes, nous allons voir comment mettre ces programmes en application pour modéliser et analyser les mécanismes du vivant.

3.1 Généralités

3.1.1 Définitions

Nous présentons ici quelques définitions de base permettant de mieux saisir la suite du cours :

Modèle : représentation abstraite, partielle, ou symbolique d'un élément étudié que l'on va utiliser comme outil de prédiction ou d'investigation pour la compréhension des phénomènes associés à cet élément.

Modèle mathématique : modèle abstrait utilisant le formalisme mathématique

Modèle informatique : logiciel destiné à simuler une abstraction d'un système particulier

Modèle biologique : espèce vivante que l'on peut facilement élever ou cultiver, afin de faire des expériences scientifiques, qu'il est plus difficile de mettre en œuvre avec des espèces plus complexes.

Simulation : outil utilisé par le chercheur, l'ingénieur, le militaire etc. pour étudier les résultats d'une action sur un élément sans réaliser l'expérience sur l'élément réel.

3.1.2 Pourquoi réaliser des modèles computationnels ?

D'une façon générale la réponse à cette question tient en deux mots : **comprendre** et **prévoir**.

- **Comprendre**

Les biologistes n'ont bien sûr pas attendu de disposer de grandes puissances de calcul pour comprendre les mécanismes du vivant mais les progrès de la connaissance fondamentale demeurent liés à l'évolution de ses outils d'exploration. Difficile par exemple d'envisager la [théorie cellulaire](#) si l'on ne dispose pas de [microscope](#) et que l'on ignore donc l'existence de la cellule. Or au même titre que le microscope, l'informatique est, entre autres, un outil permettant de traiter et d'explorer de l'information de façon non accessible à un cerveau humain ordinaire. De même que ce n'est pas le microscope qui découvre quelque chose, ce n'est pas l'ordinateur qui explique un phénomène mais il est à même de traiter l'information de manière à rendre ce phénomène intelligible et interprétable par un être humain. Nous allons illustrer ceci avec un exemple. Supposons que nous ayons des informations sur la dynamique globale de types cellulaires embryonnaires ainsi que sur leurs interactions avec des signaux chimiques identifiés et que nous voulions comprendre à l'aide de ces éléments [comment lors de l'organogenèse, le pancréas se forme chez la souris](#). En pratique cela revient à mettre en évidence une propriété émergente du système formé par le tissu embryonnaire (ici un simple amas de cellules à t_0) qui est à l'origine de la formation d'un pancréas à un stade ultérieur de développement. Cela permet donc **in fine** de mieux comprendre l'organogenèse pancréatique.

- **Prévoir**

L'application du calcul intensif à la prédiction des phénomènes naturels est relativement ancienne. Avant même que les ordinateurs s'installent chez les ingénieurs et dans les laboratoires, les champions du boulier et de la règle à calcul pouvait, avec beaucoup de temps, de patience et de rigueur, fournir des données quantitatives. Cela a été très tôt appliqué à la

physique, à la chimie, à l'astronomie et ultérieurement à la biologie. En neurophysiologie par exemple le modèle de Hodgkin-Huxley a été numériquement développé à l'aide d'une simple calculatrice mécanique car les ordinateurs n'étaient pas disponibles dans les années 50 dans les laboratoires.



Il n'est pas certain que le progrès scientifique soit directement corrélé au progrès technologique car la disponibilité d'une puissance de calcul phénoménal peut parfois avoir l'effet pervers de détourner des questions de fond au profit d'un culte de l'outil mais si celui-ci est utilisé à bon escient il permet de prévoir **in silico** des résultats qu'il ne serait pas envisageable d'obtenir autrement. Il existe dans le milieu de la recherche et du développement un très grand nombre d'applications de la simulation informatique à la prédiction depuis les modèles météorologiques ou géologiques jusqu'aux modèles de résistance des matériaux ou d'écoulements fluides. Comme nous le verrons cet outil peut être également avoir des applications pratiques pour la prédiction de processus biologiques. Un exemple sur lequel nous reviendrons concerne par exemple la prédiction de la qualité d'un vin (en terme d'équilibre de ses principaux composants) à partir de sa composition de départ et des conditions physico-chimiques de fermentation.

3.1.3 A quelle échelle travailler ?

L'objectif principal de la biologie est de comprendre le vivant ce qui implique de l'explorer à plusieurs échelles en fonction des données qui sont accessibles au modélisateur :

- échelle moléculaire
- échelle cellulaire
- échelle tissulaire
- échelle de l'organe
- échelle de l'organisme
- échelle de la population ...

L'échelle de description du modèle dépend également en partie des facteurs étudiés sur le système de référence ¹² ainsi que du temps de développement et de la puissance de calcul disponible. Enfin l'échelle de simulation est en général inférieure ou égale à l'échelle de prédiction. Cette idée relativement intuitive fait qu'on ne peut pas expliquer directement un mécanisme à l'échelle moléculaire quand on travaille à l'échelle cellulaire par exemple.

3.2 Formaliser un problème de biologie

Lorsque l'on veut écrire un modèle à des fins de simulation, l'une des premières étapes consiste à formaliser le système étudié. Il n'existe pas une méthode unifiée pour aborder la modélisation informatique en biologie. Celle-ci dépend de plusieurs facteurs parmi lesquels l'échelle à laquelle le système est décrit, la nature de ce système, le type de réponse attendue etc. Nous allons présenter ici deux formalismes classiques (et non mutuellement exclusifs) permettant de décrire la dynamique d'un système de référence :

- la mise en place d'un système d'équations différentielles que nous résoudrons numériquement
- la description du système sous la forme d'agents en interaction dans un environnement

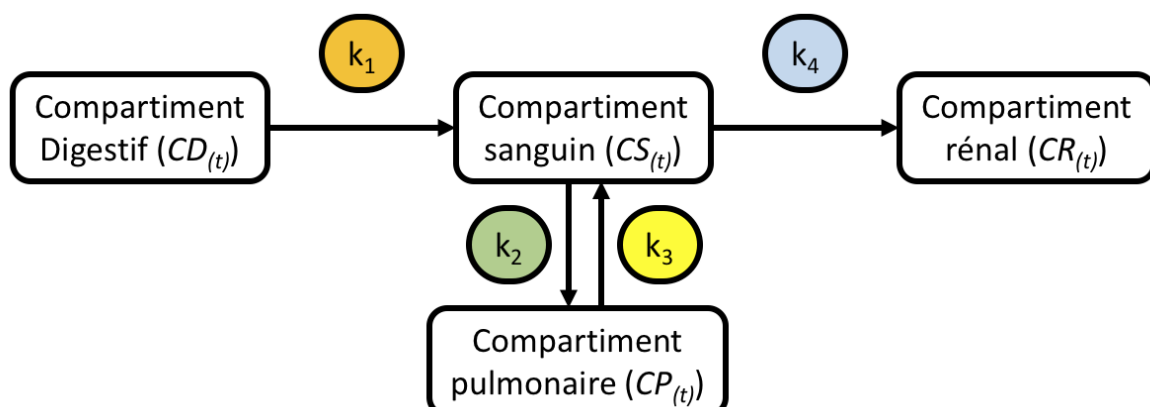
Ces deux approches aboutiront à la production de données numériques résultant de nos simulations, données que nous pourrons utiliser tout d'abord pour explorer et valider le modèle et ensuite pour l'exploiter.

3.2.1 Les équations différentielles

Cet enseignement n'a pas vocation à se substituer à un cours de mathématiques et l'abord analytique des équations différentielles ne sera pas détaillé ici. Nous verrons simplement comment les utiliser pour simuler la dynamique d'un système à travers leur résolution numérique et nous allons illustrer cette approche à travers un exemple concret : un problème de pharmacocinétique. Pour avoir une idée du type de connaissance que cette approche permet de développer le lecteur pourra se reporter à [cet article par exemple](#). Mais nous allons voir ici un modèle moins complexe. Supposons que nous désirions étudier l'évolution de la concentration pulmonaire d'un antibiotique (molécule **DA-42**) pris par voie orale au cours du temps. Nous savons que ce médicament est entièrement assimilé au niveau intestinal et va donc se retrouver dans la circulation sanguine. A partir de là il y a deux possibilités :

- une partie s'échange du sang vers le poumon
- une partie est excrétée par les reins

Notre système de référence est donc ici un organisme humain et si nous voulons représenter le diagramme de ce modèle nous pouvons utiliser le schéma suivant :



Sur cette figure les 4 compartiments sont représentés avec leurs interconnexions. La diffusion du **DA-42** se fait suivant les différentes flèches entre les compartiments et avec des constantes k_i différentes. Les concentrations entre les différents compartiments sont donnés par $CD(t)$, $CS(t)$, $CP(t)$ et $CR(t)$. **DA-42** diffuse au cours du temps du compartiment **A** au compartiment **B** en fonction de sa concentration dans **A** et de la constante de diffusion $k_{A \rightarrow B}$ ce qui nous permet d'écrire le système d'équations différentielles ordinaires (**ODE**) suivant :

$$\frac{dCD(t)}{dt} = -k_1 \times CD(t)$$

$$\frac{dCS(t)}{dt} = k_1 \times CD(t) + k_3 \times CP(t) - k_2 \times CS(t) - k_4 \times CS(t)$$

$$\frac{dCP(t)}{dt} = k_2 \times CS(t) - k_3 \times CP(t)$$

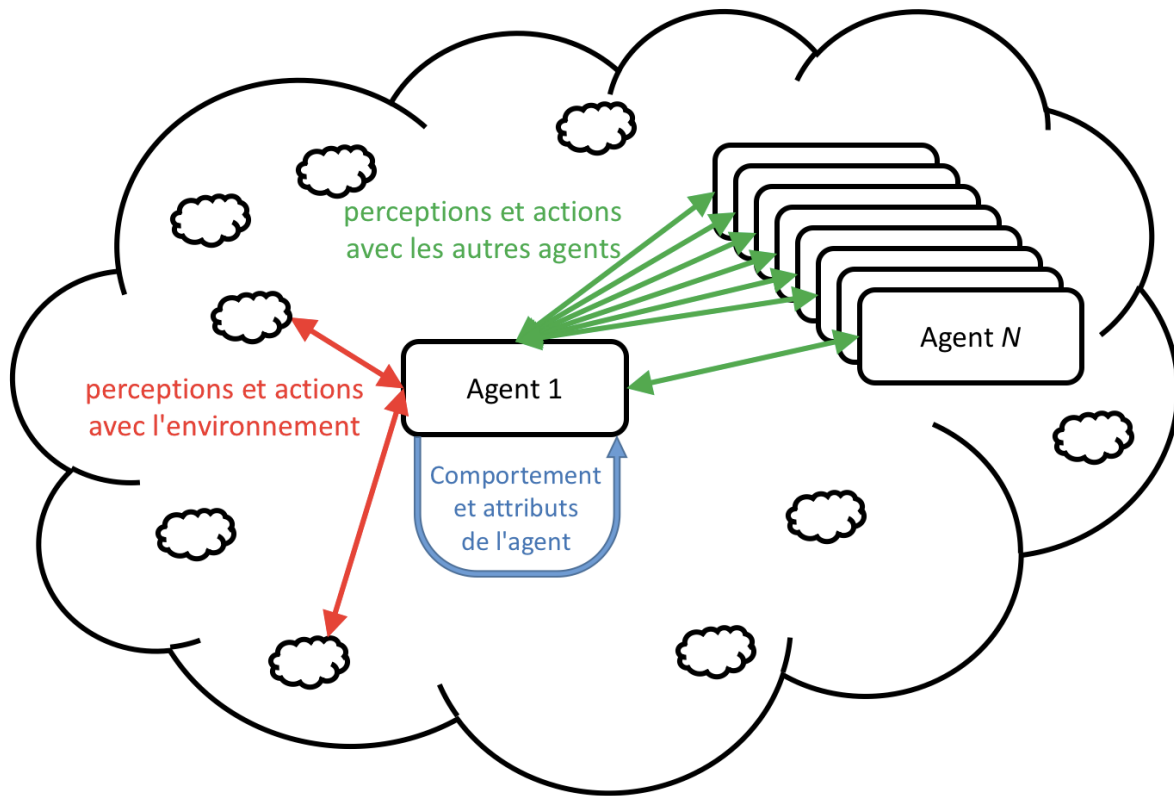
$$\frac{dCR(t)}{dt} = k_4 \times CS(t)$$

Comme on le note ici, la somme de toutes les variations dans tous les compartiments est nulle, ce qui est logique dans la mesure où le système décrit ci-dessus est fermé et en l'absence d'entrée supplémentaire ou de destruction (par exemple par voie enzymatique) la quantité totale de **DA-42** ne varie pas au cours du temps. Toutefois la partie passant dans le compartiment rénal allant dans l'urine nous pouvons l'assimiler à une sortie du système. Nous verrons dans la paragraphe [3.3](#) comment résoudre numériquement ce système équationnel mais à ce stade nous l'avons déjà formalisé mathématiquement. Il faut bien entendu avoir des données expérimentales à proposer qui permettront de fixer ou de déduire par ajustement les valeurs des paramètres k_1 , k_2 , k_3 et k_4 avec les unités adéquates pour que le modèle formel soit complet.

3.2.2 Les systèmes multi-agents

Une autre méthode intéressante pour l'étude des propriétés émergentes des systèmes est l'utilisation de systèmes multi-agents (MAS). Ceux-ci reposent sur le paradigme suivant : le système de référence est décrit (dynamiquement ou non) par un ensemble d'agents en interaction dans un environnement. Nous avons déjà évoqué précédemment mais sans le détailler [un exemple de MAS sur l'organogenèse pancréatique](#). En éthologie, ils sont également utiles pour étudier le fonctionnement et [les comportements émergents de populations d'insectes sociaux](#). Les MAS sont aussi utilisés pour comprendre [l'intelligence distribuée sur des colonies d'agents autonomes](#) par exemple.

Le principe des systèmes multi-agent est le suivant : le programme simule au cours du temps (en général) l'évolution d'un ensemble d'agents logiciels évoluant dans un environnement. Il y a des interactions entre les agents et avec l'environnement qui dépendent de règles locales (même s'il est possible d'y adjoindre des règles à une échelle supérieure également). La figure suivante résume cette approche.



La programmation objet n'est pas développée dans cet enseignement mais il faut noter qu'elle se prête particulièrement bien au développement de systèmes multi-agents car il est très simple de décrire les agents à l'aide de classes et d'objets ¹³.

3.3 Méthodes de calcul numérique appliquées aux modèles dynamiques

Nous allons aborder ici deux méthodes simples pour résoudre numériquement les systèmes d'équations différentielles ordinaires.

3.3.1 Principe général

Faute de pouvoir résoudre analytiquement un système quand celui-ci est trop complexe (ce qui est généralement le cas), les méthodes de résolution numérique donnent une excellente estimation (avec une fiabilité mesurable) de la résolution des ODE. Les ODE sont des équations différentielles qui comprennent une ou plusieurs fonctions d'une variable indépendante et de ses dérivées. L'ordre de la dérivée le plus élevé donne l'ordre de l'équation différentielle. Elles se présentent de la façon suivante :

$$\frac{dy}{dt} = f(t, y(t))$$

Avec la condition $y_{(t=0)} = y_0$ à l'instant initial.

3.3.2 Résolution analytique

Nous allons illustrer le principe de la résolution analytique à l'aide d'un exemple très simple qui est celui de la décroissance exponentielle. Cette loi mathématique est [très fréquemment retrouvée dans les sciences expérimentales](#). Nous allons l'appliquer à la décroissance du médicament **AG-50** dans le compartiment sanguin. Nous allons poser quelques hypothèses pour simplifier le problème :

- à $t = 0$ l'intégralité de **AG-50** est présente dans le compartiment sanguin (pas d'effet d'absorption ou de diffusion)

- Le médicament **AG-50** n'a qu'une seule voie d'élimination : il est catabolisé par une enzyme présente à saturation dans le sang à proportion d'une constante k_{elim} .

Si l'on considère que $C_{(t)}$ est la concentration sanguine de **AG-50** à t et que à $t = 0$, $C_{(t=0)} = C_0$ est la concentration maximale, le modèle peut s'écrire de la façon suivante :

$$C_{(t)} = C_0 \times e^{-k_{elim} \times t}$$

Pour comprendre d'où vient cette relation il faut revenir sur la réaction de dégradation de **AG-50** :

$$\frac{dC_{(t)}}{dt} = -k_{elim} \times C_{(t)}$$

Ce qui revient à dire que la variation instantanée de **AG-50** est proportionnelle à cette constante k_{elim} et à la quantité circulante $C_{(t)}$. Si nous intégrons la relation précédente nous obtenons :

$$\frac{dC_{(t)}}{dt} = -k_{elim} \times C_{(t)}$$

$$\frac{dC_{(t)}}{C_{(t)}} = -k_{elim} \times dt$$

$$\int_0^t \frac{dC_{(t)}}{C_{(t)}} = -k_{elim} \int_0^t dt$$

$$\ln\left(\frac{C_{(t)}}{C_0}\right) = -k_{elim} \times t$$

$$C_{(t)} = C_0 \times e^{-k_{elim} \times t}$$

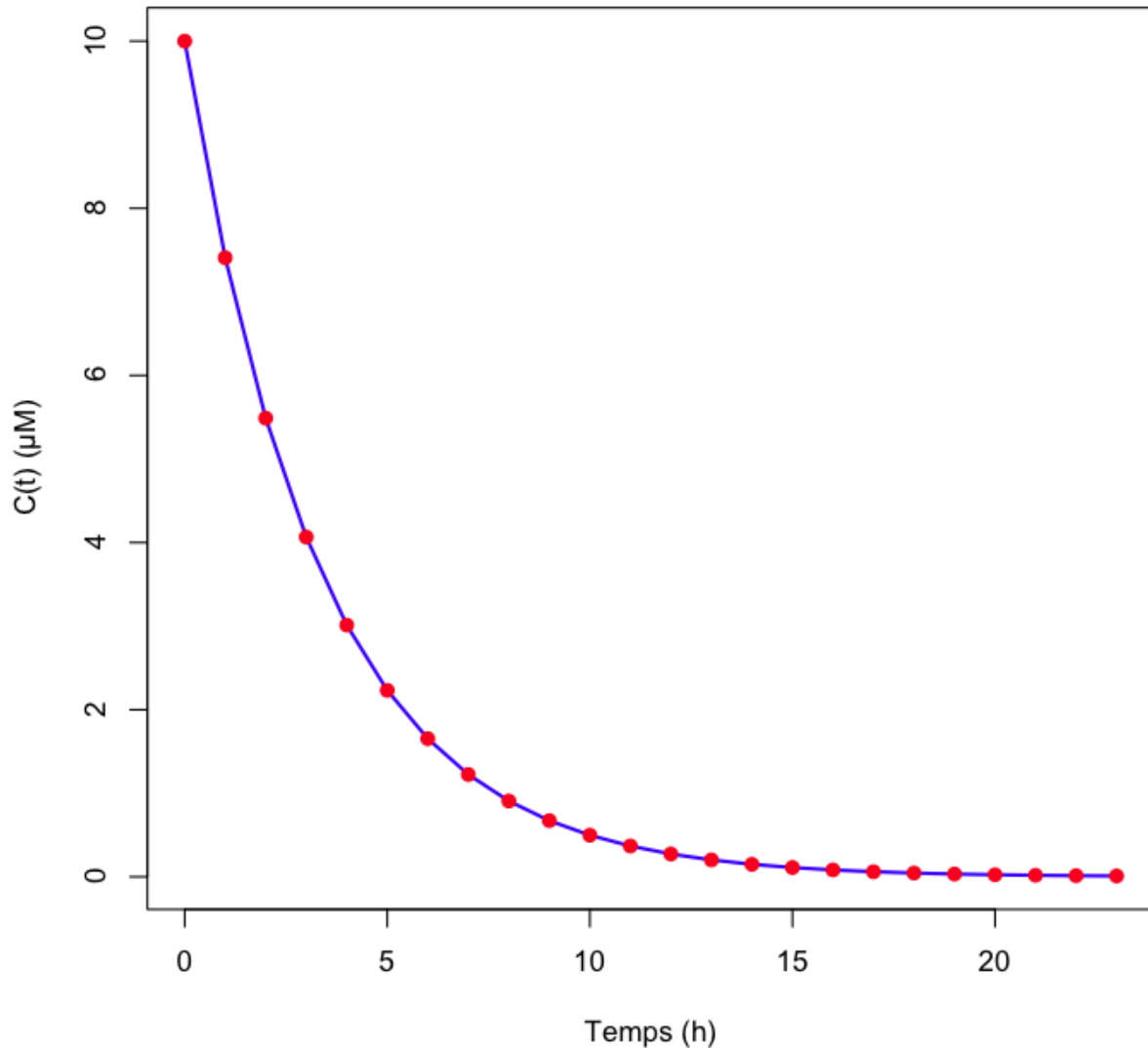
Nous allons maintenant tracer le résultat obtenu ici. Pour l'application numérique nous avons $C_0 = 10.0 \mu M$ et $k_{elim} = 0.3 \mu M \times h^{-1}$:

```
// code_33.cpp
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <math.h>
using namespace std;
int main(){
    double C0=10.0;
    double k=0.3;
    int nPoints=24;
    vector<double> time,Ct;
    for (int i=0;i<nPoints;i++){
        time.push_back((double)i);
        Ct.push_back(C0*exp(-k*(double)i));
    }
    // création de la variable data de type string pour écriture
    string data="time Ct\n";
    for (int i=0;i<nPoints;i++){
        data+=to_string(time[i])+" "+to_string(Ct[i])+"\n";
    }
    // ouverture en écriture du fichier
    ofstream fichier("test_33.txt"); // données sauvegardées dans test_33.txt
    if(fichier){
        fichier << data;
        fichier.close();
    } else cerr << "Impossible d'ouvrir le fichier !" << endl;
    return 0;
}
```

Nous utilisons le script **plot_33.r** :


```
#plot_33.r
data1=read.table("test_33.txt",header=TRUE)
plot(data1$time,data1$Ct,xlab="Temps (h)", ylab="C(t) (microm)"
      ,type="l",lwd=2,lty=1,col="blue")
points(data1$time,data1$Ct,col="red",pch=19)
```

Ce qui nous permet de tracer la figure suivante :



Comme nous le voyons il est très simple dans ce cas précis de résoudre analytiquement le système et d'en calculer l'évolution à partir de la solution exacte mais cette démarche est présentée ici à titre d'information et nous ne l'emploierons plus par la suite car elle n'est que rarement applicable en pratique. Comment en ce cas résoudre ce problème sans passer par l'analyse ?

3.3.3 Méthode d'Euler explicite (Forward Euler Method)

Pour répondre à cette question nous allons discrétiser l'expression initiale de la relation qui comporte l'équation différentielle en suivant la méthode d'Euler. La relation :

$$C'_{(t)} = \frac{dC_{(t)}}{dt} = -k_{elim} \times C_{(t)}$$

peut ainsi s'écrire :

$$\frac{C_{(t+\Delta t)} - C_{(t)}}{\Delta t} = -k_{elim} \times C_{(t)}$$

Il s'agit d'une approximation où les variations infinitésimales (dt) sont remplacées par des variations discrètes (Δt). La relation peut ensuite être développée de la façon suivante :

$$C_{(t+\Delta t)} = C_{(t)} + \Delta t \times (-k_{elim} \times C_{(t)})$$

$$C_{(t+\Delta t)} = C_{(t)} + \Delta t \times C'_{(t)}$$

Avec $C_{(t=0)} = C_0$

nous connaissons à la fois l'état initial du système ainsi qu'une méthode permettant de calculer une estimation de l'état du système à l'état t à l'état $t+\Delta t$ par un simple calcul de pente. Le corollaire en est bien sûr que plus Δt tend vers dt , plus l'intégration tend vers la solution exacte. Nous allons appliquer le modèle précédent à cette méthode de résolution dans le code **code_34.cpp** :

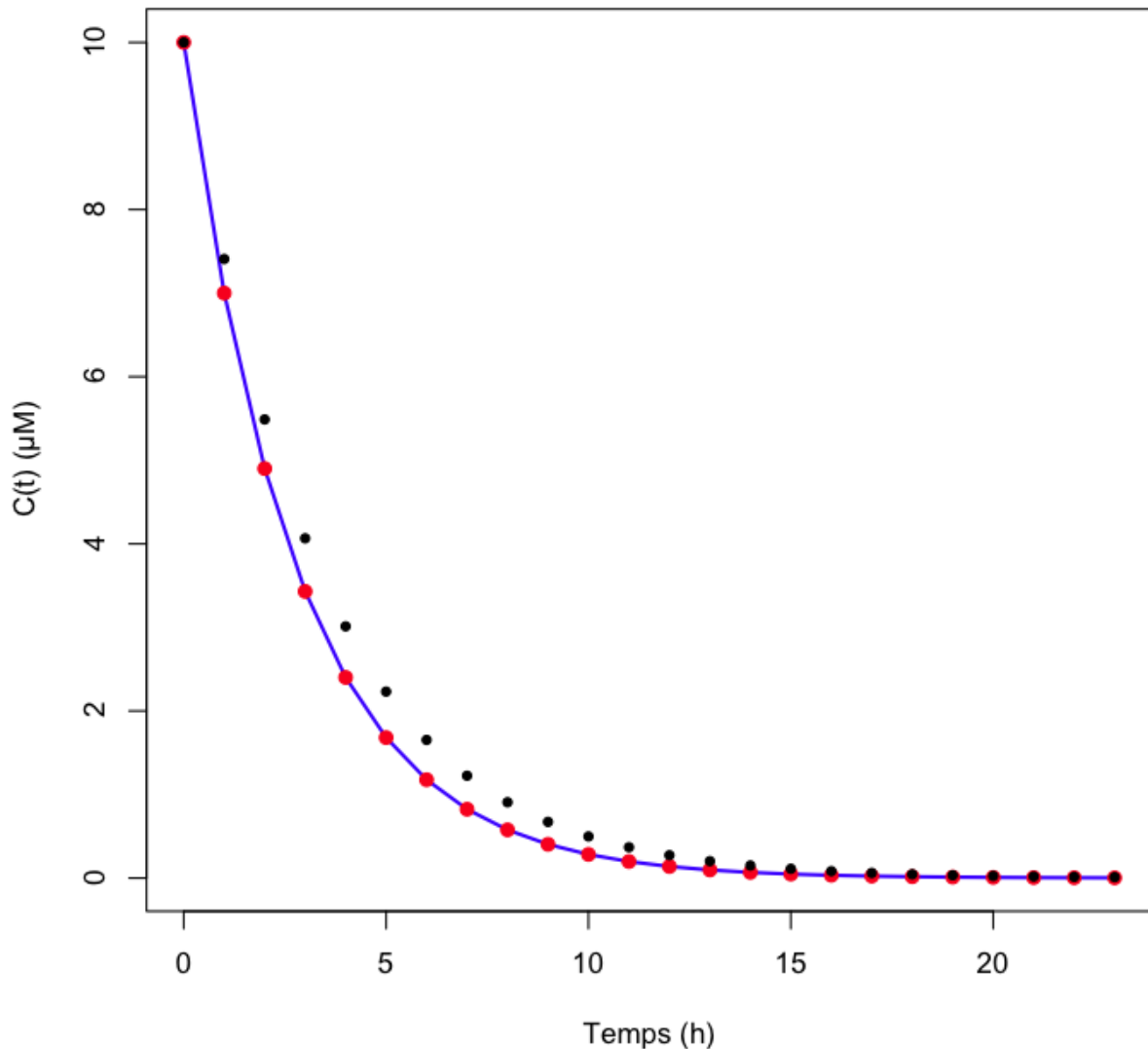
```
// code_34.cpp
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <math.h>
using namespace std; // utilisation de l'espace de nom de std

int main(){
    double C0=10.0;double k=0.3;int nPoints=24;int dt=1;
    double C=C0;
    vector<double> time,Ct,Cte;
    // calcul des valeurs "exactes" dans Cte
    for (int i=0;i<nPoints;i++){
        time.push_back((double)i);
        Cte.push_back(C0*exp(-k*(double)i));
    }
    // calcul estimé avec la méthode d'Euler
    for (int i=0;i<nPoints;i++){
        Ct.push_back(C);
        C=C+dt*(-k*C);
    }
    // création de la variable data de type string pour écriture
    string data="time Ct Cte\n";
    for (int i=0;i<nPoints;i++){
        data+=to_string(time[i])+" "+to_string(Cte[i])+"
"+to_string(Ct[i])+"\n";
    }
    // ouverture en écriture du fichier
    ofstream fichier("test_34.txt");
    if(fichier){
        fichier << data;
        fichier.close();
    } else cerr << "Impossible d'ouvrir le fichier !" << endl;
    return 0;
}
```

Le programme **code_34.cpp** implémente la méthode d'Euler explicite et sauvegarde les données dans **test_34.txt** :

```
#plot_34.r
data1=read.table("test_34.txt",header=TRUE)
plot(data1$time,data1$Ct,xlab="Temps (h)", ylab="C(t) (microm)",
      type="l",lwd=2,lty=1,col="blue")
points(data1$time,data1$Ct,col="red",pch=19)
points(data1$time,data1$Cte,pch=20,lty=1,col="black")
```

Le script **plot_34.r** permet de tracer la figure suivante :



Cette figure montre la convergence correcte des deux méthodes de calcul mais il est visible sur le tracé qu'il y a une différence, faible mais perceptible, entre les deux courbes. De fait la méthode d'Euler explicite est extrêmement simple à mettre en œuvre mais souffre d'une précision moyenne et est très fortement influencée par la taille de Δt . Elle est néanmoins très fréquemment utilisée car elle est particulièrement rapide et s'adapte à la plupart des cas. Si une précision particulièrement fine est requise, nous pouvons appliquer d'autres méthodes comme par exemple la méthode de Runge-Kutta à l'ordre 4 qui donne de très bons résultats au détriment du temps de calcul qui est sensiblement plus long.

3.3.4 Des données au modèle : validation et exploitation d'un modèle computationnel

L'objectif final de la simulation en biologie est de pouvoir reproduire des résultats expérimentaux **in silico** de façon à comprendre et à prévoir ces phénomènes.

La démarche habituelle consiste donc comme dans toute démarche en sciences expérimentales, à observer le système de référence. Il faut ensuite poser une hypothèse concernant les mécanismes qui produisent ce phénomène, qui l'expliquent de manière causale. Une fois cette hypothèse posée il faut imaginer des expériences pour la rejeter ou la conserver. Si l'hypothèse est rejetée parce qu'elle n'explique pas les résultats des expériences il faut la modifier ou en bâtir une nouvelle et recommencer.

En modélisation la démarche est la même. Nous partons d'observations et de mesures expérimentales, puis nous posons une hypothèse. De cette hypothèse nous dégagons un formalisme (équations, agents ...) et de ce formalisme un logiciel de simulation. Les paramètres du logiciel de simulation vont ensuite être ajustés pour que le modèle soit cohérent dans ses résultats avec l'observation expérimentale. A cette fin il existe plusieurs stratégies qui font appel dans les cas simples à des ajustements manuels et dans les cas plus complexes à des méthodes d'optimisation. Quand ceci est réalisé notre modèle est validé, ce qui ne signifie pas qu'il restera toujours inchangé. Il peut être ultérieurement raffiné ou étendu. L'application qui suit illustre ce propos.

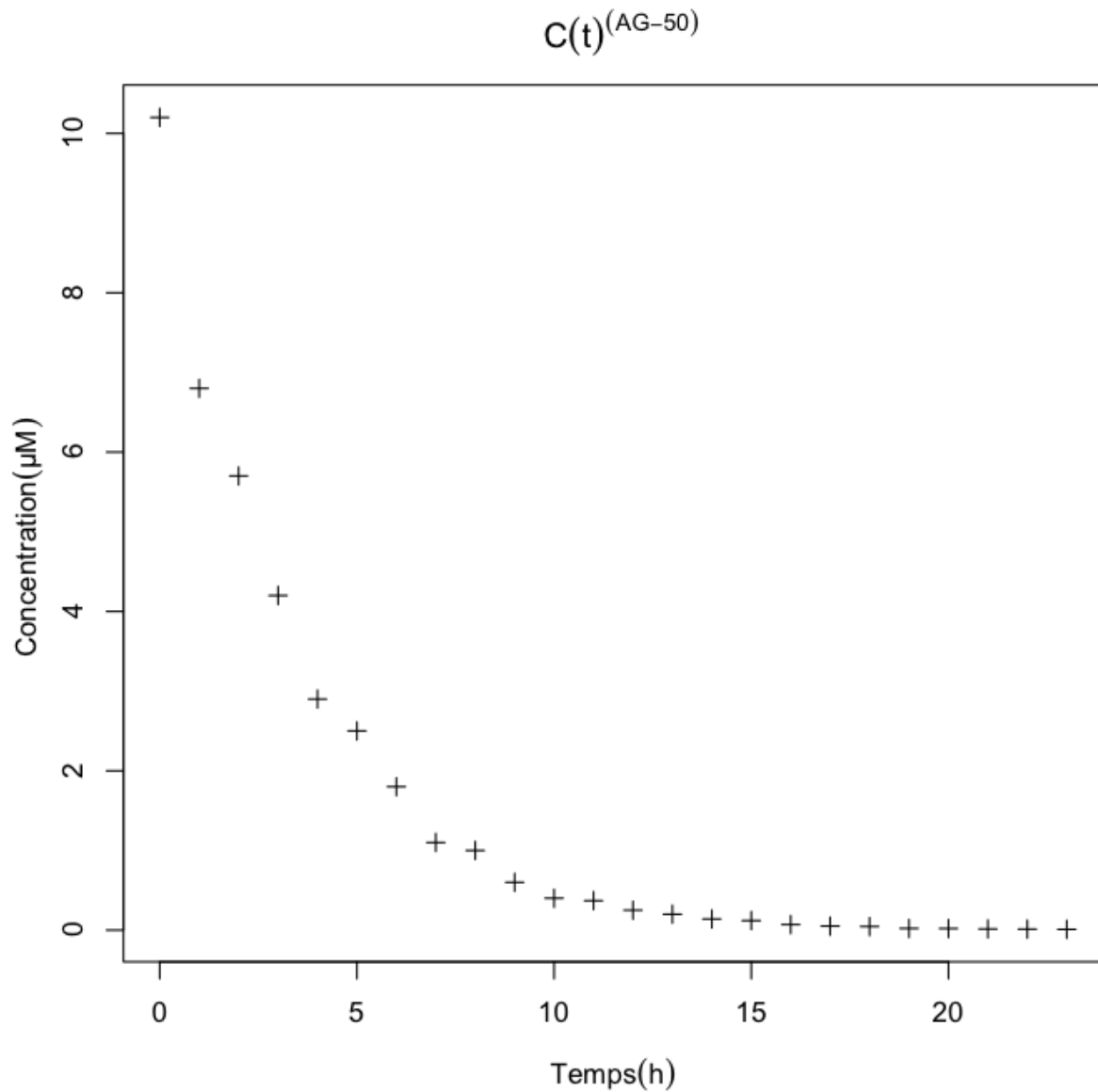
Nous allons reprendre notre modèle précédent sur la cinétique de décroissance de la molécule **AG-50** dans le compartiment sanguin injectée à $t=0$. Supposons que nous ne connaissions pas la constante k_{elim} , pour la déterminer nous allons effectuer des mesures expérimentales. Nous recueillons les mesures sur 20 patients volontaires pour lesquels nous obtenons donc 24 valeurs moyennes pour la concentration $C_{(t)}^{AG-50}$ au cours du temps. Les valeurs sont dans le tableau suivant :

Temps(h)	1	2	3	4	5	6	7	8	9	10	11	12
$C_{(t)}^{AG-50} en \mu M$	10.2	6.8	5.4	4.2	2.9	2.5	1.8	1.1	1	0.6	0.4	0.37
Temps(h)	13	14	15	16	17	18	19	20	21	22	23	24
$C_{(t)}^{AG-50} en \mu M$	0.25	0.2	0.14	0.12	0.07	0.05	0.04	0.03	0.02	0.015	0.013	0.01

Nous pouvons visualiser ces points expérimentaux à l'aide du code suivant :

```
> data=c(10.2,6.8,5.7,4.2,2.9,2.5,1.8,1.1,1,0.6,0.4,0.37,0.25,
0.2,0.14,0.12,0.07,0.05,0.045,0.021,0.02,0.015,0.013,0.01)
> time=0:23
> plot(time,data,xlab = expression(Temps (h)), ylab =
expression(Concentration (microm)),pch=3,main=expression(C(t)^(AG-50)))
```

nous obtenons la figure suivante :



Pour illustrer notre propos de façon simple nous allons chercher à minimiser la différence entre ce que le modèle donne comme valeurs et les valeurs expérimentales. Nous allons pour ça mesurer l'erreur quadratique (somme du carré des différences) et chercher pour différentes valeurs de k_{elim} dans l'intervalle $[0, 0.5]$ laquelle minimise cette erreur à l'aide du programme **code_35.cpp**. Par souci de clarté nous avons divisé ce script en plusieurs parties afin d'expliquer son principe de fonctionnement :

```
// code_35.cpp
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <math.h>
using namespace std; // utilisation de l'espace de nom de std
int main(){
    // valeurs expérimentales

    vector<double>data=10.2,6.8,5.7,4.2,2.9,2.5,1.8,1.1,1,0.6,0.4,0.37,0.25,0.2,0.1
4,
    0.12,0.07,0.05,0.045,0.021,0.02,0.015,0.013,0.01};
    int nPoints=24;
    vector<int> time;
    for (int i=0;i<nPoints;i++){
```

```

        time.push_back((double)i);
    }
    int dt=1;
    double c0=10.0;

```

Le code importe les bibliothèques nécessaires, les données expérimentales sont stockées dans la variable `data` et les paramètres et variables de fonctionnement sont créés et initialisés.

```

// code_35.cpp suite
# variables d'ajustement
double lowError=1e300;
double bestK=0;

```

lowError contient la valeur de l'erreur par défaut. Celle-ci est choisie très grande (10^{300}) pour que dès la première recherche le résultat soit mis à jour. En effet si nous initialisons cette valeur à 0 il y a peu de chance qu'elle soit améliorable. **bestK** contiendra la meilleure valeur de k_{elim} qui sera trouvée. Elle sera donc mise à jour quand l'erreur trouvée sera inférieure à **lowError**.

```

// code_35.cpp suite
// recherche de la valeur de k minimisant l'erreur
for (double k=0;k<=0.5;k+=0.01){
    double C=C0;double error=0;
    for (int t=0;t<nPoints;t++){
        //Ct.push_back(C);
        error+=pow((data[t]-C),2);
        C=C+dt*(-k*C);
    }
    if (error<lowError){
        lowError=error;
        bestK=k;
    }
}

```

La boucle 1 parcourt avec k les valeurs de 0 à 0.5 de 0.01 en 0.01. Une sous-boucle (boucle 2) utilise ensuite cette valeur de k pour calculer les valeurs de la courbe à l'aide de la méthode d'Euler. A l'issue de ce calcul, l'erreur quadratique entre les valeurs expérimentales et les valeurs simulées est calculée (**error**). Cette erreur est ensuite comparée à la meilleure valeur courante (**lowError**). Si elle est inférieure alors la nouvelle meilleure valeur courante sera **error** et donc la meilleure valeur courante de k_{elim} (**bestK**) sera k .

```

// code_35.cpp suite et fin
// boucle du tracé avec la meilleure valeur trouvée pour k
double C=C0;vector<double> Ct;
for (int i=0;i<nPoints;i++){
    Ct.push_back(C);
    C=C+dt*(-bestK*C);
}
// création de la variable data de type string pour écriture
string dataTS="time Ct\n";
for (int i=0;i<nPoints;i++){
    dataTS+=to_string(time[i])+" "+to_string(Ct[i])+"\n";
}
// ouverture en écriture du fichier
ofstream fichier("test_35.txt");
if(fichier){

```

```

    fichier <- dataTS;
    fichier.close();
  } else cerr <- "Impossible d'ouvrir le fichier !" <- endl;
  return 0;
}

```

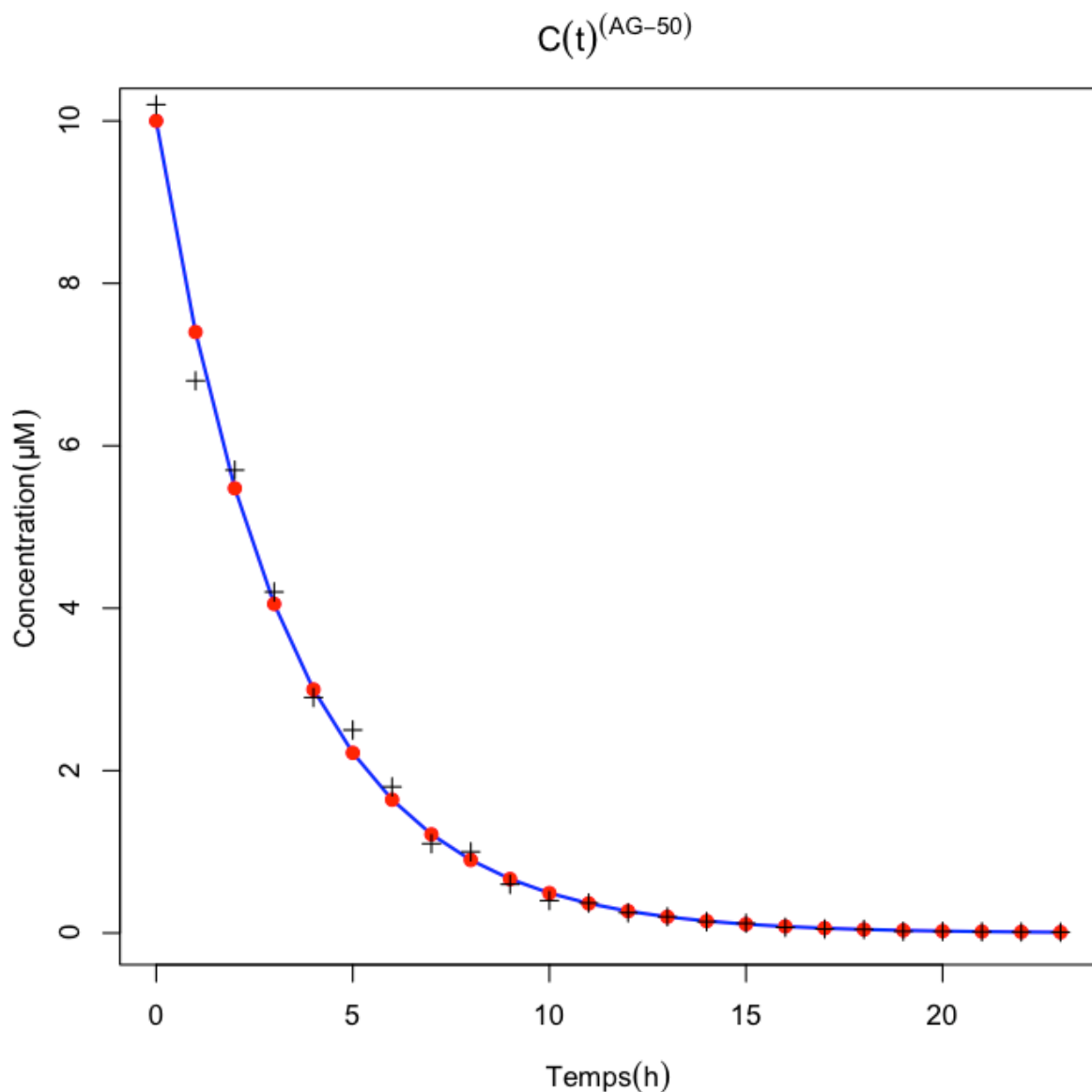
Une fois l'intervalle $[0, 0.5[$ parcouru, nous calculons la courbe ayant présenté l'erreur la plus faible avec les données expérimentales (donc celle qui utilise la valeur **bestK**) toujours en utilisant la méthode d'Euler. Ces données sont ensuite sauvegardées dans le fichier **test_35.txt**.

```

#plot_35.r
data=c(10.2,6.8,5.7,4.2,2.9,2.5,1.8,1.1,1,0.6,0.4,0.37,0.25,0.2,
       0.14,0.12,0.07,0.05,0.045,0.021,0.02,0.015,0.013,0.01)
data1=read.table("test_35.txt",header=TRUE)
plot(data1$time,data1$Ct,xlab = expression(Temps (h)), ylab =
     "Concentration (microM)",type="l",lwd=2,lty=1,col="blue",
     main="C(t)^(AG-50)")
points(data1$time,data1$Ct,col="red",pch=19)
points(data1$time,data,pch=3,lty=1,col="black")

```

Nous traçons avec le script R ci-dessus les courbes annotées sur la figure suivante :



La meilleure valeur trouvée ($k=0.26$) est proche de la valeur de $k_{\text{elim}}=0.3$ et le **fit** (ajustement) est correct malgré un pas Δt très large. Nous avons ainsi accompli une première étape vers la validation du modèle.

Utilisation du modèle

Maintenant que les paramètres du modèle ont été ajustés nous pouvons l'utiliser pour évaluer une prédiction. Par exemple supposons que nous voulions avoir une estimation du taux sanguin de **AG-50** au bout de 5h après injection d'une quantité de drogue qui amène $CA_{(t=0)}^{AG-50} = 20\mu M$. Le programme **code_36.cpp** va répondre à cette question :

```
// code_36.cpp
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
#include <math.h>
using namespace std; // utilisation de l'espace de nom de std

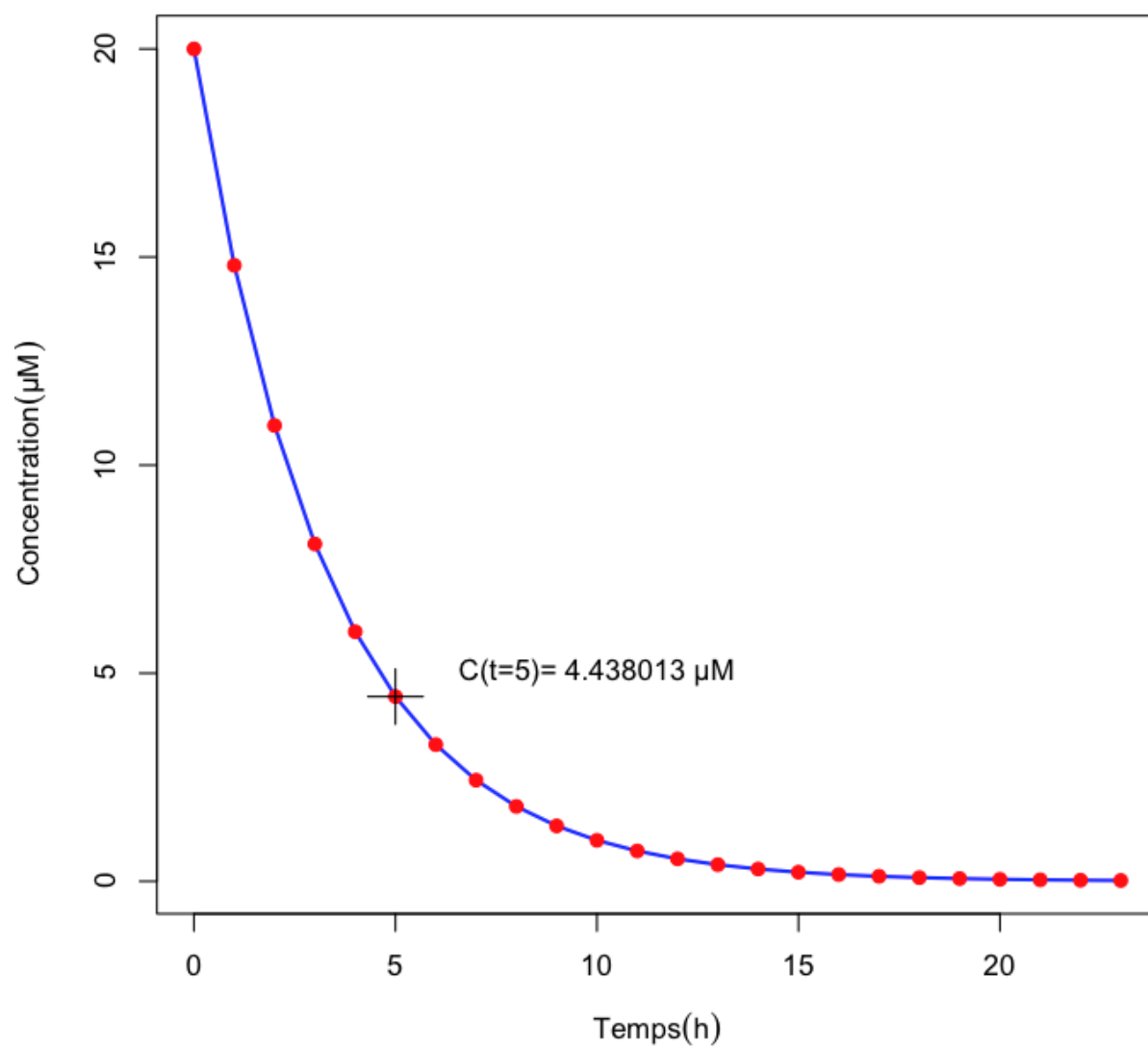
int main(){
    // boucle du tracé avec la meilleure valeur trouvée pour k (0.26) et c0=20
    int nPoints=24;vector<int> time;
    for (int t=0;t<nPoints;t++) time.push_back((double)t);
    int dt=1;double C0=20.0;double k=0.26;double Ct5;
    // boucle du tracé avec la meilleure valeur trouvée pour k
    double C=C0;vector<double> Ct;
    for (int t=0;t<nPoints;t++){
        Ct.push_back(C);
        C=C+dt*(-k*C);
    }
    // création de la variable data de type string pour écriture
    string dataTS="time Ct\n";
    for (int t=0;t<nPoints;t++){
        dataTS+=to_string(time[t])+" "+to_string(Ct[t])+"\n";
    }
    // ouverture en écriture du fichier
    ofstream fichier("test_36.txt");
    if(fichier){
        fichier << dataTS;
        fichier.close();
    } else cerr << "Impossible d'ouvrir le fichier !" << endl;
    return 0;
}
```








Après exécution nous utilisons le script **plot_36.r** :

```
#plot_36.r
data1=read.table("test_36.txt",header=TRUE)
Title="Trace de C(t)^(AG-50) avec C(t)^(AG-50)=20microm"
plot(data1$time,data1$Ct,xlab = "Temps (h)",
     ylab = "Concentration (microm)",type="l",lwd=2,lty=1,col="blue",
     main=Title)
points(data1$time,data1$Ct,col="red",pch=19)
points(data1$time[6],data1$Ct[6],pch=3,cex=3,col="black")
text(10,5,paste("C(t=5)=",data1$Ct[6],"microm"))
```

qui affiche la figure suivante :

Trace de $C(t)^{(AG-50)}$ avec $C(t)^{(AG-50)} = 20\mu\text{M}$



-
1. à ce propos le **code obfuscation** fait maintenant l'objet d'une discipline et même de concours (<http://www.ioccc.org/>) 
 2. Pour Windows les exécutables produits se finissent en général par *.exe 
 3. L'utilisation des balises <> indique que le fichier .h concerné se trouve dans le chemin d'inclusion par défaut lors de la compilation (au besoin celui-ci peut être modifié par l'argument -I lors de la compilation). L'utilisation des balises "" implique de passer en revanche le chemin et le nom complet du fichier d'entête concerné mais permet aussi d'importer des fichiers différents de fichiers .h 
 4. [https://en.wikipedia.org/wiki/Naming_convention_\(programming\)](https://en.wikipedia.org/wiki/Naming_convention_(programming)) 
 5. https://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange 
 6. Il est toujours possible de récupérer le tableau de char associé à un string à l'aide de la méthode `c_str()` 
 7. Une description globale de ces fonctions est donnée ici : <http://www.cplusplus.com/reference/string/> pour la conversion de chaînes en valeurs et ici: http://www.cplusplus.com/reference/string/to_string/ pour les conversions inverses 
 8. Ce support de cours offre une présentation suffisante pour développer et utiliser des classes à des fins de modélisation mais n'a pas la prétention d'avoir une approche exhaustive du sujet 
 9. Pour éclairer et approfondir ces notions le site suivant <https://openclassrooms.com/courses/programmez-avec-le-langage-c/introduction-la-verite-sur-les-strings-enfin-devoilee> propose une approche concrète intéressante 
 10. [https://en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science)) 
 11. <https://gcc.gnu.org/onlinedocs/gcc-7.2.0/cpp/Duplication-of-Side-Effects.html> 
 12. Le système biologique étudié 
 13. Il faut entendre ici classe et objet au sens informatique du terme 