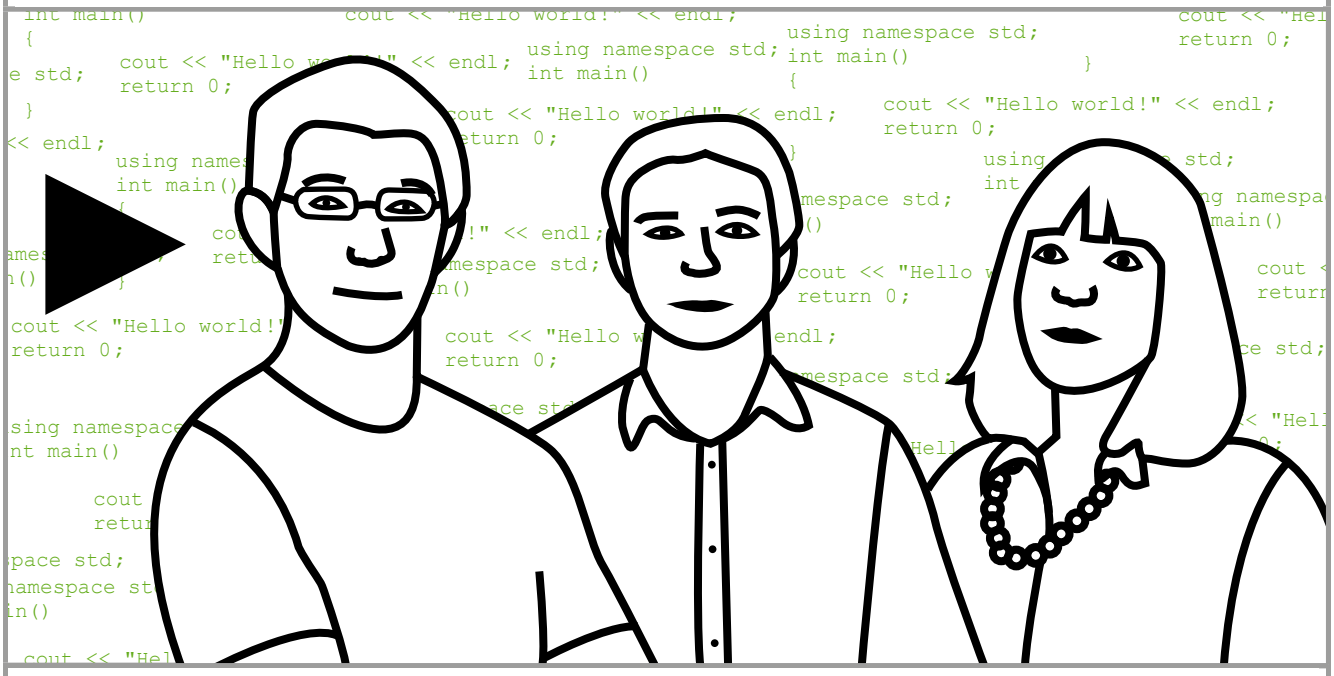
INITIATION À LA PROGRAMMATION EN C++

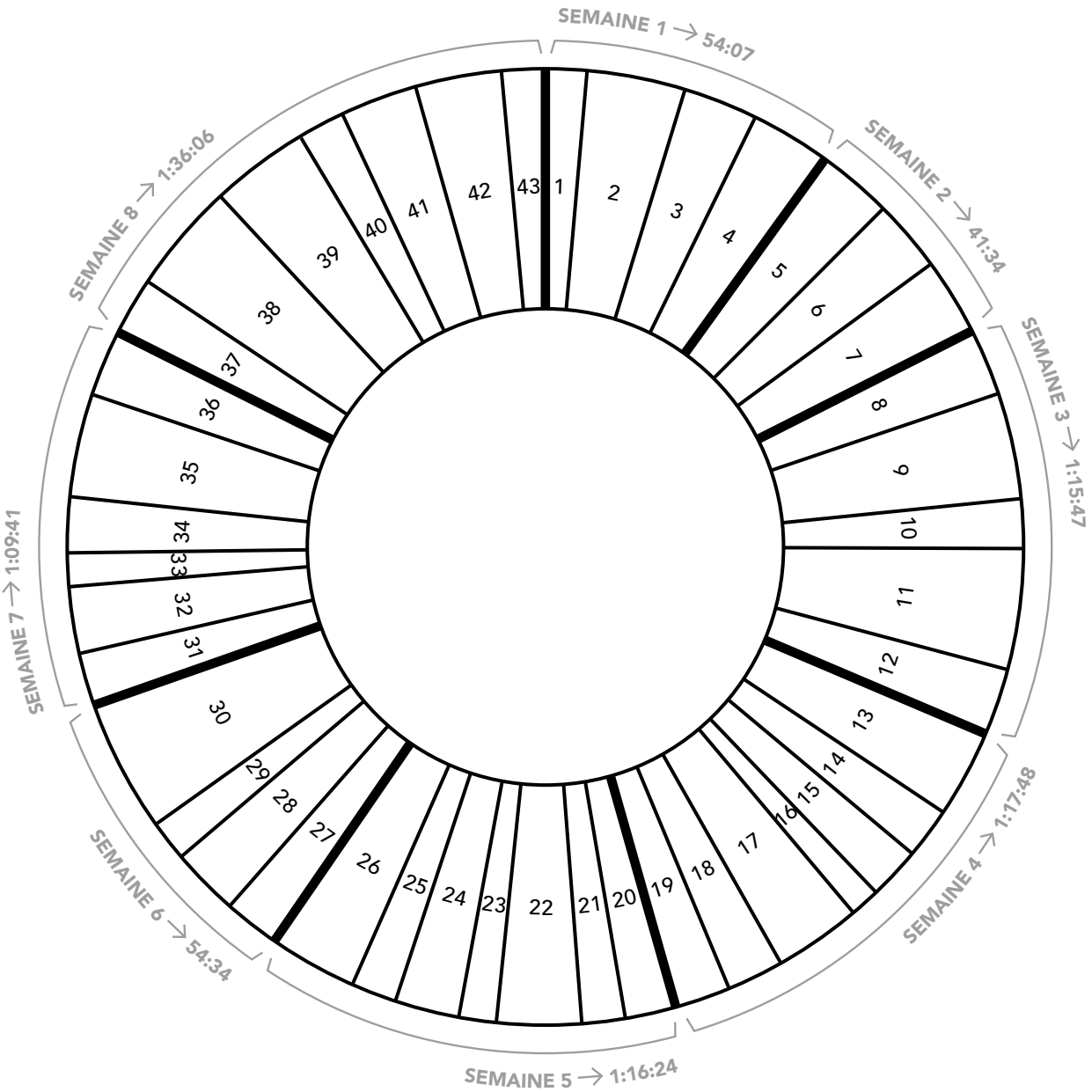


Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

LEARN FASTER, LEARN BETTER!

BOOCs EPFL

Initiation à la programmation en C++
Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit



CONTENU

1. INTRODUCTION

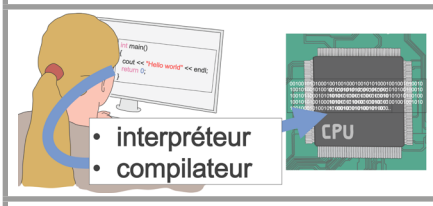


FIGURE 1

1:15 9:55

Traduction du programme par le compilateur ou l'interpréteur.

La programmation consiste à écrire des séquences d'instructions dans un langage compréhensible par les humains, tel que le C++, qui seront ensuite traduites en instructions élémentaires directement exécutables par le microprocesseur de l'ordinateur. C'est un programme appelé « compilateur » qui a pour charge de traduire un programme rédigé en C++, en une séquence d'instructions exécutable par le microprocesseur.

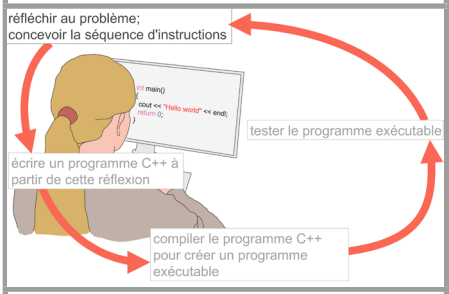


FIGURE 2

3:21 9:55

Étapes de la programmation.

ÉTAPES DE LA PROGRAMMATION

Pour un problème donné, il faut d'abord concevoir la séquence d'instructions que le programme exécutera. Ensuite, on écrit le programme C++ qui correspond à cette séquence d'instructions et on le compile pour créer un programme exécutable. La compilation peut échouer, parce que le programme ne respecte pas les règles du C++ ; pour que le programme soit compilé, il faut qu'il respecte les règles du C++. Enfin, quand le programme est compilé, on le teste en l'exécutant. Il se peut que le programme ne fasse pas ce que l'on souhaite parce que l'on a mal conçu la séquence d'instructions au départ. Dans ce cas, il faut repenser la séquence d'instructions, modifier le programme, et continuer ce cycle de développement.

ÉCRITURE DU PREMIER PROGRAMME

On commence par ouvrir un environnement de développement. D'usage, quand on apprend un nouveau langage, on commence avec un programme qui affiche le message « Hello world ». En C++, il s'écrit comme ceci :

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

COMPILATION

Avant d'appeler le compilateur, on sauvegarde le programme avec l'extension .cc ou .cpp. Ces extensions permettent d'indiquer qu'il s'agit d'un fichier C++. En colorisant le programme, l'environnement de développement indique qu'il sait maintenant que le texte est un fichier C++. La compilation se fait en cliquant sur le bouton *build*, qui lance à la fois la compilation et qui crée un programme exécutable : un message apparaît pour indiquer que la compilation s'est bien passée.

EXÉCUTION

Si la compilation réussit, on exécute le programme, en appuyant sur le bouton *execute*. Une fenêtre s'ouvre pour afficher le message « Hello world ». Dans le répertoire, on trouve maintenant un nouveau fichier, « helloworld ! » sans extension ou avec l'extension .exe. Il s'agit du fichier exécutable que l'on vient de créer. En général, on peut aussi le lancer directement en cliquant dessus.

EN CAS D'ERREUR

Il est probable de faire une faute de frappe en rédigeant le programme. Pour minimiser les fautes, il faut :

– être rigoureux quand on tape les programmes en faisant attention à la présentation ;
– en cas d'erreur, corriger toujours la première erreur et recompiler ; il est probable que ceci corrige certaines des erreurs suivantes ;
– d'abord regarder le numéro de ligne indiqué par le compilateur pour trouver l'erreur, en sachant que l'erreur peut être survenue à la ligne précédente ; ensuite interpréter le message d'erreur donné pour comprendre le problème.

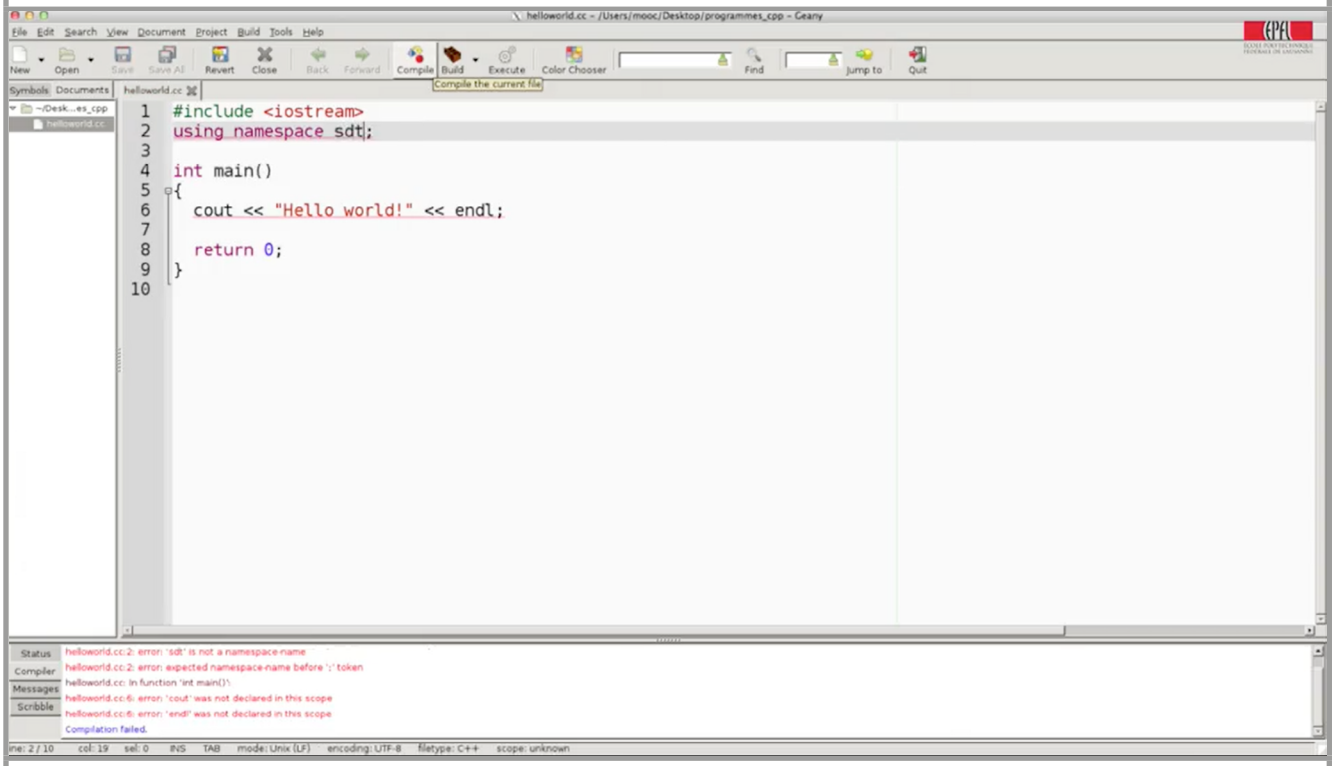


FIGURE 3

8:50 9:55

Exemple de l'erreur : on a tapé `sdt` au lieu de `std`.

2. VARIABLES

TRAITEMENTS ET DONNÉES

Un programme n'est rien d'autre qu'une séquence d'instructions travaillant avec des données. Ces séquences d'instructions sont assimilables à la notion de traitement. Les traitements opèrent sur les données, et les données influencent à leur tour les traitements opérés : selon la nature des données, différents types de traitements sont mis en œuvre.

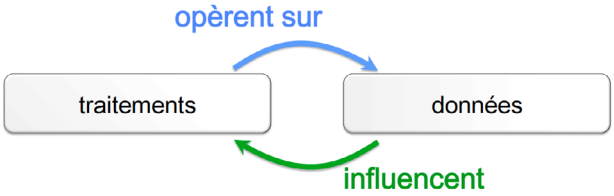


FIGURE 1

3:47 18:08

Interrelation entre les traitements et les données.

Une des briques de base pour exprimer des traitements est la notion d'***expression***, par exemple une expression arithmétique. Pour que les traitements opèrent sur une donnée, il est nécessaire de retrouver la valeur qui lui est associée à différents endroits du programme par le truchement de son nom.

VARIABLES

Une ***variable*** est un moyen en programmation de stocker une donnée, d'y associer un nom. Concrètement, dans un programme s'exécutant sur une machine, une variable correspond à une zone en mémoire, qui a un nom que l'on appelle un ***identificateur*** de la variable. Cette zone mémoire stocke une valeur, associée à l'identificateur. C++ exige également que l'on définisse le genre associé à chaque donnée, ce que l'on appelle le ***type*** de la variable. Il faut le spécifier au moment de la ***déclaration***.

```
int m(1);
int p(1), q(0);
double x(0.1), y;
```

FIGURE 2

9:30 18:08

Exemple de déclaration de variables.

DÉCLARATION DES VARIABLES

Pour utiliser la variable que l'on souhaite, on doit la déclarer dans le programme son identificateur. C'est au travers de cet identificateur que l'on accède à la valeur stockée dans cette variable. La déclaration d'une variable se fait comme suit :

```
type identificateur;
```

Initialiser une variable consiste à lui donner une valeur de départ. Ceci se fait habituellement au moment de la déclaration. :

```
type identificateur(valeur_initiale);
```

En C++, il est possible de déclarer une variable sans l'initialiser. Cependant, il faut toujours initialiser une variable avant d'utiliser sa valeur. En effet, les traitements avec une variable non initialisée sont complètement imprédictibles, et peuvent produire des résultats erronés. Il est alors déconseillé de déclarer une variable sans initialisation.

Le type associé à la variable au moment de sa déclaration est fondamental. Il conditionne le type de traitement que l'on peut réaliser avec la variable en question. Il faut être attentif au fait que le type d'une variable ne peut pas changer : lorsque l'on déclare une variable d'un certain type, cette variable garde ce type jusqu'à la fin de son existence dans le programme.

Il est possible en C++ de déclarer deux variables sur la même ligne comme le deuxième exemple de la figure 2. Dans ce cas, on n'indique qu'une seule fois le type associé, et l'on sépare les différentes déclarations par une virgule. Cette façon peut donner lieu à des ambiguïtés, donc il est conseillé de ne pas en abuser.

NOM DE VARIABLES

Il existe un certain nombre de conventions à respecter lorsque l'on déclare une variable :
- l'identificateur doit être constitué de lettres et de chiffres sans espace ni symboles ;
- le caractère souligné __ est considéré comme une lettre ;
- les accents ne sont pas autorisés ;
- le premier caractère est nécessairement une lettre ;
- l'identificateur ne doit pas être un mot réservé du langage (par exemple « `if` ») ;
- les majuscules et minuscules sont autorisées mais pas équivalentes ; les noms `ligne` et `Ligne` désignent deux variables différentes..

TYPE DE VARIABLES

Il existe plusieurs types de variable prédéfinis en C++. En voici quelques exemples :
- *`int`* : nombres entiers ;
- *`double`* : nombres réels en décimal ;
- *`char`* : caractères usuels (A…Z etc.) ;
- *`unsigned int`* : les entiers positifs.

AFFECTATIONS

Le changement de la valeur de la variable se fait au moyen de la notion d'***affectation***. L'affectation se pratique au moyen de l'opérateur d'affectation `=`. En affectant une valeur à une variable, on change sa valeur actuelle pour y stocker la nouvelle valeur. La valeur que la variable contient au préalable est effacée et remplacée par la nouvelle valeur. L'affectation d'une variable se fait comme suit :

```
nom_de_variable = expression;
```

Une expression peut se réduire à une simple valeur élémentaire ou à une expression plus complexe, par exemple avec les opérateurs arithmétiques usuels.

Il est important de ne pas confondre l'affectation avec une égalité mathématique. Le symbole `=` est le même, mais en mathématique et en programmation, il ne signifie pas la même chose.

Considérons l'exemple suivant :

Cas 1 : `a = b;`
Cas 2 : `b = a;`

En mathématique, ces deux lignes signifient que `a` et `b` ont toujours les mêmes valeurs. Cependant, en C++, les résultats que l'on obtient à la fin sont différents. Dans le cas 1, on copie la valeur de `b` dans `a` alors que dans le cas 2, on copie la valeur de `a` dans `b`. Dans chacun des deux cas `a` et `b` se retrouvent finalement avec les mêmes valeurs, mais dans le cas 1, `a` et `b` ont la valeur que `b` contient avant l'affection, alors que dans le cas 2, ils ont la valeur que `a` contient avant l'affectation !

DÉCLARATION DE CONSTANTES

Il existe des situations où, une fois qu'une variable prend une valeur de départ, on souhaite garantir qu'elle ne change plus par la suite. Dans ce cas, il faut précéder la déclaration de la variable par le mot réservé `const`:

```
const type identificateur(valeur_initiale);
```

L'affectation de la valeur d'une variable constante cause une erreur à la compilation.

3. VARIABLES : LECTURE / ÉCRITURE

AFFICHAGE SUR L'ÉCRAN

L'affichage des valeurs sur l'écran se fait comme suit :

```
cout << valeur_1 << valeur_2 << ..... << valeur_n;
```

La ligne de l'affichage commence par le mot réservé `cout` qui représente le ***flot de sortie standard*** dans le programme, c'est-à-dire, en général, le terminal. On retrouve plusieurs sections d'affichages, regroupées avec le signe `<<` qui signifie que l'on veut afficher quelque chose. Cela représente le sens dans lequel circule l'information : l'information va de ce qui suit vers `cout`. Derrière le signe `<<`, on trouve différentes valeurs possibles d'affichage :

– une valeur littérale, entre `"..."`, qui écrit exactement la phrase donnée ;
– le nom d'une variable qui écrit la valeur de la variable stockée en mémoire ;
– une expression plus complexe (au sens C++) : par exemple, `2 * n`, qui affiche deux fois la valeur de la variable `n` ;
– le mot réservé `endl`, qui affiche un saut de ligne.

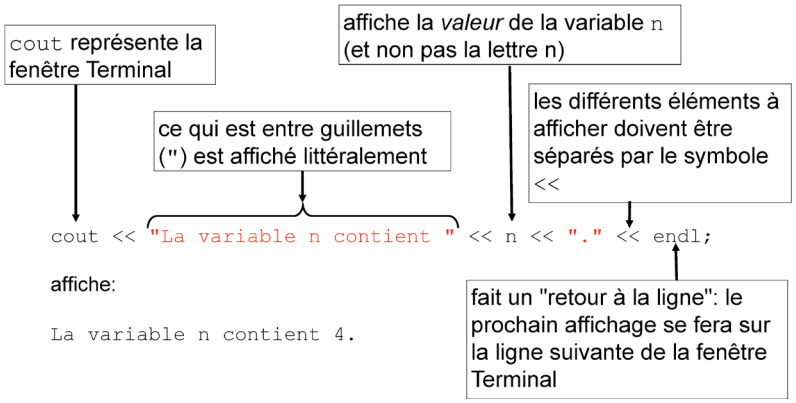


FIGURE 1

2:14 13:43

L'affichage sur l'écran.

`cout` et `endl` s'appellent en fait `std::cout` et `std::endl`. On peut les écrire simplement sans `std` parce qu'au début du programme on a mis l'instruction, `using namespace std`. Ceci permet d'éviter d'écrire à chaque fois `std::` devant les mots réservés qui sont dans ce que l'on appelle le ***namespace de la bibliothèque standard.***

```
int a(1);
int b(2);

a = b;
b = a;

cout << a << ", " << b << endl;
```

FIGURE 2

7:19 13:43

Exemple de mauvais échange de valeur.

ÉCHANGE DE VALEURS

Lorsque l'on veut échanger la valeur de deux variables, `a` et `b`, il ne faut pas procéder comme dans la figure 2. En effet, après les instructions de la figure, on retrouve, pour les deux variables, la même valeur que `b` contient au départ.

```
int a(1);
int b(2);
int temp(a);

a = b;
b = temp;
```

FIGURE 3

9:04 13:43

Échange de valeur entre deux variables.

Pour réellement échanger la valeur des deux variables, il faut utiliser une variable intermédiaire comme dans la figure 3. On crée une variable intermédiaire dans laquelle on recopie la valeur de `a`. Puis on recopie la valeur de `b` dans `a` et la valeur de `temp` dans `b`.

LECTURE AU CLAVIER

Lorsque l'on demande à l'utilisateur la valeur d'une variable, on peut la lire depuis le clavier. La lecture d'une valeur se fait grâce à l'instruction `cin` comme suit :

```
cin >> nom_variable;
```

`cin` représente le ***flot d'entrée standard*** dans le programme, c'est-à-dire en général le clavier.

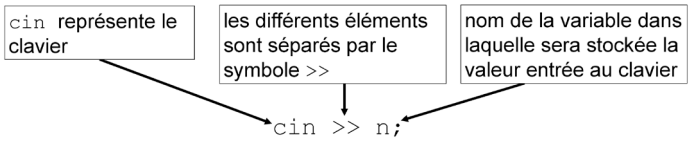


FIGURE 4

11:07 13:43

Lecture d'une valeur depuis le clavier.

On retrouve un signe `>>`, qui est similaire au signe `<<` que l'on utilise pour l'affichage, mais dirigé dans l'autre sens. Il représente le fait que l'on aille de `cin` vers la mémoire. Derrière ce signe, on a le nom de la variable dans laquelle on stocke la valeur lue depuis le clavier. Il faut faire attention à ce que ce qui suit le signe `>>`, soit le nom d'une variable. Il est interdit de mettre un message ou quoique ce soit d'autre.

On peut lire plusieurs variables à la suite, les unes derrières les autres. Il faut simplement rajouter à chaque fois le signe de lecture `>>` comme suit :

```
cin >> nom_variable1 >> nom_variable2 >> .... >> nom_variable_n;
```

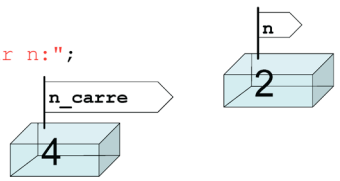
Cependant, il est préférable d'écrire en plusieurs lignes séparées pour que cela soit plus clair.

```
int n;

cout << "Entrez une valeur pour n:";
cin >> n;

int n_carre;
n_carre = n * n;

cout << "La variable n contient " << n << "." << endl;
```



Ce qui s'affiche:

```
Entrez une valeur pour n:
2
La variable n contient 2.
```

FIGURE 5

13:28 13:43

Déroulement du programme.

LEARN FASTER, LEARN BETTER!
BOOCs EPFL
Initiation à la programmation en C++
Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

4. EXPRESSIONS

Une ***expression*** apparaît par exemple dans une affectation: `nom_de_variable = expression;`. Dans l'affectation, l'expression est forcément à droite du signe `=`, puisqu'elle donne la valeur calculée à la variable, qui se trouve à gauche du signe `=`. Une expression peut être une valeur littérale comme `4` ou `3.14` ou une formule qui contient des ***opérateurs***. Remarquons que l'on écrit `3.14` au lieu de `3,14` puisque C++ utilise la notation anglo-saxonne.

TYPES DES VALEURS LITTÉRALES

Quand on écrit des expressions, les valeurs littérales ont leur propre type, comme les variables. Ainsi,
- `1` est de type `int`.;
- `1.0` et `1.` sont de type `double`. Il est préférable d'écrire `1.0` pour une meilleure lisibilité;
- les valeurs de types double peuvent s'écrire dans la notation scientifique, par exemple `3.1e2` qui vaut $3{,}1 \times 10^2$.

OPÉRATEURS

On dispose des quatre opérateurs arithmétiques usuels `+` (addition), `-` (soustraction), `*` (multiplication) et `/` (division). Notons que dans le cas de la division, si les deux valeurs qui interviennent sont de type `int`, on a une ***division entière***, qui rend le quotient de la division: par exemple, `1/2` vaut `0` car `1 = 2*0 + 1`. En revanche, si l'une des deux valeurs est de type `double`, l'autre valeur est tout d'abord convertie en `double`, et l'on obtient le résultat d'une division classique: par exemple `1/2.0` vaut `0.5`.

En C++, on dispose également des opérateurs, `+=`, `-=`, `*=` et `/=`. Par exemple, il est équivalent d'écrire `a+=b` ou `a=a+b`. Dans le cas des `int` seulement, on dispose de l'opérateur ***modulo***, qui se note `%`: il renvoie le reste de la division entière.

Il existe aussi des opérateurs d'***incrémentation*** et de ***décrémentation*** qui se notent respectivement `++` et `--`. Il s'agit d'ajouter 1 ou de soustraire 1 à des variables: par exemple, `++i` est équivalent à `i = i + 1`. Ces opérateurs sont importants dans le cas des boucles `for`, présentées dans la leçon 8.

```
double x(1.5);
int n;

n = 3 * x;
```

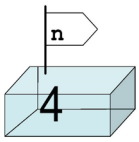


FIGURE 1

7:06 14:50

Perte de précision lors d'une affectation.

AFFECTATION D'UNE VALEUR DÉCIMALE À UNE VARIABLE ENTIÈRE

Lorsque l'on essaie d'affecter une valeur décimale à une variable de type `int`, le compilateur convertit la valeur de type `double`, en une valeur de type `int` pour effectuer l'affectation. Cette conversion se fait en perdant la partie fractionnaire. On dispose également de la conversion de `int` vers `double`. Cependant, ces conversions automatiques de types sont un cas très particulier. En règle générale, le C++ est un langage fortement typé. Par exemple, il exige que, dans une affectation, les valeurs qui se trouvent à gauche et à droite du signe `=` soient de même type.

FONCTIONS MATHÉMATIQUES

On peut utiliser des fonctions mathématiques dans des expressions. Pour cela, il faut ajouter la ligne `#include<cmath>` au début du programme. Par exemple, si l'on veut calculer le sinus d'un angle, on utilise la fonction `sin(angle)`, avec *angle* en radians.

- `sin` les fonctions trigonométriques fonctionnent en radians
- `cos`
- `tan`
- `asin` sinus inverse ou arc sinus
- `acos`
- `atan`
- `atan2` `atan2(y, x)` fournit la valeur de l'arc-tangente de `y / x`
- `sinh` sinus hyperbolique ou sh
- `cosh`
- `tanh`
- `exp`
- `log` logarithme népérien ou ln
- `log10` logarithme à base 10 ou log
- `pow` `pow(x, y)` fournit la valeur de x^y
- `sqrt` racine carrée
- `ceil` `ceil(x)` renvoie le plus petit entier qui ne soit pas inférieur à `x`: `ceil(2.6) = 3`
- `floor` `floor(x)` renvoie le plus grand entier qui ne soit pas supérieur à `x`: `floor(2.6) = 2`
- `abs` valeur absolue

FIGURE 2

12:58 14:50

Liste de quelques fonctions mathématiques disponibles en C++.

On dispose également des constantes mathématiques, par exemple `M_PI` pour π. Même si elles ne sont pas définies officiellement par le standard du C++, la plupart des compilateurs les définissent.

LEARN FASTER, LEARN BETTER!
BOOCs EPFL
Initiation à la programmation en C++
Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

5. BRANCHEMENTS CONDITIONNELS

STRUCTURE DE CONTRÔLE

Jusqu'à présent, les programmes sont simplement constitués d'instructions que l'on exécute les unes après les autres et les données n'ont aucune influence sur ces traitements. Pour changer le comportement strictement linéaire de l'exécution, on a ce que l'on appelle des ***structures de contrôle***. Il existe trois types de structures de contrôles :

- ***branchements conditionnels*** (leçon 5) : choisir une séquence de traitements ou une autre en fonction de certaines valeurs ;
- ***itérations*** (leçons 8-10) : répéter toute une séquence d'instructions sur une suite d'éléments ;
- ***boucles conditionnelles*** (leçon 11) : répéter toute une séquence d'instructions à une certaine condition.

BRANCHEMENTS CONDITIONNELS

Les branchements conditionnels permettent de sauter ou choisir certaines parties du programme à exécuter. En règle générale, le branchement conditionnel s'écrit comme suit :

```
if (condition) {
      //instuctions1
} else {
      //instructions2
}
```

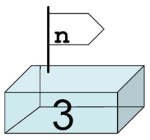
Le mot-clé `if` qui permet de faire un choix est suivi d'une condition entre parenthèses, sur laquelle on revient dans les leçons 8-9. Si la condition est vérifiée, alors `if` exécute le bloc d'instructions qui est indiqué dans les accolades. On a également une partie introduite par le mot-clé `else`. Lorsque la condition n'est pas vérifiée, alors `else` exécute le deuxième bloc d'instructions entre les accolades.

```
int n;

cout << "Entrez votre nombre:" << endl;
cin >> n;

if (n < 5) {
  cout << "Votre nombre est plus petit que 5." << endl;
} else {
  cout << "Votre nombre est plus grand ou egal a 5." << endl;
}

cout << "Au revoir" << endl;
```



Ce qui s'affiche dans la fenêtre Terminal:

```
Entrez votre nombre:
3
Votre nombre est plus petit que 5.
Au revoir
```

FIGURE 1

5:19 13:46

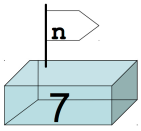
Exécution d'un bloc d'instructions lorsque la condition est vérifiée.

```
int n;

cout << "Entrez votre nombre:" << endl;
cin >> n;

if (n < 5) {
  cout << "Votre nombre est plus petit que 5." << endl;
} else {
  cout << "Votre nombre est plus grand ou egal a 5." << endl;
}

cout << "Au revoir" << endl;
```



Ce qui s'affiche dans la fenêtre Terminal:

```
Entrez votre nombre:
7
Votre nombre est plus grand ou egal a 5.
Au revoir
```

FIGURE 2

5:19 13:46

Exécution de l'autre bloc d'instructions lorsque la condition est fausse.

Un bloc d'instructions est l'ensemble des instructions que l'on veut contrôler, sur lesquelles on veut effectuer un branchement. On peut regrouper plusieurs instructions que l'on veut dans un bloc. Lorsqu'un bloc ne comprend qu'une seule instruction, l'accolade est optionnelle. Cependant, il est préférable de systématiquement les utiliser afin de rendre le code plus lisible et plus maintenable.

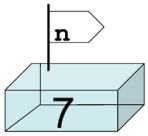
Le bloc `else` est également tout à fait optionnel: on peut écrire un seul `if` sans `else`. Dans ce cas, si la condition n'est pas vérifiée, alors on continue simplement à exécuter le programme sans avoir exécuté les instructions particulières du bloc `if`.

```
int n;

cout << "Entrez votre nombre:" << endl;
cin >> n;

if (n < 5) {
  cout << "Votre nombre est plus petit que 5." << endl;
}

cout << "Au revoir" << endl;
```



Ce qui s'affiche dans la fenêtre Terminal:

```
Entrez votre nombre:
7
Au revoir
```

FIGURE 3

8:50 13:46

Un branchement conditionnel sans bloc `else`.

CHOIX IMBRIQUÉS

Les instructions dans les blocs sont des instructions C++ tout à fait générales. En particulier, elles peuvent elles-mêmes contenir d'autres `if`, comme dans la figure 4 : il s'agit alors d'un ***choix imbriqué***. Il ne faut pas en abuser, car le code devient vite illisible.

```
if (x == y) {
  if (y == z) {
    cout << "Les trois valeurs sont egales." << endl;
  } else {
    cout << "Seules les deux premieres valeurs sont egales." << endl;
  }
} else {
  if (x == z) {
    cout << "Seules la premiere et la troisieme valeurs sont egales." << endl;
  } else {
    if (y == z) {
      cout << "Seules les deux dernieres valeurs sont egales." << endl;
    } else {
      cout << "Les trois valeurs sont differentes." << endl;
    }
  }
}
```

FIGURE 4

9:39 13:46

Utilisation d'un choix imbriqué.

6. CONDITIONS

OPÉRATEURS DE COMPARAISON

Les branchements conditionnels ont besoin d'exprimer des ***conditions*** pour pouvoir fonctionner. Une condition en C++ est une expression qui retourne une valeur parmi deux possibles, soit `true`, soit `false`. Une condition est évaluée à `true` si elle est vraie, et évaluée à `false` sinon. Les conditions que l'on a employées dans la leçon 5 sont des ***conditions simples***, qui comparent deux expressions en utilisant des ***opérateurs de comparaison***. Les opérateurs de comparaison du langage C++ sont :

– `<` qui signifie « inférieur à » ;
– `>` qui signifie « supérieur à » ;
– `==` qui signifie « égal à », à ne pas confondre avec `=` qui représente l'affectation ;
– `<=` qui signifie « inférieur ou égal à » ;
– `>=` qui signifie « supérieur ou égal à » ;
– `!=` qui signifie « différent de ».

Les opérateurs de comparaison permettent de comparer non seulement les valeurs de deux variables, mais aussi les valeurs de deux expressions de façon plus générale. Lorsque l'expression se complexifie, il est conseillé de mettre les termes entre parenthèses, pour rendre l'expression plus lisible.

```
int a(1);
int b(2);

if (a == b) {
  cout << "Cas 1" << endl;
} else {
  cout << "Cas 2" << endl;
}

if (2 * a == b){
  cout << "b est egal au double de a." << endl;
}
```

affiche

```
Cas 2
b est egal au double de a.
```

FIGURE 1

5:04 13:14

Exemple d'utilisation des opérateurs de comparaison.

OPÉRATEURS LOGIQUES

Il est souvent nécessaire de combiner plusieurs de ces conditions simples pour formuler une condition plus complexe. Pour combiner des expressions simples, on utilise, les ***opérateurs logiques***:

– L'opérateur `and` ou `&&` (ET), qui fonctionne avec deux expressions logiques, retourne `true` uniquement lorsque chacun des opérandes vaut `true`. Si l'un des deux opérandes et a fortiori les deux, sont `false`, le résultat de l'évaluation de l'expression globale est aussi `false`.
– L'opérateur `or` ou `||` (OU), qui fonctionne avec deux expressions logiques, retourne `true` si au moins l'un des deux opérandes est `true`. Elle retourne `false` uniquement lorsque les deux opérandes retournent `false`.
– L'opérateur `not` ou `!` (NON), qui n'attend qu'une seule expression logique, retourne la négation de la valeur de son opérande. Lorsque l'opérande est évalué à `false`, l'évaluation de l'expression globale avec le `not` retourne la négation de `false`, c'est-à-dire `true`. De même, lorsque l'évaluation de l'opérande retourne `true`, l'expression globale retourne `false`.

faux
faux

```
if ((n >= 1) and (n <= 10)) {
  cout << "correct" << endl;
} else {
  cout << "incorrect" << endl;
}
```

FIGURE 2

8:55 13:14

Exemple d'utilisation de l'opérateur logique ET.

vrai
vrai

```
if ((m >= 0) or (n >= 0)) {
  cout << "au moins une valeur est positive" << endl;
} else {
  cout << "les deux valeurs sont negatives" << endl;
}
```

FIGURE 3

11:06 13:14

Exemple d'utilisation de l'opérateur logique OU.

7. ERREURS DE DÉBUTANT, LE TYPE BOOL

ERREURS CLASSIQUES

La première erreur fréquente est d'écrire le test d'égalité avec le symbole `=`, qui est pour l'affectation, au lieu du symbole `==`. Par exemple, le code `if(a = 1)` est accepté par le compilateur, mais il *affecte* la valeur 1 à la variable sans tester si `a` est égal à `1`. Même si le compilateur accepte le code, la plupart des compilateurs affichent un *warning*, c'est-à-dire, un message d'avertissement.

Une deuxième erreur courante correspond au code suivant dont l'exécution peut paraître étonnante :

```
if (a == 1); //!!!
    cout << "a vaut 1" << endl;
```

Quelle que soit la valeur de `a`, le programme affichera toujours le message `a vaut 1`. L'erreur provient du point-virgule après la condition du `if` qui est considéré comme une instruction qui ne fait rien. Le code suivant est identique pour le compilateur :

```
if (a == 1)
;
cout << "a vaut 1" << endl;
```

Il est donc clair que l'instruction `cout << "a vaut 1" << endl;` est en réalité située *après* le `if` et n'est donc pas soumise à sa condition.

Une troisième erreur courante consiste à oublier les accolades comme dans la figure 1. Dans ce cas, le compilateur affiche le message d'erreur : `syntax error before else`. En fait, la première instruction `cout` est considérée comme étant à l'intérieur du branchement conditionnel, mais la deuxième instruction est considérée comme étant après le branchement conditionnel. Le compilateur tombe sur le mot-clé `else`, et pour lui, il n'y a aucune instruction `if`, qui se rattache à `else`, puisque l'on est déjà sorti du branchement conditionnel.

```
if (n < p)
  cout << "n est plus petit que p" << endl;
  max = p;
else
  cout << "n est plus grand ou egal a p" << endl;
```

FIGURE 1

2:42 14:34

Oubli des accolades dans un bloc d'instructions.

TYPE BOOL

Le type ***bool*** est le type des conditions, qui permet de déclarer des variables contenant une valeur de vérité. Une variable de type `bool`, souvent appelée un booléen, ne peut prendre que deux valeurs, `true` et `false`. On déclare et initialise un booléen comme les variables des autres types. On peut également utiliser des opérateurs logiques, `and`, `or` et `not`, entre variables de type `bool`, ou utiliser les booléens comme des conditions dans un branchement conditionnel.

```
int a(1);
int b(2);

bool c(true);
bool d(a == b);
bool e(d or (a < b));

if (e) {
  cout << "e vaut true" << endl;
}
```

FIGURE 2

11:52 14:34

Utilisation des booléens.

8. ITÉRATIONS INTRODUCTION

Une ***itération*** ou une ***boucle for*** est une structure de contrôle qui permet de répéter les mêmes instructions. Une boucle `for` commence par le mot-clé `for`, suivi de la déclaration et de l'initialisation d'une variable qui ne sont exécutées qu'une seule fois avant d'entrer dans la boucle. Cette variable sert à contrôler le nombre de tours de boucle. Ensuite, on teste une condition avant l'exécution de chaque tour de boucle : si elle est vraie, on exécute le bloc d'instructions entre les accolades et sinon, on sort de la boucle, en exécutant les instructions après la boucle. Puis, une incrémentation, exécutée à la fin de chaque tour de boucle, permet de changer la valeur de la variable considérée par la boucle `for`. La syntaxe d'une boucle `for` est la suivante :

```
for(déclaration_et_initialisation; condition; incrémentation) {
      // les instructions à répéter
}
```

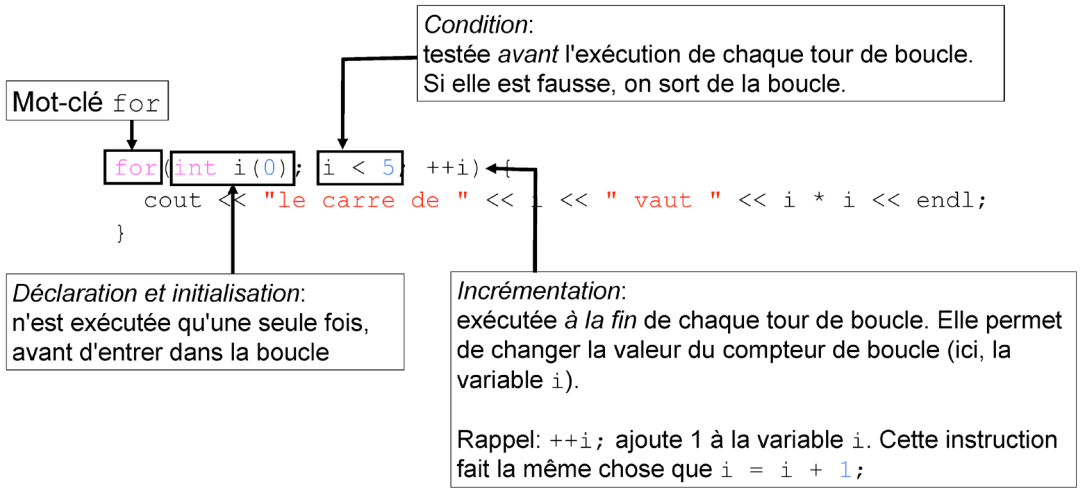


FIGURE 1

1:28 12:37

Exemple d'une boucle `for` qui affiche le carré des cinq premiers entiers positifs.

Notons que l'on ne met pas de point-virgule après les parenthèses. Les accolades pour le bloc d'instructions ne sont obligatoires qu'avec plusieurs instructions à exécuter. Cependant, même si l'on n'a qu'une seule instruction, il est conseillé de les garder pour minimiser les erreurs. Remarquons que la variable déclarée par la boucle `for` n'existe que dans le corps de la boucle et ne peut plus être utilisée après la boucle.

A priori, une condition et une incrémentation doivent porter sur la variable déclarée au début, même s'il n'y a aucune obligation. Si la condition ne devient jamais fausse, les instructions dans le corps sont répétées indéfiniment. Il existe également, depuis C++ 2011, une autre forme de boucle `for` pour les tableaux, qui est présentée dans la leçon 22.

La figure 2 présente un exemple d'utilisation d'une boucle `for` qui affiche une table de multiplication.

On peut remplacer:

```
cout << "5 multiplie par 1 vaut " << 5 * 1 << endl;
cout << "5 multiplie par 2 vaut " << 5 * 2 << endl;
cout << "5 multiplie par 3 vaut " << 5 * 3 << endl;
cout << "5 multiplie par 4 vaut " << 5 * 4 << endl;
cout << "5 multiplie par 5 vaut " << 5 * 5 << endl;
...
```

par

```
for(int i(1); i <= 10; ++i) {
  cout << "5 multiplie par " << i << " vaut " << 5 * i << endl;
}
```

FIGURE 2

8:31 12:37

Exemple d'une boucle `for` qui affiche la table de multiplication de 5.

9. ITÉRATIONS : APPROFONDISSEMENTS ET EXEMPLES

EXEMPLE D'AUTRES FORMES DE BOUCLE FOR

Voici quelques formes de boucle `for` variées de celles de la leçon 8 :

- `for(int p(0); p < 10; p += 2)` : la variable `p` est incrémentée de 2 à chaque itération, prenant ainsi les valeurs 0, 2, 4, 6, 8.
- `for(int k(10); k > 0; --k)` : la variable `k` est décrémentée de 10 à 1, prenant les valeurs 10, 9, ..., 0.
- `for(int i(0); i >= 0; ++i)` : la boucle répète indéfiniment car la condition `(i >= 0)` est toujours vraie ici.

ERREURS AVEC BOUCLE FOR

Une boucle `for` se répète indéfiniment :

- lorsque l'on se trompe sur la condition, par exemple,

```
for(int i(0); i > -1; ++i);
```

- lorsque l'on se trompe sur l'instruction de l'incrémentation, par exemple,

```
for(int i(0); i < 10; ++j).
```

Il faut aussi faire attention à ne pas mettre le point-virgule à la fin de la boucle `for`, comme ci-dessous.

```
for(int i(0); i < 10; ++i);
    cout << "bonjour" << endl;
```

Dans ce code, le point-virgule est considéré comme étant dans le corps de la boucle et l'instruction cout est considérée comme étant en dehors de la boucle. Donc la boucle `for` répète l'instruction vide et l'instruction `cout` n'est exécutée qu'une seule fois (après la boucle). Il est également important de ne pas oublier les accolades, comme ci-dessous.

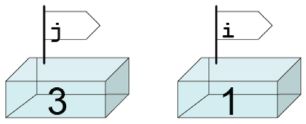
```
for(int i(0); i < 5; ++i)
cout << "i = " << i << endl;
cout << "Bonjour" << endl;
```

La première instruction `cout` est considérée comme étant à l'intérieur de la boucle `for` et la deuxième instruction `cout` est considérée comme étant après la boucle. Donc, la première instruction est répétée plusieurs fois tandis que la deuxième n'est exécutée qu'une seule fois. Il faut les mettre entre les accolades pour que les deux instructions soient effectivement répétées.

Il faut éviter de modifier la variable qui sert à contrôler le nombre de tours de boucle à l'intérieur du corps de la boucle car cela risque d'être source de confusion : en général, la boucle `for` modifie également la variable de son côté et le comportement final de la boucle ne sera peut-être pas celui escompté au départ. De plus, un relecteur risque de ne pas s'apercevoir que l'on modifie la valeur de la variable à l'intérieur du corps de la boucle.

BOUCLES IMBRIQUÉES

On peut mettre une boucle `for` à l'intérieur d'une autre boucle `for`. La figure 1 en fournit un exemple : on affiche une table de multiplication de 2 à 10.




```
for(int j(2); j <= 10; ++j) {
  cout << "Table de multiplication par " << j << ":" << endl;
  for(int i(1); i <= 10; ++i) {
    cout << j << " multiplie par " << i << " vaut " << j * i << endl;
  }
}
```

```
Table de multiplication par 2:
2 multiplie par 1 vaut 2
2 multiplie par 2 vaut 4
...
2 multiplie par 10 vaut 20
Table de multiplication par 3:
3 multiplie par 1 vaut 3
```

FIGURE 2

18:51 19:17

Affichage de table de multiplication de 2 à 10.

10. ITÉRATIONS : QUIZ

Afin de tester notre connaissance des boucles, essayons de trouver ce qu'affiche l'exécution du code de la figure 1. Il est conseillé de répondre au quiz dans les figures avant de lire la suite.

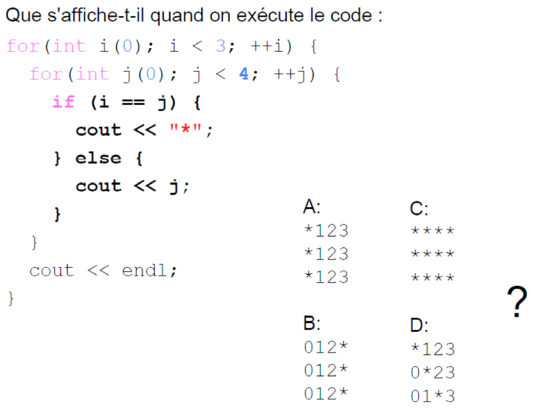


FIGURE 1

0:09 9:04

Première question.

La bonne réponse est la réponse D. La variable `i`, qui est déclarée dans la première boucle `for`, prend les valeurs `0`, `1` et `2` et la variable `j`, déclarée par la deuxième boucle `for`, prend de même les valeurs `0`, `1`, `2`, `3`. Dans la deuxième boucle, si la condition `(i==j)` est vérifiée, on affiche `*` ; sinon, on affiche la valeur de `j`. On recommence chaque fois la boucle `j` avec une valeur différente de `i`, jusqu'à la fin de la première boucle `for`. Notons que le retour à la ligne, indiqué par `cout<<endl;`, est effectué après l'exécution de la deuxième boucle `for`.

La deuxième question, sur la figure 2 est un peu plus difficile que la première.

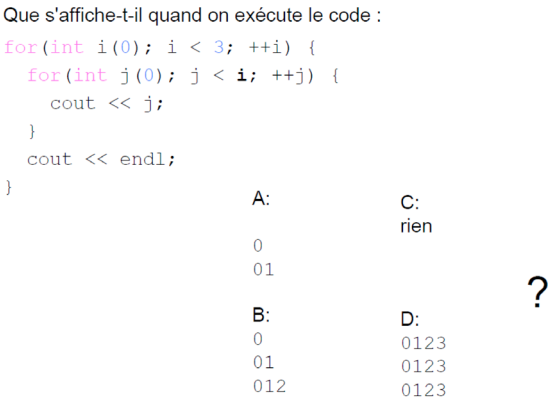


FIGURE 2

5:11 9:04

Deuxième question.

La bonne réponse est la réponse A. Remarquons que l'on utilise la valeur de la variable `i`, qui est déclarée dans la première boucle `for`, pour la condition d'arrêt de la deuxième boucle `for`. La variable `i`, déclarée dans la première boucle `for`, prend les valeurs `0`, `1` et `2`, tandis que la variable `j`, déclarée dans la deuxième boucle `for`, prend les valeurs des entiers positifs strictement inférieurs à la valeur de `i`, c'est-à-dire, les valeurs entre `0` et `i-1`. Donc, pour chaque première boucle `for`, on affiche les entiers entre `0` et `i-1`, avec un retour à la ligne après chaque exécution de la deuxième boucle `for`.

11. BOUCLES CONDITIONNELLES

Les itérations, vues dans les leçons 8-10, permettent de répéter les instructions lorsque le nombre de répétitions est connu *a priori*. Cependant, il existe des situations où l'on souhaite répéter un traitement tant qu'une condition est vérifiée. On utilise alors **des boucles conditionnelles**, c'est-à-dire des boucles `do...while` ou `while`. La figure 1 fournit un exemple de l'utilisation de boucle `do...while`: on répète les instructions pour demander à l'utilisateur d'entrer le nombre de notes, tant que le nombre entré est négatif ou nul.

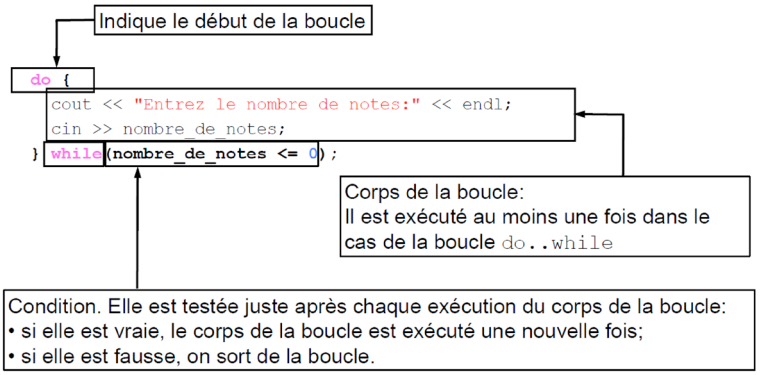


FIGURE 1

2:36 22:31

Exemple d'une boucle `do...while`.

SYNTAXE DE LA BOUCLE DO...WHILE

La syntaxe d'une boucle `do...while` est la suivante:

```
do {
    instructions
} while (condition);
```

Notons qu'il existe un point-virgule à la fin de la boucle `do...while`, après la condition.

Les traitements du corps sont répétés tant que la condition, qui est obligatoirement entourée par les parenthèses, est vraie (évaluée à `true`). Comme pour l'instruction if, on peut utiliser des opérateurs logiques dans la condition d'arrêt de la boucle. Remarquons également que le corps de la boucle `do...while` est exécuté au moins une fois.

Si la condition ne devient jamais fausse, les traitements sont répétés indéfiniment. Il faut donc être attentif à bien formuler la condition d'arrêt pour éviter une boucle infinie.

SYNTAXE DE LA BOUCLE WHILE

Lorsque l'on souhaite tester la condition de la boucle avant même d'entrer dans la boucle, il faut utiliser la boucle `while`. La syntaxe de la boucle `while` est la suivante :

```
while(condition) {
    bloc
}
```

Le principe de fonctionnement de la boucle `while` est analogue à celui de la boucle `do...while`. La seule différence est que la condition est testée *avant* même d'entrer dans la boucle. Si la première évaluation de la condition est fausse, on passe à l'instruction suivant et le bloc n'est donc jamais exécuté.

```
int i(100);
do {
  cout << "bonjour" << endl;
} while (i < 10);
```
affichera une fois `bonjour`.

Dans les 2 cas, la condition `i < 10` est fausse.

```
int i(100);
while (i < 10) {
  cout << "bonjour" << endl;
}
```
n'affichera rien.

FIGURE 2

8:57 22:31

Différence entre les boucles `do...while` et `while`.

Notons qu'il n'y a pas de point-virgule à la fin de boucle `while` (c'est un bloc). Si l'on en met un, il est considéré comme étant dans le corps de la boucle et l'on exécute le corps de l'instruction vide.

CHOIX DE BOUCLE À UTILISER

Si le nombre de répétitions des traitements est connu a priori, il vaut mieux utiliser une boucle `for`, par exemple, si l'on souhaite calculer la moyenne d'une série de nombres.

Si le nombre d'itérations n'est pas connu a priori, on choisit une boucle conditionnelle `while` ou `do...while`. Lorsque l'on doit exécuter au moins une fois le corps, par exemple, pour demander une valeur à l'utilisateur entre deux bornes, il faut utiliser une boucle `do...while`. Sinon, on choisit la boucle `while`, où l'évaluation de la condition se fait avant même d'entrer dans le corps de la boucle.

12. BLOCS D'INSTRUCTIONS

En C++, les instructions peuvent être regroupées dans des blocs, indépendamment de toute structure de contrôle : il suffit d'entourer une séquence d'instructions avec les accolades, comme dans la figure 1.

```
{
  int i;
  double x;

  cout << "Valeurs pour i et x : " << endl;
  cin >> i >> x;
  cout << "Vous avez entré : i = " << i << i
       << ", x = " << x << endl;
}
```

FIGURE 1

0:32 12:18

Exemple d'un bloc.

écrivez:

```
if (i != 0) {
  int j(0);

  ...
  j = 2 * i;
  ...
}
```

plutôt que:

```
int j(0);
if (i != 0) {

  ...
  j = 2 * i;
  ...
}
```

FIGURE 2

2:51 12:18

Déclarer les variables au plus près de leur utilisation.

En C++, les blocs ont une grande autonomie : ils contiennent leurs propres variables, dont la déclaration et l'initialisation sont faites à l'intérieur même du bloc. Les variables qui sont déclarées dans un bloc sont les ***variables locales*** au bloc : on ne peut plus les utiliser dès que l'on quitte le bloc dans lequel elles sont définies.

Les variables déclarées hors de tout bloc, y compris du `main`, sont des ***variables globales***. Il ne faut jamais déclarer de variables globales car il est extrêmement difficile de suivre la valeur d'une variable globale, qui est accessible n'importe où dans le programme. La bonne pratique est de déclarer les variables au plus près de leur utilisation.

```
if (i != 0) {
  int j(0);

  ...
  j = 2 * i;
  ...
  if (j != 2) {
    int j(0);

    ...
    j = 3 * i;
    ...
  }
  ...
}
```

FIGURE 3

5:25 12:18

Exemple de variables de même nom, de différente portée.

PORTÉE

La ***portée*** d'une variable est l'ensemble des lignes de code où la variable est accessible. Par exemple, la portée d'une variable déclarée dans le corps de l'instruction `if` est le bloc contrôlé par l'instruction `if` : on ne peut plus l'utiliser dès que l'on quitte le bloc.

En C++, on peut avoir des variables de même nom mais de portée différente. Dans ce cas-là, la variable la plus proche est choisie lors de l'utilisation, par les ***règles de résolution de portée***.

Par exemple, dans la figure 3, on a un bloc contrôlé par une première instruction `if`, dans laquelle on déclare une variable `j`, et un autre bloc, contrôlé par une deuxième instruction `if` dans laquelle on déclare aussi une variable `j` de même nom. Alors, `j` utilisé dans le deuxième bloc réfère le `j` déclaré le plus proche, dans le deuxième bloc. La variable `j` déclarée dans le bloc du plus haut niveau est parfaitement utilisable, mais elle est masquée dans le deuxième bloc. Pour éviter toute ambiguïté, il est conseillé de ne pas nommer les différentes variables avec un même nom.

La portée d'une boucle `for` est un cas particulier. Dans la boucle `for`, on déclare une variable dont la portée est le bloc contrôlé par l'instruction for. Cette variable est locale à la boucle `for` et l'on ne peut plus l'utiliser après la boucle.

```
for(int i(0); i < 5; ++i) {
  cout << i << endl;
}
// A partir d'ici, on ne peut plus utiliser ce i
```

FIGURE 4

7:58 12:18

Portée dans les boucles `for`.

LEARN FASTER, LEARN BETTER!
BOOCs EPFL
Initiation à la programmation en C++
Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

13. FONCTIONS : INTRODUCTION

La fonction est un traitement qui opère sur des données. Jusqu'à maintenant les programmes que l'on a écrits sont constitués d'une séquence d'instructions sans organisation plus globale et sans partage des tâches répétées.

La fonction est une portion de programme que l'on définit à un endroit d'un programme afin de la réutiliser plusieurs fois à différents endroits du code, en évitant la duplication du code. Notons qu'il ne faut jamais dupliquer de code car cela rend le programme, inutilement long, difficile à comprendre et difficile à maintenir.

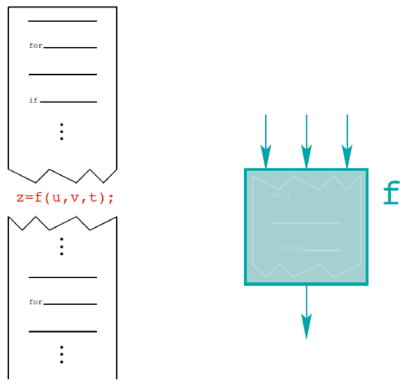


FIGURE 1

11:11 16:07

Illustration d'une fonction.

CARACTÉRISATION D'UNE FONCTION

Pour utiliser une fonction, on identifie le code à y mettre, on l'extrait dans un endroit du programme et l'on remplace la partie de code extraite par l'***appel*** à la fonction. La fonction reçoit des valeurs d'entrée nécessaires pour et fournit une valeur au reste du programme.

Une fonction est caractérisée par :
- un ***corps*** : la portion de programme réutilisée ou à mettre en évidence ;
- un ***nom*** : l'identificateur qui permet de faire référence à cette fonction ;
- des ***paramètres*** : l'ensemble des variables extérieures dont le corps a besoin pour fonctionner ;
- un ***type*** et une ***valeur de retour*** : la valeur que la fonction fournit au reste du programme.

L'utilisation d'une fonction dans le reste du programme se nomme un « appel à la fonction ».

TROIS FACETTES D'UNE FONCTION

Une fonction a trois facettes :
- Le ***prototype*** est un résumé de ce que doit faire la fonction. Il contient le nom, les paramètres qui sont les valeurs nécessaires pour le fonctionnement de la fonction, et le type de la valeur de retour de la fonction.
- La ***définition*** contient le prototype et le ***corps*** de la fonction, qui est le code exécuté lors de l'utilisation de la fonction.
- L'***appel*** correspond à l'utilisation de la fonction en lui donnant des valeurs effectives pour ses paramètres. La fonction fournit une valeur que l'on utilise dans une expression.

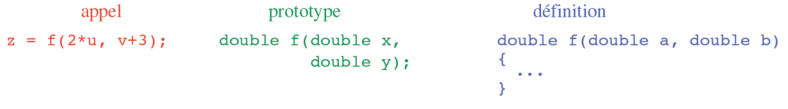


FIGURE 2

12:04 16:07

Les trois facettes d'une fonction.

En pratique le programmeur-concepteur, qui écrit la définition de la fonction, n'est pas forcément la même personne que le programmeur-utilisateur, qui utilise la fonction. Le programmeur-utilisateur a simplement besoin de connaître le prototype de la fonction pour l'utiliser, sans connaître son corps : le prototype sert donc d'accord entre le programmeur-utilisateur et le programmeur-concepteur.

LEARN FASTER, LEARN BETTER!
BOOCs EPFL
Initiation à la programmation en C++
Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

14. FONCTIONS : APPELS

Dans la leçon 13, on a vu que la notion de fonction est composée de trois facettes : le prototype, la définition et l'appel. Dans cette leçon, on détaille ce qui se passe au moment de l'***appel***. La figure 1 présente un programme qui affiche la moyenne de deux notes, entrées par l'utilisateur. Le calcul de la moyenne est réalisé au moyen d'un appel de fonction. Les arguments passés à la fonction au moment de l'appel correspondent aux paramètres attendus par la fonction pour qu'elle puisse s'exécuter.

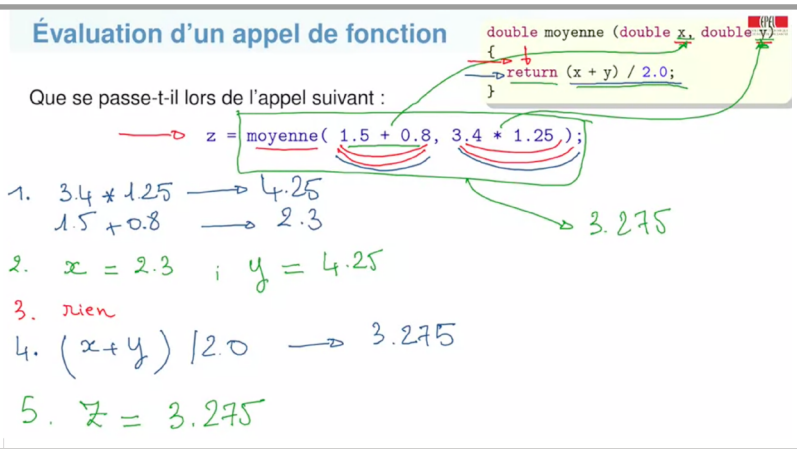


FIGURE 1

4:21 9:35

Appel de la fonction qui calcule la moyenne de deux valeurs.

Notons que l'on appelle usuellement ***paramètres*** les données nécessaires à la fonction pour qu'elle puisse s'exécuter telle que décrite dans le prototype, et ***arguments*** les valeurs que l'on passe effectivement à la fonction au moment de l'appel.

Dans le cas plus général où une fonction nécessite pour des paramètres entrants pour fournir en sortie une valeur concrète, l'appel de fonction se passe en cinq étapes :

1. Les expressions qui sont passées en argument à la fonction sont évaluées. Remarquons que l'on ne peut pas, en C++, présupposer de l'ordre selon lequel cette évaluation est faite.
2. Les valeurs résultant de l'évaluation des expressions sont affectées aux paramètres de la fonction.
3. Les paramètres de la fonction disposent désormais de valeurs concrètes avec lesquelles toutes les instructions du corps de la fonction sont exécutées.
4. L'expression qui suit la première commande `return` rencontrée à l'exécution est évaluée.
5. Le résultat de cette évaluation `return` est retourné comme résultat de l'appel : cette valeur remplace désormais l'expression de l'appel.

Il existe des situations où ce schéma en cinq étapes est simplifié :

– Les étapes 1 et 2 n'ont pas lieu lorsqu'une fonction fournit un résultat en sortie, mais n'a pas besoin d'arguments (elle n'a pas de paramètres).
– Les étapes 4 et 5 n'ont pas lieu lorsqu'une fonction réalise des traitements, mais ne fournit en sortie aucune valeur concrète, par exemple lorsqu'elle affiche simplement une valeur sur un terminal. Son type est alors `void`.
– L'étape 2 n'a pas lieu lors du passage par référence que l'on verra dans la leçon 15.

On peut aussi appeler une fonction pendant l'exécution d'une autre fonction : la figure 2 en fournit un exemple. La fonction `affiche_score` affiche le score d'un joueur, lequel est calculé au moyen de l'appel d'une fonction `score`.

```
int score (double points, double temps_jeu);
void affiche_score (double points, double temps_jeu);

void affiche_score(int joueur, double points, double temps)
{
  cout << "  Joueur " << joueur
       << score(points, temps) << " points" << endl;
}

int score (double points, double temps_jeu)
{ // ... comme avant ...
}
```

FIGURE 2

6:31 9:35

Exemple de l'appel d'une fonction dans une autre fonction.

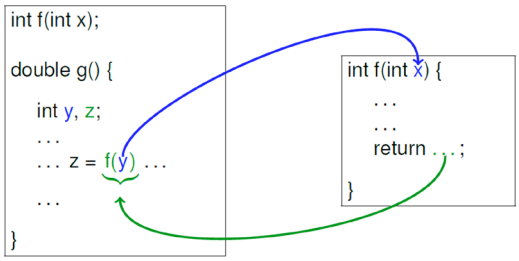


FIGURE 3

7:27 9:35

Résumé de l'appel fonctionnel.

Il existe un certain jargon utilisé au moment de l'appel :
- « Appeler la fonction `f` » : utiliser la fonction. Exemple : dans le code `x = 2*f(3);`, on *appelle* la fonction `f`.
- « La valeur est passée en argument » : (lors d'un appel) la valeur est copiée dans un paramètre de la fonction. Exemple : dans le code `x = 2*f(3);`, la valeur 3 est *passée* en argument.
- « La fonction retourne la valeur » : l'expression de l'appel de la fonction sera remplacée par la valeur retournée. Exemple : `cos(0)` retourne le cosinus de 0, donc retourne la valeur 1.

Notons qu'il ne faut jamais oublier de prototyper la fonction avant l'appel : seules les fonctions préalablement prototypées peuvent être appelées.

15. PASSAGE DES ARGUMENTS

Dans les exemples des leçons 13 et 14, les arguments passés à la fonction sont soit de simples valeurs, soit des expressions à évaluer. Dans cette leçon, on examine ce qui se passe lorsque les arguments passés à la fonction sont des variables.

En C++, il existe deux types de passage d'arguments :
- Lors d'un ***passage par valeur***, la fonction travaille sur une copie de l'argument dans la zone locale à la fonction. Toute altération n'a d'incidence que sur la zone locale et ne se répercute pas sur la variable passée en argument.
- Lors d'un ***passage par référence***, on indique que le paramètre de la fonction est un nom supplémentaire pour la variable passée en argument. Toute altération du paramètre altère aussi la variable passée en argument. Le passage par référence doit être explicitement indiqué dans l'entête de la fonction en utilisant le symbole `&` après le type ; par exemple, `double f(double& x)`.

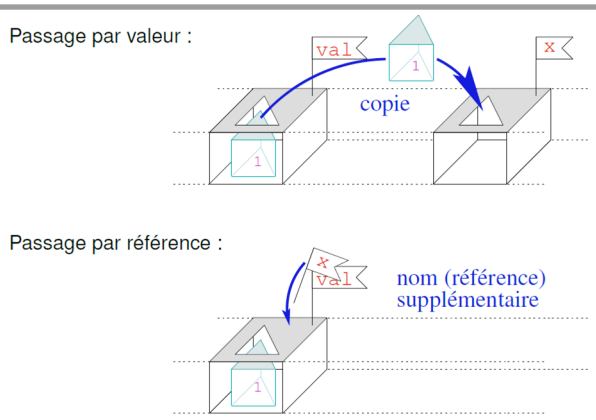


FIGURE 1

4:31 9:29

Schéma de passages d'argument.

La figure 2 présente l'exemple d'une fonction qui utilise le ***passage par valeur***. Lors de l'appel de la fonction `f()`, on passe la variable `val` par valeur. Cela signifie que `x` est une zone locale, et donc l'altération de `x` n'a d'incidence que sur `val`. Si l'on affiche `x`, on perçoit la modification de `x`, tandis que, si l'on affiche `val` après l'exécution de la fonction, on voit que la valeur d'origine reste inchangée.

```
void f(int x) {
  x = x + 1;
  cout << "x=" << x;
}
int main() {
  int val(1);
  f(val);
  cout << " val=" << val << endl;
  return 0;
}
```

L'exécution de ce programme produit l'affichage :

```
x=2 val=1
```

FIGURE 2

4:57 9:29

Exemple de passage par valeur.

La figure 3 présente le même exemple, mais d'une fonction qui utilise le passage par référence. Au moment de l'appel de la fonction, on indique que `x` est un autre nom pour `val`. Donc, lorsque l'on altère `x`, on altère également `val`. L'affichage de `val`, après l'exécution de la fonction, montre l'altération de `val`.

```
void f(int& x) {
  x = x + 1;
  cout << "x=" << x;
}
int main() {
  int val(1);
  f(val);
  cout << " val=" << val << endl;
  return 0;
}
```

L'exécution de ce programme produit l'affichage :
`x=2 val=2`

FIGURE 3

5:51 9:29

Exemple de passage par référence.

UTILISATION DU PASSAGE PAR RÉFÉRENCE

On utilise le passage par référence :
- lorsqu'une fonction doit être capable de modifier une variable qui lui est passée en argument ; par exemple, pour une fonction capable de saisir un entier ;
- lorsqu'une fonction retourne plusieurs résultats ; par exemple, pour une fonction qui convertit des coordonnées cartésiennes reçues comme argument en coordonnées polaires. Puisqu'en C++ une fonction ne peut retourner qu'un seul résultat, on lui fournit en paramètres deux variables passées par références qui stockent le résultat final ;
- lorsque l'on échange le contenu de deux variables.

- pour saisir une valeur :

```
void saisie_entier(int& a_lire);
...
int i(0);
...
saisie_entier(i);
```

Alternative : retourner la valeur

```
int saisie_entier();
...
i = saisie_entier();
```

- pour « retourner » plusieurs valeurs :

```
void cartesiennes_vers_polaires(double x, double y,
                                double& angle, double& rayon);
```

Alternative : utiliser les structures (futur cours)

- pour « échanger » des variables :

```
void swap(int& i, int& j);
```

FIGURE 4

6:35 9:29

Exemples d'utilisation de passage par référence.

16. FONCTIONS : PROTOTYPES

En C++, les fonctions, tout comme les variables, doivent être annoncées avant d'être utilisées : c'est ce que l'on appelle le ***prototype***. Le prototype sert à déclarer :
– le nom de la fonction ;
– les paramètres ;
– le type de la valeur de retour.

La syntaxe générale de déclaration d'un prototype de fonction est la suivante :

`type nom ( type_1 id_param_1, ..... , type_N id_param_N );`

```
double moyenne(double x, double y);
int nbHasard();
int score (double points, double temps_jeu);
double sqrt(double x);
```

FIGURE 1

0:32 5:56

Exemples de prototypes.

La liste des paramètres est éventuellement vide si la fonction n'a pas besoin de recevoir de paramètres.

La partie qui précède le point-virgule dans le prototype, est appelée « ***entête*** » de la fonction.

BONNES PRATIQUES POUR ÉCRIRE UN PROTOTYPE
Le prototype sert à annoncer au reste du programme le comportement de la fonction. C'est une sorte de contrat. Il est donc important de choisir un nom pertinent qui illustre exactement ce que fait la fonction. Mais il est impératif que la fonction ne fasse bien que ce pour quoi elle a été prévue et il ne faut absolument pas qu'elle ait des effets cachés, ce que l'on appelle des « effets de bords ». Par exemple, une fonction qui calcule la racine d'un nombre, dont le prototype dans la bibliothèque standard est `double sqrt(double x)`, calcule uniquement la racine carrée et ne doit pas polluer l'affichage avec des messages.

Il est toujours préférable de commencer par écrire le prototype de la fonction. Il permet de clarifier les paramètres que la fonction doit recevoir et le type de retour qu'elle doit fournir. On spécifie ces deux choses avant même de se préoccuper d'écrire le corps de la fonction.

DIFFÉRENTS ASPECTS DE LA SYNTAXE
On résume différents aspects de la syntaxe rencontrés jusqu'ici :
– `int a;` : la déclaration d'une variable non initialisée ;
– `int a();` : le prototype d'une fonction sans paramètres ;
– `int a(5);` : la déclaration et l'initialisation d'une variable ;
– `a(5);` : l'appel d'une fonction.

17. DÉFINITIONS

La ***définition*** d'une fonction sert à spécifier l'ensemble des instructions que la fonction exécute. Elle est constituée de l'entête et du corps de la fonction. La syntaxe de la définition d'une fonction est la suivante :

```
type nom (liste de paramètres)
{
    Instructions du corps de la fonction;
    return expression;
}
```

Le corps de la fonction est simplement un bloc d'instructions compris entre accolades. On y a accès aux paramètres de la fonction, qui sont des variables supplémentaires que l'on peut utiliser dans ce bloc comme n'importe quelle variable usuelle. Il contient aussi une ou plusieurs expressions `return`, qui mettent fin à l'exécution du corps de la fonction.

L'INSTRUCTION RETURN

L'instruction `return` suivie par une expression donne la valeur à retourner. Elle doit être de même type que le type de retour de la fonction. L'instruction `return` :
- permet de préciser la valeur de retour qui est fournie par la fonction au reste du programme ;
- termine l'exécution du corps de la fonction. Dès le premier `return` rencontré, la fonction s'arrête et retourne la valeur correspondant à l'expression.

Voici quelques remarques sur l'instruction `return` :
- On peut placer plusieurs instructions `return` dans une même fonction, par exemple dans une structure de `if else`. Un seul des `return` est exécuté pour un appel donné de la fonction.
- L'expression qui suit une instruction `return` dans une fonction doit être de même type que le type de retour indiqué dans l'entête de la fonction. Par exemple, le code `double f() { return true; }` génère une erreur car la valeur `true` n'est pas de type `double`.
- L'instruction `return` est la dernière instruction exécutée puisqu'elle met fin à l'exécution de la fonction : les instructions après `return` ne sont jamais exécutées.
- Le compilateur doit toujours pouvoir exécuter un `return`. Si le corps de la fonction se termine sans `return`, le compilateur renvoie une erreur, par exemple, si un `return` est placé dans un bloc `if`, sans d'autres `return`.

FONCTIONS PARTICULIÈRES

Il existe des fonctions qui n'ont aucune valeur de retour à fournir au reste du programme : on les appelle ***procédures***. Ces fonctions commencent par le mot réservé `void` pour indiquer qu'elles n'ont pas de type de retour. L'instruction `return;` est optionnelle dans une fonction de type `void`, mais pourrait être utile lorsque l'on doit arrêter préalablement le corps de la fonction.

On peut définir des fonctions qui n'ont besoin d'aucun paramètre de l'extérieur pour fonctionner : il suffit de mettre une liste de paramètres vide dans l'entête ; par exemple, `double saisie();`.

La fonction `main` est aussi une fonction : celle qui est appelée au début du programme. Elle n'a que deux prototypes autorisés, `int main();` et `int main(int argc, char** argv);` : dans ce cours, on n'utilise que le premier. La fonction `main` retourne un entier, typiquement un code d'erreur avec la convention que la valeur 0 indique qu'il n'y a pas d'erreur, à l'environnement dans lequel on lance le programme.

Initiation à la programmation
en C++
Jean-Cédric Chappelier,
Jamila Sam et Vincent Lepetit

18. FONCTIONS : MÉTHODOLOGIE

Voici un résumé de la méthodologie générale qu'il est conseillé de suivre pour concevoir une fonction :

- On identifie ce que la fonction doit faire. On s'intéresse à ce qu'elle doit faire et non pas à comment elle doit le faire. Il faut aussi faire attention à ne pas faire des effets de bord : la fonction doit faire exactement ce pourquoi elle a été prévue.
- On se demande quels arguments la fonction doit recevoir. Par exemple, la fonction qui calcule la moyenne de deux nombres prend deux nombres réels. Donc dans cette étape, on décide qu'elle reçoit deux arguments de type `double`.
- On décide si l'on doit passer les arguments par valeur ou par référence. Si la fonction modifie les arguments reçus, on utilise le passage par référence et sinon, le passage par valeur. On se demande optionnellement si l'on peut donner une valeur par défaut au paramètre correspondant. Ceci sera traité dans la leçon 19.
- On se demande de quel type doit être la valeur que la fonction doit retourner au reste du programme. Pour cela, il faut savoir si le code `z = f(...)` fait sens pour la fonction `f`. Si oui, le type de retour doit être le type de `z`. Sinon, le type de retour est `void`.
- Maintenant et seulement maintenant, on se préoccupe de comment écrire le corps de la fonction.

19. FONCTIONS : ARGUMENTS PAR DÉFAUT ET SURCHARGE

Dans un prototype de fonction, on peut donner des ***valeurs par défaut*** à certains des paramètres de la fonction. Dans ce cas-là, il n'est plus nécessaire de fournir d'arguments à ces paramètres lors de l'appel de la fonction.

La syntaxe d'un paramètre avec une valeur par défaut est la suivante :

```
type identificateur = valeur
```

Les paramètres avec valeur par défaut doivent forcément apparaître en dernier dans la liste des paramètres de la fonction.

```
void affiche_ligne(char elt, int nb = 5);

int main() {
  affiche_ligne('*');
  affiche_ligne('+', 8);
  return 0;
}

void affiche_ligne(char elt, int nb) {
  for(int i(0); i < nb; ++i) {
    cout << elt;
  }
  cout << endl;
}
```

Résultat :

```
*****
++++++++
```

FIGURE 1

0:58 10:25

Exemple de la fonction avec argument par défaut.

Notons que les valeurs par défaut doivent être spécifiées lors de leur déclaration dans le prototype de la fonction et non pas dans la définition de la fonction. De plus, lors de l'appel à une fonction à plusieurs paramètres ayant des valeurs par défaut, les arguments omis doivent être les derniers et omis dans l'ordre de la liste des paramètres.

```
void f(int i, char c = 'a', double x = 0.0);

 f(1)          → correct (vaut f(1, 'a', 0.0))
 f(1, 'b')     → correct (vaut f(1, 'b', 0.0))
 f(1, 3.0)     → incorrect !
 f(1, , 3.0)   → incorrect !
 f(1, 'b', 3.0) → correct
```

FIGURE 2

3:07 10:25

Exemple des appels de la fonction avec plusieurs arguments par défaut.

SURCHARGE DE FONCTIONS

En C++, il est possible de définir plusieurs fonctions qui ont le même nom si le nombre ou le type des paramètres sont différents : c'est ce que l'on appelle la ***surcharge de fonctions***. Elle est utile pour des fonctions qui font des traitements similaires, à partir de données différentes.

En C++, deux fonctions sont différenciées non seulement par leur nom, mais aussi par le type de leurs paramètres, que l'on appelle la ***signature*** de la fonction. Remarquons que l'on n'a pas le droit d'avoir deux fonctions de même nom et de mêmes paramètres même si le type de retour est différent. Par exemple, on ne peut pas avoir deux fonctions `int f(int)` et `double f(int)`.

```
void affiche(int x) {
  cout << "entier : " << x << endl;
}
void affiche(double x) {
  cout << "reel : " << x << endl;
}
void affiche(int x1, int x2) {
  cout << "couple : " << x1 << x2 << endl;
}
```

FIGURE 3

7:02 10:25

Exemple de surcharge de fonctions.

Dans l'exemple de la figure 3, on a trois fonctions différentes qui s'appellent toutes `affiche` :

– `affiche(1)` affiche un entier : 1.
– `affiche(1.0)` affiche un réel : 1.
– `affiche(1, 1)` affiche un couple : 11.

20. TABLEAUX : INTRODUCTION

Après s'être principalement concentré sur les traitements, le cours s'axe maintenant sur les données. De nouveaux types de données seront donc abordés. Les tableaux seront présentés en premier ; suivront ensuite les structures, les chaînes de caractères et enfin les pointeurs.

EXEMPLE
Imaginons que l'on crée un programme de gestion d'un jeu, qui calcule le score des joueurs ainsi que la moyenne de ces scores. Sur la base de nos connaissances, le code pour cinq joueurs ressemblerait à celui de la figure 1. On remarque que certaines lignes sont copiées-collées telles quelles. Il faut garder à l'esprit que le copier-coller n'est jamais une bonne solution. De plus, ce code est rigide, on doit le modifier pour un nombre différent de joueurs.

C'est ici tout l'intérêt des tableaux, qui regroupent les données de même type et qui peuvent avoir une taille variable. Cela simplifierait l'affichage des scores, mais aussi la définition de la fonction moyenne. Le code ressemblerait à celui de la figure 2, s'il était amélioré avec des tableaux.

```
int score1(calcule_score(....));
int score2(calcule_score(....));
int score3(calcule_score(....));
int score4(calcule_score(....));
int score5(calcule_score(....));

// Calcul de la moyenne
int moyenne_joueurs(moyenne(score1, score2, score3, score4, score5));

// Affichages
cout << "Score         Ecart Moyenne" << endl;
cout << score1 << "     " << score1 - moyenne_joueurs << endl;
cout << score2 << "     " << score2 - moyenne_joueurs << endl;
cout << score3 << "     " << score3 - moyenne_joueurs << endl;
cout << score4 << "     " << score4 - moyenne_joueurs << endl;
cout << score5 << "     " << score5 - moyenne_joueurs << endl;
```

FIGURE 1

3:14 9:34

Code avec nos connaissances actuelles.

```
unsigned int nb_joueurs(5);
vector<int, nb_joueurs> scores;

for(auto score : scores) {
  score = calcule_score(....);
}

// Calcul de la moyenne
int moyenne_joueurs(moyenne(scores));

// Affichages
cout << "Score         Ecart Moyenne" << endl;
for(auto score : scores) {
  cout << score << "     " << score - moyenne_joueurs << endl;
}
```

FIGURE 2

4:59 9:34

Code avec les tableaux.

TABLEAUX

Un ***tableau*** est, par définition, une collection de valeurs homogènes ; ce qui signifie concrètement que c'est une collection ordonnée de valeurs de même type. On peut faire des tableaux avec tous les types de données que C++ met à notre disposition. Notons que l'on peut faire des tableaux de types élémentaires (`int`, `double`, etc.), mais aussi de types de données plus complexes, et notamment de tableaux.

TYPES DE TABLEAUX

De façon générale on peut classer les tableaux en quatre catégories, selon deux critères principaux : si la taille est connue au départ (*avant* l'exécution du programme), et si la taille peut varier lors de l'utilisation. C'est au programmeur de choisir le type de tableau selon ses besoins, lors de la conception de son programme.

En C++ cependant, seuls deux types de tableaux existent, les `array` et les `vector`. Les `vector` sont des tableaux dits ***dynamiques*** et les `array` sont des tableaux dits ***statiques***. La taille d'un tableau dynamique est variable, alors que celle d'un tableau statique doit être connue au moment de la compilation. Il existe un autre type de tableau, hérité du C, l'ancêtre du C++. Ce sont les tableaux dits « à la C ». Depuis C++11, leur usage est déprécié au profit de celui des `array`.

		taille initiale connue *a priori* ?	
		non	oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	`vector`	(`vector`)
	non	(`vector`)	`array` (C++11) tableaux « à la C »

FIGURE 3

9:01 9:34

Types de tableaux.

21. TABLEAUX : DÉCLARATION ET INITIALISATION DES VECTOR

Cette leçon porte sur la manière de déclarer et d'initialiser des tableaux dynamiques. Un tableau dynamique est une collection de données homogènes, dont le nombre, c'est-à-dire la taille du tableau, peut changer au cours du déroulement du programme. En C++, les tableaux dynamiques sont implémentés au moyen du type `vector`.

DÉCLARATION

Premièrement, pour pouvoir utiliser les `vector`, il faut ajouter la ligne `#include<vector>` en début de fichier. Ensuite, la syntaxe pour déclarer un `vector` est la suivante :

```
vector<type> nom;
```

où *type* correspond au type des éléments du tableau et *nom* au nom du tableau. Le type des éléments du tableau peut être n'importe quel type C++, y compris des `vector`.

INITIALISATIONS

Depuis C++11, il existe cinq manières d'initialiser des tableaux dynamiques :

- Initialiser un tableau vide : `vector<type> nom;` Il ne faut pas mettre de parenthèses après nom.
- Initialiser un tableau avec des valeurs initiales différentes : `vector<type> nom({val1, …, valn});` On peut aussi écrire `vector<type> nom={val1, …, valn};`
- Initialiser un tableau avec une taille initiale : `vector<type> nom(taille);` les éléments du tableau seront tous nuls. Définir une taille initiale ne signifie pas que le tableau ne pourra plus changer de taille par la suite. C'est pour cela qu'il ne faut pas confondre cette syntaxe d'initialisation avec la syntaxe de déclaration des `array`, qui eux sont statiques. Cette syntaxe est : `array<type, taille> nom;`
- Initialiser un tableau avec la même valeur : `vector<type> nom(taille, valeur);` Cette syntaxe est similaire à celle ci-dessus, excepté le fait que tous les éléments du tableau ne sont pas initialisés à zéro, mais à *valeur*. Par ailleurs, *valeur* doit être du même type que *type*.
- Initialiser un tableau à l'aide d'une copie d'un autre tableau : `vector<type> nom(autre);` Avec cette méthode, chaque élément de *autre* est recopié dans notre tableau, mais on obtient bien deux tableaux distincts, *autre* n'étant pas modifié.

En C++11, il y a cinq façons d'initialiser un tableau dynamique :

- vide

```
vector<int> tab;
```

- avec un ensemble de valeurs initiales

```
vector<int> tab({ 20, 35, 26, 38, 22 });
```

- avec une taille initiale donnée et tous les éléments « nuls »

```
vector<int> tab(5);
```

- avec une taille initiale donnée et tous les éléments à une même valeur donnée

```
vector<int> tab(5, 1);
```

- avec une copie d'un autre tableau

```
vector<int> tab(tab2);
```

FIGURE 1

6:49 7:44

Résumé.

LEARN FASTER, LEARN BETTER!
BOOCs EPFL
Initiation à la programmation en C++
Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

22. TABLEAUX : UTILISATION DES VECTOR

Cette leçon traite des différentes opérations que l'on peut effectuer sur des tableaux. Ce sont des opérations élémentaires très utiles. On abordera en premier lieu la manipulation du tableau et de ses éléments, puis les itérations sur les tableaux.

AFFECTATION GLOBALE

Un tableau peut, comme toute variable, être affecté globalement. Pour un tableau, l'affectation signifie sa copie élément par élément dans un deuxième tableau. Le déroulement de l'affectation est similaire à celui des types élémentaires dans le sens où les deux tableaux restent indépendants. La figure 1 présente un exemple d'affectation globale.

```
vector<int> tab1({ 1, 2, 3 });
vector<int> tab2;
...
tab2 = tab1 ; // copie de tout tab1 dans tab2
```

FIGURE 1

0:16 15:44

Exemple d'affectation globale. Notons que la modification de `tab2` pour la suite n'induit pas de modification de `tab1`.

ACCÈS AUX ÉLÉMENTS

L'***accès à l'élément*** numéro `i+1` du tableau `tab` s'écrit `tab[i]`. Il est important de noter que la numérotation des tableaux commence à 0 en C++ ; donc pour accéder au premier élément du tableau, on écrit `tab[0]`, pour accéder au deuxième `tab[1]` et ainsi de suite. Par ailleurs, il n'y a pas de contrôle de ***débordement***. En effet, le compilateur n'affichera aucune erreur si l'on tente d'accéder à un élément dont l'indice est supérieur à la taille du tableau. Cependant, lors de l'exécution, le programme risque de produire l'erreur fatale *Segmentation Fault*.

Enfin, lorsque l'on initialise un tableau vide, sa taille est égale à zéro ; donc il ne contient même pas une case. Si l'on veut le remplir, on doit utiliser la fonction spécifique `push_back()` présentée à la leçon 24.

ITÉRATION SUR L'ENSEMBLE DES VALEURS

Depuis C++11 on peut utiliser la syntaxe

`for(auto elt: tab)`

pour parcourir le tableau `tab`. Dans cette boucle itérative, `elt` prendra tour à tour les valeurs des éléments de `tab`; c'est-à-dire qu'il prendra la valeur du premier élément, puis du deuxième et ainsi de suite jusqu'au dernier. Si l'on veut modifier les éléments de `tab` à travers `elt`, il faut ajouter une esperluette (`&`) après le `auto`. C'est, comme pour les fonctions, le symbole du passage par référence.

La figure 2 présente un exemple d'utilisation de ce type d'itération, qu'il est préférable d'utiliser dans les cas simples. Cependant, ce n'est pas adapté à toutes les situations; dans certaines on devra alors utiliser une boucle `for` classique.

```
vector<int> ages(5);

for(auto& age : ages) {
  cout << "Age de l'employé suivant ? ";
  cin >> age;
}

cout << "Age des employés : " << endl;
for(auto age : ages) {
  cout << "    " << age << endl;
}
```

FIGURE 2

6:34 15:44

Exemple d'itération avec `for(auto elt: tab)`.

ITÉRATION CLASSIQUE

L'utilisation d'une itération classique est nécessaire dans certains cas, par exemple lors du parcours simultané de plusieurs tableaux, de l'accès à plusieurs éléments ou alors lors d'un parcours plus complexe des éléments (de deux en deux, etc.). On utilise alors, par exemple, une boucle `for` comme suit:

`for(size_t i(0); i<tab.size(); ++i)`

et on accède à chaque élément par `tab[i]`. La fonction spécifique `size()` retourne la taille du tableau sous la forme d'une valeur de type `size_t` (qui est un entier positif ou nul). Bien sûr, il est possible de modifier les paramètres de la boucle `for` selon ses besoins.

```
vector<int> ages(5);

for(size_t i(0); i < ages.size(); ++i) {
  cout << "Age de l'employé " << i+1 << " ? ";
  cin >> ages[i];
}
```

FIGURE 3

12:30 15:44

Exemple d'itération sur un tableau avec une boucle `for` classique.

23. TABLEAUX : EXEMPLES SIMPLES (VECTOR)

Cette leçon aborde, au travers d'exemples simples, les utilisations de base des `vector`. On rappelle qu'il y a deux principales manières de parcourir un tableau : l'itération avec une boucle `for` classique et, depuis C++11, l'itération sur l'ensemble des valeurs. C'est la seconde qui est privilégiée, sauf si l'on doit expliciter les indices ; auquel cas la première s'impose. Pour les trois exemples suivants, les deux types de parcours sont possibles, le choix est fait par le programmeur selon ses besoins (en particulier le besoin d'expliciter les indices). Durant cette leçon, on se dote d'un tableau `tab` déclaré comme suit :

`vector<double> tab(10);` c'est-à-dire un tableau contenant dix éléments, tous nuls.

AFFICHAGE D'UN TABLEAU

La figure 1 présente deux exemples d'affichage d'un tableau. Le premier exemple traite la situation la plus simple, c'est-à-dire celle où l'on veut afficher les éléments les uns après les autres. Le deuxième illustre une situation dans laquelle on veut expliciter les indices, dans ce cas afficher l'élément et son numéro. Plusieurs choses sont à noter à propos de ces exemples. Pour le premier, étant donné que le tableau n'est pas modifié durant l'itération, le symbole du passage par référence (`&`) n'est pas nécessaire. Le deuxième exemple est plus lourd, mais plus souple. En effet, on peut parcourir de la manière que l'on souhaite, par exemple à reculons.

- si l'on n'a pas besoin d'expliciter les indices :

```
cout << "Le tableau contient : ";
for(auto element : tab) {
  cout << element << " ";
}
cout << endl;
```

- si l'on veut expliciter les indices :

```
for(size_t i(0); i < tab.size(); ++i) {
  cout << "L'élément " << i << " vaut " << tab[i] << endl;
}
```

FIGURE 1

0:14 6:53

Deux manières d'afficher un tableau.

AFFECTATION D'UN TABLEAU

La figure 2 présente deux manières d'affecter la même valeur (ici `1.2`) à tous les éléments d'un tableau. Cela peut se faire soit élément par élément, soit globalement. Dans le cas de l'affectation élément par élément, le symbole du passage par référence (`&`) est nécessaire, en effet, le tableau est modifié. Lors de l'affectation globale, on crée ce que l'on appelle un tableau anonyme, qui a la taille de tab et qui contient la valeur `1.2` dans toutes ses cases.

(ré)Affectation de tous les éléments à la valeur 1.2 :

```
for(auto& el : tab) {
  el = 1.2;
}
```

ou alors

```
tab = vector<double>(tab.size(), 1.2);
```

FIGURE 2

2:35 6:53

Deux manières d'affecter un tableau.

SAISIE AU CLAVIER

Pour la saisie au clavier de la valeur des éléments, nous choisissons à nouveau d'illustrer les deux types de boucles en figure 3. Pour le parcours sur l'ensemble des éléments, le symbole du passage par référence (`&`) est nécessaire car le tableau est modifié. On peut aussi utiliser l'itération classique pour afficher le numéro de l'élément demandé.

- si l'on n'a pas besoin d'expliciter les indices :

```
for(auto& element : tab) {
  cout << "Entrez l'élément suivant :" << endl;
  cin >> element;
}
```

- si l'on veut expliciter les indices :

```
for(size_t i(0); i < tab.size(); ++i) {
  cout << "Entrez l'élément " << i <<  ":" << endl;
  cin >> tab[i];
}
```

FIGURE 3

5:04 6:53

Deux manières de saisir les éléments du tableau au clavier.

24. TABLEAUX : FONCTIONS SPÉCIFIQUES (VECTOR)

Cette leçon présente différentes «fonctions spécifiques» (ou «méthodes» en programmation orientée objet) associées aux `vector`. Leur utilisation se fait de la façon suivante :

```
tab.fonction(arg1, arg2, …);
```

où *tab* est le nom du `vector` sur lequel la fonction spécifique `fonction()` s'applique, avec les arguments *arg1*, *arg2*… Cependant, à l'instar d'une fonction classique, une fonction spécifique peut ne pas nécessiter d'arguments. Elle a aussi généralement un type de retour. Nous présentons d'abord différentes fonctions spécifiques associées aux vector, puis un exemple de leur utilisation.

FONCTIONS SPÉCIFIQUES

Introduire l'énumération :

- *tab*`.size()` : renvoie la taille du tableau *tab*, sous la forme d'une variable de type `size_t`.
- *tab*`.front()` : renvoie une référence au premier élément de *tab*, elle est donc équivalente à *tab*`[0]`.
- *tab*`.back()` : renvoie une référence au dernier élément de *tab*, elle est donc équivalente à *tab*`[`*tab*`.size()-1]`.
- *tab*`.empty()` : renvoie un `bool` indiquant si le tableau *tab* est vide, c'est-à-dire s'il contient zéro case.
- *tab*`.clear()` : supprime tous les éléments de *tab*, le transformant donc en un tableau vide. Cette fonction spécifique ne renvoie rien.
- *tab*`.pop_back()` : supprime le dernier élément de *tab*. Cette fonction spécifique ne renvoie rien.
- *tab*`.push_back(`*val*`)` : ajoute un nouvel élément à la fin de *tab*, de valeur *val*. Cette fonction spécifique ne renvoie rien.

```
Saisie de 3 valeurs :
Entrez la valeur 0 : 5
Entrez la valeur 1 : 2
Entrez la valeur 2 : 0
Entrez la valeur 0 : 7
Entrez la valeur 1 : 2
Entrez la valeur 2 : -4
Entrez la valeur 1 : 4
Entrez la valeur 2 : 12

-> 7 4 12
```

FIGURE 1

4:08 11:50

Exemple de déroulement de la fonction `saisie()`.

EXEMPLE

On se propose d'écrire une fonction permettant à l'utilisateur de saisir le contenu d'un tableau d'entiers strictement positifs. Cette fonction prend comme arguments le tableau à remplir (passé par référence), ainsi que la taille souhaitée. La fonction saisie demande à l'utilisateur d'entrer un entier, tant que la taille souhaitée n'est pas atteinte. On fixe par ailleurs des conventions pour l'utilisation de cette fonction. Lorsque l'utilisateur saisit un nombre strictement positif, celui-ci est ajouté en fin de tableau. S'il entre zéro, le tableau est effacé, et la saisie recommence du début. Et enfin, s'il entre un nombre négatif, le dernier élément du tableau est effacé. La figure 1 montre un exemple de saisie d'un tableau à l'aide de la fonction `saisie()`.

La fonction `saisie` permet aussi de réinitialiser un tableau. Sur la figure 2, le premier appel de `saisie` permet d'initialiser `tab` à cinq éléments, puis le deuxième de le réinitialiser à quatre éléments (la valeur par défaut de la taille souhaitée). Enfin, le dernier appel montre que l'on peut réinitialiser un tableau grâce à sa taille.

```
vector<int> tab;

saisie(tab, 5); // saisie de 5 éléments

saisie(tab); // saisie de 4 éléments

vector<int> tab2(12);

saisie(tab2, tab2.size());
```

FIGURE 2

5:54 11:50

Exemple de code utilisant la fonction `saisie()`.

La figure 3 montre le code de la fonction `saisie()`. Cet exemple permet de mettre en situation cinq des sept fonctions spécifiques présentées ci-dessus. La fonction commence par vider le tableau (avec `clear()`); ensuite vient une boucle qui se répète tant que la taille du tableau est plus petite que celle passée en paramètre. Puis on demande la saisie de l'élément que l'on ajoutera à la fin du tableau (s'il est positif). Le numéro de cet élément est donné par la fonction spécifique `size()`. En effet, le numéro du dernier élément est égal à `tab.size()-1`. Enfin, on traite les différents cas, si la valeur entrée est strictement positive, on l'ajoute à la fin du tableau (avec `push_back()`), si elle est égale à 0, on vide le tableau (avec `clear()`) et si elle est strictement négative et si le tableau n'est pas vide (on le vérifie avec `empty()`), on supprime la dernière case (avec `pop_back()`). C'est ici que se finit la boucle ainsi que la fonction `saisie()`.

```
void saisie(vector<int>& vect, size_t taille = 4)
{
  vect.clear();
  cout << "Saisie de " << taille << " valeurs :" << endl;
  while (vect.size() < taille) {
    cout << "Entrez la valeur " << vect.size() << " : ";
    int val;
    cin >> val;
    if ((val < 0) and (not vect.empty())) { vect.pop_back();     }
    else if (val == 0)                    { vect.clear();        }
    else if (val > 0)                     { vect.push_back(val); }
  }
}
```

FIGURE 3

6:47 11:50

Code de la fonction `saisie()` illustrant l'utilisation de fonctions spécifiques aux `vector`.

25. TABLEAUX : TABLEAUX DYNAMIQUES MULTIDIMENSIONNELS

Cette leçon porte sur les ***tableaux dynamiques multidimensionnels***. Un tableau de dimension *n* correspond à un tableau de tableaux de dimension *n*–1. Par exemple un tableau d'entiers de dimension deux correspond à un tableau de tableaux d'entiers de dimension un (c'est-à-dire les tableaux que l'on a rencontrés jusqu'à maintenant). Par exemple, si l'on souhaite représenter les notes de tous les étudiants du MOOC à tous les devoirs, on utilisera un tableau de dimension deux, c'est-à-dire un tableau de tableaux de notes, un par étudiant.

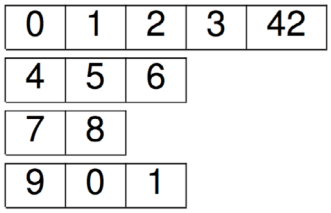


FIGURE 1

4:09 8:35

Exemple de tableau dynamique de dimension deux.

TABLEAUX DYNAMIQUES MULTIDIMENSIONNELS

La déclaration de tableaux dynamiques d'entiers de dimension deux se fait comme suit :

```
vector<vector<int>> tab(5, vector<int>(6));
```

On voit bien que `tab` est un `vector` de `vector` de `int`. `tab` est initialisé à cinq lignes, avec un tableau dynamique d'entier qui contient six cases. Comme vu à la leçon 23, on utilise un tableau anonyme pour initialiser `tab`. L'accès à la case de la ligne `i+1` et de la colonne `j+1` s'écrit `tab[i][j]`. En effet `tab[i]` retourne le tableau représentant la ligne `i+1`, et donc `tab[i][j]` est l'élément numéro `j+1` de la ligne numéro `i+1`. Il convient de noter une petite subtilité syntaxique : la syntaxe de fermeture des types, représentée par `>>`, juste avant `tab`, n'est supportée que par les compilateurs C++11, sinon il faut les séparer par une espace `> >`.

Enfin, ces tableaux étant des tableaux dynamiques, ce ne sont pas nécessairement des matrices (au sens mathématique). En effet, les lignes sont aussi représentées par des tableaux *dynamiques*, donc chaque ligne peut avoir une taille différente, comme illustré sur la figure 1.

EXEMPLE

Le code de la figure 2 initialise le tableau de dimension deux `tableau` à la valeur du tableau de la figure 1. La valeur initiale est donnée par une liste d'éléments, et chaque élément est lui-même une liste de valeurs. Puis, le code permet d'afficher, le contenu de `tableau`, à l'aide de boucles imbriquées. La première boucle énumère sur les lignes de `tableau` et la deuxième énumère sur les valeurs de la ligne. Enfin la dernière partie du code permet d'afficher la taille des lignes de `tableau`, afin d'illustrer le fait qu'elles sont bien des tableaux dynamiques. C'est une boucle `for` classique qui est utilisée, car on veut afficher les indices des lignes.

```
vector<vector<int>> tableau(
   { { 0, 1, 2, 3, 42 },
     { 4, 5, 6        },
     { 7, 8           },
     { 9, 0, 1        }  }
);

for(auto ligne : tableau) {
  for(auto element : ligne) {
    cout << element << " ";
  }
  cout << endl;
}

for(size_t i(0); i < tableau.size(); ++i) {
  cout << "tableau[" << i <<"].size()="
       << tableau[i].size() << endl;
}
```

FIGURE 2

4:58 8:35

Exemple de code utilisant le tableau dynamique de la figure 1.

LEARN FASTER, LEARN BETTER!
BOOCs EPFL
Initiation à la programmation en C++
Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

26. TABLEAUX : ARRAY

Cette leçon traite des ***tableaux statiques***, qui sont implémentés par les tableaux « à la C » ou le type `array` depuis C++11. Il est préférable d'utiliser des `array` car les tableaux « à la C » n'ont aucun avantage. La taille des `array` doit être connue à la compilation et ne peut pas changer lors de l'exécution du programme. Cette situation peut arriver assez souvent ; par exemple, si l'on souhaite modéliser des vecteurs pour un programme de géométrie dans le plan ou si l'on souhaite créer un programme de jeu d'échecs, la taille de la grille, qui sera modélisée par un tableau statique, ne variera pas.

```
const size_t taille(5);

array<int, taille> ages (
 { 20, 35, 26, 38, 22 } );
// ou :
array<int, taille> ages =
  { 20, 35, 26, 38, 22 };
```

FIGURE 1

5:23 16:04

Exemples d'initialisation.

DÉCLARATION ET INITIALISATION

Les `array` sont définis dans la bibliothèque `array`, donc pour les utiliser, il faut l'inclure à l'aide de la ligne `#include <array>`. La déclaration se fait de la manière suivante :

`array<type, taille> tab;`

où `type` est le type des éléments du tableau, `taille` la taille du tableau et `tab` le nom du tableau. Le nombre `taille` doit être connu lors de la compilation, sinon l'on doit utiliser un `vector`. La syntaxe d'initialisation d'un `array` est semblable à celle d'un `vector` à l'aide d'un autre tableau. C'est-à-dire :

`array<type, taille> tab({val`$_1$`,…,val`$_n$`});`

ou de manière équivalente :

`array<type, taille> tab={val`$_1$`,…,val`$_n$`};`

Certains compilateurs ne tolèrent que cette seconde manière d'initialiser.

Un `array` doit toujours être initialisé ; en effet, un `array` non initialisé contient des valeurs aléatoires, tout comme les types élémentaires.

UTILISATION D'UN TABLEAU DE TAILLE FIXE

Les `array` s'utilisent exactement comme les tableaux dynamiques (se référer à la leçon 22). Cependant, les fonctions spécifiques modifiant le nombre d'éléments du tableau ne sont pas disponibles (c'est-à-dire `push_back()`, `pop_back()`).

TABLEAUX STATIQUES MULTIDIMENSIONNELS

On peut, tout comme les `vector`, faire des tableaux multidimensionnels avec des `array`, ils s'utilisent de la même manière (se référer à la leçon 25), mais la taille de chaque tableau doit être spécifiée lors de la déclaration. Des exemples d'`array` multidimensionnels sont présentés sur la figure 2. Les couleurs sur la figure 2 permettent de visualiser à quel tableau s'applique la taille, dans la déclaration d'un `array` multidimensionnel. L'initialisation d'un `array` multidimensionnel diffère légèrement de celle d'un `vector` multidimensionnel, en effet, les compilateurs ne supportent actuellement qu'une seule paire d'accolades, comme sur la figure 3.

Exemples :

```
array<array<double, 2>, 2> rotation;
array<array<int, nb_statistiques>, nb_cantons> statistiques;
array<array<array<double, 4>, 2, 3> tenseur;

rotation[1][0] = 0.231;
```

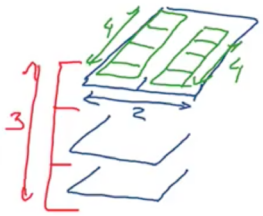


FIGURE 2

11:12 16:04

Exemples d'`array` multidimensionnels.

```
array<array<int, 3>, 4>
matrice = {
        0, 1, 2 ,
        3, 4, 5 ,
        6, 7, 8 ,
        9, 0, 1
};
```

0	1	2
3	4	5
6	7	8
9	0	1

FIGURE 3

12:40 16:04

Initialisation d'un `array` multidimensionnel.

RÉSUMÉ DE LA SEMAINE

La figure 4 fournit un résumé de ce qui a été présenté cette semaine sur les tableaux et permet aussi de visualiser les différences entre les `array` et les `vector`.

Tableaux dynamiques	Tableaux statiques
`#include <vector>`	`#include <array>`
`vector<double> tab;`	`array<double, 5> tab;`
`vector<double> tab2(5);`	
`tab[i][j]`	
`tab.size()`	
`for(auto element : tab)`	
`for(auto& element : tab)`	
`tab.push_back(x);`	—
`tab.pop_back();`	—
`vector<vector<int>> tableau( { { 0, 1, 2, 3, 42 }, { 4, 5, 6 }, { 7, 8 }, { 9, 0, 1 } } );`	`array<array<int, 3>, 4> matrice = { 0, 1, 2 , 3, 4, 5 , 6, 7, 8 , 9, 0, 1 };`

FIGURE 4

13:27 16:04

Résumé de la semaine.

27. STRING : INTRODUCTION

Nous passons maintenant à la présentation des types `string` et `char`, du mécanisme des `typedef` et enfin aux structures de données. Le type `string` permet de représenter des chaînes de caractères, le type `char` permet de représenter un seul caractère, un `typedef` permet de créer un nouveau nom pour un type déjà existant et une structure permet de regrouper des données de différents types. Cette leçon présente les types `string` et `char`.

TYPE STRING

Le type `string` permet de représenter des ***chaînes de caractères***. Une chaîne de caractères est une liste de caractères, elle peut donc contenir un mot ou une phrase. Le type `string` est défini dans la bibliothèque `string`, qu'il faut inclure à l'aide de la ligne `#include <string>` au début du programme. La déclaration d'une variable de type `string` se fait de manière usuelle :

```
string nom;
```

où `nom` est le nom de la variable. L'initialisation peut se faire dès la déclaration :

```
string nom("contenu");
```

Une variable de type `string` déclarée mais pas initialisée contient une chaîne de caractères vide. Le caractère `"` (« double quote ») délimite une chaîne de caractères, mais n'en fait pas partie.

```
#include <string>
...
// déclaration (chaîne vide)
string un_nom;

// déclaration avec initialisation
string message("Bonjour à tous !");
...
```

FIGURE 1

0:20 10:09

Exemple de déclaration et d'initialisation d'une string.

TYPE CHAR

Le type `char` est un type élémentaire qui permet de définir des variables ne contenant qu'un seul caractère. La déclaration et l'initialisation se font aussi de manière habituelle, mais un `char` est délimité par le caractère `'`, c'est-à-dire une apostrophe (« simple quote »). Il faut bien distinguer une variable de type `char` qui ne peut contenir qu'un seul caractère, et une variable de type `string` pouvant en contenir un ou plusieurs.

```
char c('x');
char u;
//...
u = 's';
```

FIGURE 2

2:00 10:09

Exemples de déclaration et d'initialisation d'un `char`.

AFFECTATION

Comme tous les autres types, une variable de type `string` ou `char` peut être modifiée par une affectation (tant qu'elle n'a pas été définie constante). La conversion de `char` vers `string` est automatique, alors que la conversion inverse n'est évidemment pas possible.

```
string chaine ;           // -> chaine vaut ""
string chaine2("test") ;  // -> chaine2 vaut "test"
chaine = "test3" ;        // -> chaine vaut "test3"
chaine = chaine2 ;        // -> chaine vaut "test"
chaine = 'a' ;            // -> chaine vaut "a"
```

FIGURE 3

4:05 10:09

Exemple d'affectations de chaînes.

CONCATÉNATION

La concaténation de deux chaînes, c'est-à-dire la mise bout à bout, se fait à l'aide de l'opérateur `+`. `chaine1 + chaine2` retourne une nouvelle chaîne dans laquelle le contenu de `chaine2` est ajouté à la fin de celui de `chaine1`. On peut non seulement concaténer deux variables de type `string`, mais on peut aussi concaténer une variable de type `string` et une valeur littérale (c'est-à-dire entre `"`), et une variable de type `string` avec une variable de type `char`. Enfin, ces concaténations peuvent se faire dans les deux sens. Dans l'exemple de la figure 4, on peut tout à fait remplacer le (`" "`) par un (`' '`), car les deux ne contiennent qu'un caractère : une espace. La concaténation avec une variable de type `char` est souvent utilisée pour ajouter un seul caractère en début ou en fin de phrase.

```
string nom;
string prenom;
string famille;
...
nom = famille + " " + prenom;
```

FIGURE 4

5:35 10:09

Exemple de concaténation.

COMPARAISON

On peut, comme avec les types élémentaires, comparer deux chaînes de caractères grâce aux opérateurs `==` et `!=`.

```
// il faut être sûr qu'ils ont compris :
do {
  reponse = poser_question();
} while (reponse != "oui");
```

FIGURE 5

9:17 10:09

Exemple d'utilisation de == sur des chaînes de caractères.

LEARN FASTER, LEARN BETTER!
BOOCs EPFL
Initiation à la programmation en C++
Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

28. STRING : TRAITEMENTS

Cette leçon présente les traitements généraux que l'on peut appliquer aux variables de type `string`.

```
string demo("ABCD");
char premier;
char dernier;

premier = demo[0];
// premier reçoit 'A'

dernier = demo[3];
// dernier reçoit 'D'
```

FIGURE 1

0:20 12:37

Exemple d'accès aux caractères d'une chaîne.

ACCÈS AUX CARACTÈRES

On peut accéder à l'élément numéro `i+1` d'une chaîne de caractères de la même manière qu'un tableau, c'est-à-dire *`chaine`*`[i]`. La figure 1 présente un exemple d'accès aux caractères d'une chaîne. La numérotation commence à zéro, comme pour les tableaux. Le caractère retourné est de type `char`. Cela signifie que l'on peut manipuler une chaîne caractère par caractère dans une boucle `for`. C'est ce qui est fait dans l'exemple de la figure 2. Il permet de former le mot « assise » à l'aide du mot « essai ».

```
string essai("essai");
string test;

for(int i(1); i <= 3; ++i) {
    test = test + essai[6-2*i];
    test = essai[i] + test;
}

cout << test << endl;
```

FIGURE 2

1:50 12:37

Exemple de parcours d'une chaîne caractère par caractère.

FONCTIONS SPÉCIFIQUES

Le type `string` dispose, tout comme le type `vector`, de certaines fonctions spécifiques permettant des manipulations assez complexes :

- *`chaine`*`.size()` : retourne la taille de la chaîne `chaine`, sous la forme d'une variable de type `size_t`.
- *`chaine`*`.insert(`*`position, chaine2`*`)` : insère la chaîne `chaine2` à la position `position` dans `chaine`. La figure 3 présente son utilisation.
- *`chaine`*`.replace(`*`position, n, chaine2`*`)` : remplace `n` caractères de la chaîne `chaine` à partir de la position `position` par `chaine2`. Par exemple, après l'exécution du code de la figure 4, exemple vaut `"a1234d"`. Par ailleurs, `replace()` peut servir à supprimer des caractères dans une chaîne. Pour cela, il suffit de passer la chaîne vide (`""`) comme troisième argument. Les `n` caractères de la chaîne à partir de `position` seront donc remplacés par la chaîne vide, c'est-à-dire supprimés.
- *`chaine`*`.find(`*`souschaine`*`)` : retourne l'indice dans `chaine` du premier caractère de l'occurrence la plus à ***gauche*** de `souschaine` dans `chaine`. Il existe aussi la fonction spécifique `rfind()` qui elle retourne l'indice dans `chaine` du premier caractère de l'occurrence la plus à ***droite*** de `souschaine` dans `chaine`. La figure 5 montre les deux situations. Dans le cas où il n'y a pas d'occurrence de `souschaine` dans `chaine`, `find()` et `rfind()` retournent la valeur prédéfinie `string::npos`.
- *`chaine`*`.substr(`*`depart, longueur`*`)` : retourne la sous-chaîne de `chaine`, de longueur `longueur`, commençant à la position `depart`. La figure 6 en présente un exemple.

```
string exemple("abcd");  // exemple vaut "abcd"
exemple.insert(1, "xx"); // exemple vaut "axxbcd"
```

FIGURE 3

5:48 12:37

Exemple d'utilisation de `insert()`.

```
string exemple("abcd");
exemple.replace(1, 2, "1234");
```

construit, dans `exemple`, la chaîne `"a1234d"`.

FIGURE 4

7:05 12:37

Exemple d'utilisation de `replace()`.

`string exemple("baabbaab");`
`exemple.find("ab")` renvoie 2.

`string exemple("baabbaab");`
`exemple.rfind("ab")` renvoie 6.

FIGURE 5

9:08 12:37

Exemple d'utilisation de `find()` et `rfind()`.

`string exemple("Salut à tous !");`
`exemple.substr(8, 4)` renvoie la `string` `"tous"`.

FIGURE 6

11:22 12:37

Exemple d'utilisation de `substr()`.

LEARN FASTER, LEARN BETTER!
BOOCs EPFL
Initiation à la programmation en C++
Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

29. TYPEDEF : ALIAS DE TYPES

Cette leçon présente les alias de types, qui en C++ sont implémentés par la mécanique des `typedef`.

```
typedef vector<double> Vecteur;

typedef vector<Vecteur> Matrice;

Matrice rotation(3, Vecteur(3, 1.0));
```

FIGURE 1

0:22 7:30

Exemple de `typedef`.

ALIAS DE TYPES

Un ***alias de type*** permet de donner un autre nom à un type déjà défini. Cela se fait de la manière suivante :

```
typedef type alias;
```

où *`type`* est le type déjà défini et *`alias`* le nouveau nom que l'on veut donner à `type`. Par exemple, la syntaxe des tableaux étant un peu lourde, en particulier pour les tableaux multidimensionnels, on peut vouloir leur donner de nouveaux noms. Sur la figure 1, on remplace le type `vector` de `double` par le type `Vecteur`, et l'on remplace le type `vector` de `Vecteur`, c'est-à-dire `vector` de `vector` de `double` par le type `Matrice`. On peut aussi bien écrire `Matrice m;` que `vector<vector<double>> m;`. Ces deux types sont tout à fait équivalents, mais on voit bien que cela améliore l'écriture, la lecture et la manipulation de ces types complexes.

RAISON DE LEUR UTILISATION

Les `typedef` sont utiles pour une raison principale, ils permettent une définition claire et conceptuelle des données. En effet, un `typedef` permet d'identifier clairement les concepts. Par exemple, il est plus logique qu'une variable représentant une distance soit de type `Distance` plutôt que `int` ou `double`. Par ailleurs, cela facilite les changements de types ultérieurs. Imaginons que l'on ait initialement représenté les distances par des `int`, puis que l'on veuille les représenter par des `double`. Sans `typedef`, il nous faudra transformer tous les `int` devant des distances par des `double`, mais il faudra aussi laisser les autres `int`, qui peuvent représenter d'autres données (par exemple des âges), intacts. Tandis qu'avec un `typedef`, seule une ligne aura à être modifiée, celle du `typedef`, précisément. D'autres raisons sont subsidiaires ; par exemple, les `typedef` permettent une écriture plus claire et compacte. Cela permet de rendre tout le code plus lisible et en particulier les prototypes des fonctions et les déclarations de tableaux.

30. STRUCTURES

En programmation, il est très fréquent d'avoir à représenter des ***données structurées***. Il y en a deux types, les données structurées ***homogènes*** et les données structurées ***hétérogènes***. Des données structurées sont homogènes lorsque les éléments qui les composent sont tous de même type, elles sont hétérogènes lorsque les éléments qui les composent sont de différents types. Par exemple, les tableaux, qu'ils soient statiques ou dynamiques, sont homogènes. Ils sont très utiles pour représenter, par exemple une liste d'âges ; mais ils ne sont pas adaptés pour représenter des données hétérogènes. Par exemple, une personne, dont les caractéristiques sont le nom, la taille l'âge et le sexe, ne peut pas être représentée par un tableau. En effet, les éléments composants cette personne sont de types différents ; c'est pourquoi on utilise pour la représenter un nouveau type : une ***structure***.

UTILISATIONS DES STRUCTURES

Les structures peuvent avoir différentes utilisations :

- Elles servent à représenter des entités qui doivent être décrites par plusieurs données, et qui peuvent de ce fait être manipulées comme un tout. La figure 1 présente trois exemples de structures, une date, définie par un jour, un mois, une année ; un étudiant, défini par son nom, sa section, la liste des cours auxquels il est inscrit et sa moyenne ; une particule, définie par sa position, sa vitesse, toutes deux représentées par un tableau à trois éléments, sa masse et sa charge.
- Elles peuvent permettre à une fonction de retourner plusieurs valeurs.
- Elles simplifient la conception et l'écriture des programmes, en regroupant les données de manière conceptuelle.

Les structures sont particulièrement intéressantes lorsque l'on a besoin de tableaux regroupant des données hétérogènes. On utilise alors des tableaux de structures.

```
struct Date
{
  int jour;
  int mois;
  int annee;
};
```

```
struct Etudiant
{
  string nom;
  string section;
  vector<Cours> inscriptions;
  double moyenne;
};
```

```
struct Particule
{
  array<double, 3> position;
  array<double, 3> vitesse;
  double masse;
  double charge;
};
```

FIGURE 1

3:02 24:18

Exemple de structures.

```
struct Simple {
  int    souschamp1;
  double souschamp2;
};

struct Compliquee {
  vector<double> champ1;
  int            champ2;
  Simple         champ3;
};
```

FIGURE 2

6:15 24:18

Exemple de structure plus complexe.

DÉCLARATION

La déclaration d'un nouveau type «structure» se fait de la manière suivante :

```
struct Nom_du_type {
       type_1 identificateur_1;
       type_2 identificateur_2;
       ...
};
```

où *Nom_du_type* est le nom donné à la structure et `type_i identificateur_i` les déclarations des types et identificateurs des ***champs*** de la structure. Les champs sont les données regroupées dans la structure. Il ne faut pas oublier le point-virgule après l'accolade fermante.

Les champs des structures peuvent être des types complexes, c'est-à-dire des tableaux ou même des structures. Par exemple, sur la figure 2, le champ `champ3` de la structure `Compliquee` est de type `Simple`, qui est une structure. Étant donné qu'une structure définit un nouveau type, on peut déclarer des variables comme tout autre type :

```
Nom_du_type nom_de_la_variable;
```

INITIALISATION

Une variable de type structure peut être initialisée comme suit :

```
Type identificateur={val_1, val_2, …};
```

où chaque `val_i` est de type `type_i` correspondant au champ de position `i`. En C++11, on peut aussi utiliser cette syntaxe pour l'affectation. Auparavant, il fallait affecter chaque champ un par un. Pour accéder au champ `champ` de la variable de type structure `structure`, on utilise la syntaxe suivante : `structure.champ`. Par exemple, si l'on a déclaré `d` de type `Date` (comme dans la figure 1), on peut modifier le champ jour comme ceci `d.jour=30;`. L'opérateur `++` étant prioritaire sur l'opérateur `.`, il faut écrire `++(structure.champ);` plutôt que `++structure.champ;` si l'on veut incrémenter `champ`.

EXEMPLE

On se propose d'écrire un programme permettant de faire évoluer une `Personne`. La figure 3 présente le `main()` du programme ainsi que la déclaration de la structure `Personne`. Une `Personne` est définie par son nom, sa taille, son âge et son sexe. Le `main()` débute par l'initialisation d'une `Personne` à l'aide de la fonction `naissance()`, présentée sur la figure 4. Cette fonction est chargée de demander les caractéristiques d'une personne et de les retourner sous la forme d'une `Personne`. Elle commence par créer une `Personne`, puis demande à l'utilisateur d'initialiser ses champs un à un et enfin elle la retourne. Donc, dans le `main()`, la première ligne permet d'initialiser `untel`. Ensuite, est appelée la fonction `anniversaire()`, qui permet d'incrémenter l'âge de la `Personne` passée en argument. Pour que la `Personne` puisse être modifiée, elle doit être passée par référence. Puis, on affiche `untel` à l'aide de la fonction `affiche()`.

```
struct Personne {
  string nom;
  double taille;
  int    age;
  char   sexe;
};

int main()
{
  Personne untel( naissance() );

  anniversaire(untel);
  // un an de plus

  affiche(untel);
  cout << endl;

  return 0;
}
```

FIGURE 3

12:00 24:18

Déclaration de `Personne` et `main()`.

Une structure étant généralement une variable de grosse taille, il est préférable, lorsqu'elle est en argument d'une fonction, de la passer par référence ou référence constante si elle n'est pas modifiée. Cela permet d'éviter d'effectuer une copie coûteuse, ce qui aurait été le cas avec un passage par valeur. La fonction `affiche()` affiche simplement les champs un à un. Après l'appel à `affiche()`, le `main()` se termine.

```
Personne naissance() {
  Personne p;

  cout << "Saisie d'une nouvelle personne" << endl;
  cout << "  Entrez son nom : ";
  cin >> p.nom;
  cout << "  Entrez sa taille (m) : ";
  cin >> p.taille;
  cout << "  Entrez son age : ";
  cin >> p.age;
  do {
    cout << "  Homme [M] ou Femme [F] : ";
    cin >> p.sexe;
  } while ((p.sexe != 'F') and (p.sexe != 'M'));

  return p;
}
```

FIGURE 4

12:28 24:18

Fonction `naissance()`.

```
void anniversaire(Personne& p) {
  ++(p.age);
}

void affiche(Personne const& p) {
  cout << p.nom    << ", ";
  switch (p.sexe) {
    case 'M': cout << "homme"; break;
    case 'F': cout << "femme"; break;
    default : cout << "alien"; break;
  }
  cout << ", "
       << p.taille << " m, "
       << p.age    << " an";
  if (p.age > 1) {
    cout << 's';
  }
}
```

```
int main()
{
  Personne untel( naissance() );

  anniversaire(untel); // un an de plus

  affiche(untel);
  cout << endl;

  return 0;
}
```

FIGURE 5

15:00 24:18

Fonctions `anniversaire()` et `affiche()`.

MANIPULATIONS SUR LES STRUCTURES

L'affectation est la seule opération que l'on peut faire globalement sur les structures, toutes les autres opérations (comparaison, affichage) doivent se faire champ par champ. L'affectation globale est équivalente à une affectation champ par champ. Pour effectuer les autres opérations, l'idéal est de coder des fonctions.

FONCTIONS À PLUSIEURS VALEURS DE RETOUR

Les fonctions ne peuvent retourner qu'une seule valeur. C'est pourquoi l'on a recourt à différentes « astuces » lorsque l'on souhaite retourner plusieurs valeurs :

1. Renvoyer une structure contenant les valeurs à retourner.
2. Passer les « variables de retour » par référence et les affecter dans la fonction.
3. Renvoyer un tableau dynamique si les valeurs à retourner sont toutes de même type.
4. Combiner 1 et 3, c'est-à-dire renvoyer des tableaux de structures ou des structures contenant des tableaux.

Ces « astuces » sont illustrées par le cas de la division euclidienne sur la figure 6. En effet, la division euclidienne retourne deux entiers, le quotient et le reste. Trois prototypes de cette fonction sont donc présentés, correspondant aux trois premières situations ci-dessus.

1.
```
struct Resultat {
  int quotient;
  int reste;
};
Resultat division_euclidienne(int dividende, int diviseur);
```

2.
```
void division_euclidienne(int dividende, int diviseur,
                          int& quotien, int& reste);
```

3.
```
array<int, 2> division_euclidienne(int dividende, int diviseur);
```

ou

```
vector<int> division_euclidienne(int dividende, int diviseur);
```

FIGURE 6

22:22 24:18

Exemples de ces « astuces ».

31. POINTEURS ET RÉFÉRENCES : INTRODUCTION

Cette leçon est une introduction aux ***pointeurs*** et aux ***références***. On y désigne, sous le terme générique « pointeur », les références et les vrais pointeurs. Les références seront présentées plus spécifiquement dans la leçon 32 et les pointeurs dans les leçons 33-36. Les pointeurs ont la réputation d'être difficiles, mais il n'y a pas de raison. En effet, un ***pointeur*** est tout simplement une ***adresse***. Par exemple dans un navigateur Internet, on ne stocke pas tous les sites que l'on souhaite visiter, mais on stocke leurs adresses, sous forme de signets, afin de pouvoir y accéder plus tard. Un signet permet de créer un lien vers un site Internet. On utilise les pointeurs de la même manière dans un programme, ils permettent de créer un ***lien***, une ***référence universelle*** vers une variable.

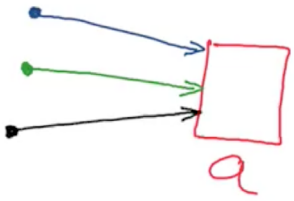


FIGURE 1

2:20 9:56

Référence.

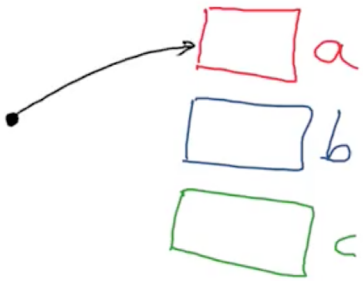


FIGURE 2

3:05 9:56

Généricité..

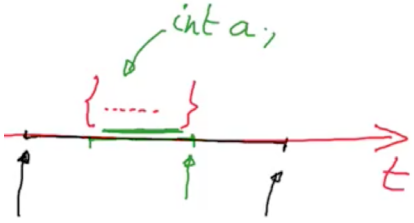


FIGURE 3

5:25 9:56

Durée de vie d'une variable et allocation dynamique...

UTILISATIONS DES POINTEURS

Il y a trois situations typiques d'utilisation des pointeurs :

- ***Référence*** : pouvoir partager des données sans les dupliquer. Par exemple, dans le cas d'un site Internet, plusieurs signets, sur plusieurs navigateurs différents permettent de partager le même contenu, le site, sans le dupliquer. Sur la figure 1, le carré représente le contenu partagé et les flèches les pointeurs qui y accèdent.
- ***Généricité*** : pouvoir choisir des éléments non connus *a priori*. Cette situation est un peu symétrique par rapport à la situation précédente. En effet, au lieu d'avoir plusieurs pointeurs pointant sur une même donnée, on a un seul pointeur qui pointe tour à tour sur plusieurs données. Par exemple, si dans un carnet d'adresses on note le numéro de téléphone d'un ami, puis celui-ci change de numéro par la suite. On a bien un seul pointeur, une seule adresse, qui pointe vers des données différentes. Sur la figure 2, les carrés représentent des données et la flèche un pointeur.
- ***Allocation dynamique*** : pouvoir manipuler des objets dont la durée de vie dépasse la portée dans laquelle ils ont été définis. La ***portée*** d'une variable est l'ensemble des lignes de code dans lesquelles cette variable est définie. La ***durée de vie*** est le temps pendant lequel un objet existe en mémoire. L'allocation dynamique permet à une variable de survivre à sa portée, c'est-à-dire de ne pas être supprimée alors que sa portée prend fin. Par exemple sur la figure 3, la partie verte de l'axe correspond à la durée de vie de la variable a sans allocation dynamique, et la partie noire à ce qu'elle pourrait être avec allocation dynamique. La leçon 34 sera consacrée à ce sujet.

Lors de l'utilisation de pointeurs, il faut toujours garder à l'esprit pour quelle(s) raison(s) on les utilise.

TYPES DE « POINTEURS »

Depuis C++11, il existe trois sortes de « pointeurs » :

- Les ***références*** : elles ne sont pas de vrais pointeurs, mais elles permettent tout de même de créer un lien vers une variable. Elles sont très sûres car elles sont totalement gérées par le compilateur.
- Les ***pointeurs intelligents*** : ils sont aussi appelés *smart pointers*, et sont disponibles depuis C++11. Ils sont utilisés pour l'allocation dynamique et doivent être gérés par le programmeur, mais fournissent des garde-fous. Il en existe trois : `unique_ptr`, `shared_ptr`, `weak_ptr`, définis dans la bibliothèque `memory`.
- Les ***pointeurs « à la C »*** : ce sont les pointeurs les plus puissants, ils permettent de tout faire, mais ce sont aussi les plus dangereux, car tout doit être géré par le programmeur.

En règle générale, il vaut mieux utiliser des références lorsqu'on le peut et utiliser des pointeurs lorsqu'on le doit. Par ailleurs certains types de « pointeurs » sont mieux adaptés aux cas d'utilisation décrits ci-dessus :

- ***Référence*** : les références, bien sûr, mais aussi les pointeurs « à la C » (qui sont utilisables dans tous les cas).
- ***Généricité*** : les pointeurs « à la C », ou les index d'un tableau si les données peuvent être regroupées dans un tableau.
- ***Allocation dynamique*** : les pointeurs intelligents, en particulier `unique_ptr`, ou les pointeurs « à la C ».

32. RÉFÉRENCES

Une ***référence*** est un ***alias***, un ***autre nom*** pour une variable existant déjà. Une référence permet de désigner indirectement un objet. C'est ce qui est utilisé lors d'un passage par référence, dans le cadre de l'appel d'une fonction, qui permet de désigner la même variable par des « étiquettes » différentes, une hors de la fonction, et une dans la fonction. Une référence se déclare de la manière suivante :

```
type& nom_reference(identificateur);
```

où `nom_reference` est le nom de la référence sur *`identificateur`*. Ce n'est pas une nouvelle variable, mais une nouvelle « étiquette », un nouveau nom, au travers duquel on peut accéder à la variable `identificateur`. Après cette déclaration, la référence peut être utilisée partout où la variable peut l'être et de la même façon.

PIÈGES À ÉVITER

Il convient de distinguer la déclaration d'une référence sur une variable et l'initialisation d'une variable à l'aide de la copie d'une autre. Par exemple sur la figure 1, à gauche, `j` est une référence sur `i`, mais à droite, `j` est une autre variable, qui a été initialisée avec la valeur de `i`, et qui est indépendante de `i`.

```
int i(3);
int& j(i); // alias
/* i et j sont la MÊME *
 * case mémoire        */
i = 4; // j AUSSI vaut 4
j = 6; // i AUSSI vaut 6
```

```
int i(3);
int j(i); // copie
/* i et j vivent leur *
 * vie séparément     */
i = 4; // j vaut encore 3
j = 6; // i vaut encore 4
```

FIGURE 1

3:02 12:04

Signification de l'opérateur =.

Il est possible de définir une `const`-référence sur une variable qui elle n'est pas constante. Cela signifie que l'on pourra modifier le contenu de la variable au travers de la variable mais pas au travers de la référence, comme illustré sur la figure 2.

```
int i(3);
const int& j(i); /* i et j sont les mêmes.                *
                  * On ne peut pas changer la valeur VIA J *
                  * (mais on peut le faire par ailleurs).  */
j = 12; // NON
i = 12; // OUI, et j AUSSI vaut 12 !
```

FIGURE 2

5:30 12:04

`const`-référence.

SPÉCIFICITÉS

Les références permettent tout comme les pointeurs de créer des liens vers des données, cependant, ce ne sont pas des pointeurs. Elles sont moins souples que ceux-ci et doivent obéir à certaines ***règles***.

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

FIGURE 3

7:15 12:04

Spécificité 1.

Premièrement, une référence doit absolument être initialisée. En effet, on ne peut pas déclarer de référence puis la lier plus tard à un autre objet, comme illustré sur la figure 3.

```
int i;
int& ri(i);
int j(2);
ri = j;  /* ne veut pas dire que ri est maintenant un alias de j, *
          * mais que i prend la valeur de j !!                    */
j = 3;
cout << i << endl; // affiche 2
```

FIGURE 4

7:55 12:04

Spécificité 2.

Deuxièmement, on ne peut pas modifier la variable sur laquelle une référence est liée. C'est pour cela que la généricité est impossible avec des références. La figure 4 illustre cette caractéristique.

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

FIGURE 5

10:12 12:04

Spécificité 3.

Troisièmement, on ne peut pas référencer une référence, car une référence n'est pas un objet en mémoire, mais une étiquette, comme illustré sur la figure 5. Enfin, on ne peut pas faire de tableaux de références.

33. POINTEURS : CONCEPT ET ANALOGIE

CONCEPT

La différence fondamentale entre une ***référence*** et un ***pointeur*** réside dans le fait qu'un pointeur n'est pas seulement une étiquette ; il est en fait une variable à part entière qui contient l'***adresse en mémoire*** de la variable pointée. En effet, toute variable (y compris les pointeurs) est physiquement identifiée de manière unique à l'aide d'une adresse, qui correspond à l'adresse de l'emplacement mémoire qui contient sa valeur. On peut déjà remarquer qu'un pointeur contient un niveau de plus qu'une référence, il est une variable à part entière. C'est pour cela que l'on peut faire des pointeurs de pointeurs ainsi que des tableaux de pointeurs.

ANALOGIE

Un pointeur est donc semblable à une page d'un carnet d'adresses, où l'on ne peut écrire qu'une seule adresse. On va s'aider de cette analogie pour illustrer un certain nombre d'utilisations concrètes de pointeurs en C++. Dans ces exemples, la syntaxe de C++ sera utilisée, même si elle n'est présentée en détail qu'à la leçon 34.

- Déclarer un pointeur : `int* ptr;` c'est équivalent à ajouter une page à notre carnet d'adresses, cependant, comme les types élémentaires, un pointeur non initialisé peut contenir n'importe quelle adresse.
- Affecter un pointeur : `ptr = &x;` c'est équivalent à écrire une adresse sur la page du carnet d'adresses, c'est-à-dire stocker l'adresse de `x` dans `ptr`.
- Allouer un pointeur : `ptr = new int(42);` c'est équivalent à aller acheter un terrain, y construire une maison et noter son adresse sur la page du carnet d'adresses, c'est-à-dire réserver une zone mémoire et stocker son adresse dans le pointeur. Il ne faut pas confondre le pointeur et la valeur pointée, c'est-à-dire l'adresse de la maison et la maison elle-même.
- Libérer un pointeur : `delete ptr;` c'est équivalent à vendre le terrain que l'on avait acheté, mais sans supprimer son adresse dans la page du carnet d'adresses, c'est-à-dire libérer la zone mémoire contenant la valeur pointée par `ptr`.
- Annuler un pointeur : `ptr = nullptr;` c'est équivalent à effacer l'adresse contenue dans la page du carnet d'adresses, mais cela ne s'accompagne pas de la vente du terrain. Cela correspond à la remise à zéro du pointeur. Les opérations de libération et d'annulation vont généralement de pair, car dans le cas contraire, on peut aboutir à deux situations : soit on ne peut plus retrouver la maison car on en a perdu l'adresse, mais elle nous appartient toujours, soit on a encore l'adresse d'une maison qui n'est plus la nôtre.
- Copier un pointeur : `p2 = p1;` c'est équivalent à donner l'adresse de sa maison à un ami pour qu'il l'écrive sur une page de son carnet d'adresses.

La leçon 34 explique comment effectuer ces actions en C++.

34. POINTEURS : DÉCLARATION ET OPÉRATEURS DE BASE

```
int* ptr(nullptr);

int* ptr(&i);

int* ptr(new int(33));
```

FIGURE 1

0:17 10:24

Différents exemples d'initialisation d'un pointeur.

```
int x(3);
int* px(nullptr);

px = &x;
```

FIGURE 2

3:20 10:24

Différents exemples d'initialisation d'un pointeur.

DÉCLARATION

Un pointeur ***se déclare*** de la manière suivante :

```
type* identificateur;
```

où `identificateur` est le nom de ce pointeur et `type` le type de la valeur sur laquelle `identificateur` pourra pointer. Par exemple, un pointeur sur un entier se déclare comme ceci :

```
int* ptr;
```

Un pointeur ***s'initialise*** de la manière suivante :

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

où `adresse` est l'adresse en mémoire de la valeur sur laquelle pointe ce pointeur. Des exemples d'initialisations sont présentés sur la figure 1. À la première ligne, `ptr` est initialisé à `nullptr`, ce qui signifie que `ptr` ne pointe sur rien. Pour reprendre l'analogie du carnet d'adresses, `nullptr` est équivalent à une page vide. La valeur spéciale `nullptr` peut être affectée à tout type de pointeurs, afin d'indiquer qu'un pointeur ne pointe sur rien. À la deuxième ligne, `ptr` est initialisé à l'adresse mémoire de la valeur d'une variable `i` de type `int`. La syntaxe `&i` retourne en effet l'adresse en mémoire de `i`. Enfin, à la troisième ligne, `ptr` est initialisé avec l'adresse d'une zone mémoire allouée dynamiquement. L'allocation dynamique est présentée en détail dans la leçon 35.

OPÉRATEURS SUR LES POINTEURS

C++ met à disposition deux opérateurs pour manipuler des pointeurs : l'***opérateur*** `&` et l'***opérateur*** `*`.

L'opérateur `&` permet de retourner l'***adresse mémoire*** de la valeur d'une variable. Il se place avant la variable à laquelle il s'applique. Par exemple, `&i` retourne l'adresse en mémoire de `i`, si `i` est de type `type`, alors `&i` est de type `type*`, c'est-à-dire pointeur sur `type`. La figure 2 est un petit programme illustrant l'utilisation de cet opérateur. La première ligne permet d'initialiser une variable `x` de type entier à la valeur 3. Puis, le pointeur sur un `int ptr` est initialisé à `nullptr`, c'est-à-dire que pour l'instant il ne pointe vers rien. Enfin, l'adresse de `x` est affectée à `ptr` à l'aide de l'opérateur `&`. C'est cette affectation qui permet de créer le lien entre le pointeur et la zone pointée.

L'opérateur `*` permet de retourner la ***valeur pointée*** par un pointeur. Il se place aussi avant le pointeur auquel il s'applique. Par exemple, `*ptr` retourne la valeur de la zone mémoire sur laquelle `ptr` pointe, si `ptr` est de type `type*`, alors (`*px`) est de type `type`. Dans l'exemple de la figure 3, on retrouve les instructions de la figure 2, permettant à `ptr` de pointer sur `x`. La dernière ligne permet d'afficher la valeur de `x` à travers `ptr`, à l'aide de l'opérateur `*`.

```
int x(3);          // x est de type entier
int* px(nullptr); // px est un pointeur sur entier

px = &x;       // px pointe sur la variable x
cout << *px    // affiche la valeur pointee par px : 3
     << endl;
```

FIGURE 3

5:35 10:24

Exemple d'utilisation de l'opérateur `*`.

`*&i` est strictement équivalent à `i`, en effet, si `i` est de type `type`, `&i` retourne un `type*`, c'est-à-dire un pointeur sur un `type`. On peut appliquer l'opérateur `*` à ce pointeur, donc `*&i` retourne la valeur de `i`.

CONFUSIONS POSSIBLES AVEC * ET &

En C++ les symboles `&` et `*` peuvent être utilisés pour deux choses différentes. Lorsqu'ils sont placés avant une variable ou un pointeur, ils permettent de retourner l'adresse de la variable ou la valeur pointée par le pointeur, respectivement. Lorsqu'ils sont placés après un type, ils définissent une référence sur ce type ou un pointeur sur ce type, respectivement. Par exemple, `&x` retourne l'adresse de `x`, alors que `int& x(y);` déclare `x` comme une référence sur `y`. `*ptr` retourne la valeur pointée par `ptr` et `int* ptr;` déclare `ptr` comme un pointeur sur un `int`.

Initiation à la programmation en C++
Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

35. POINTEURS : ALLOCATION DYNAMIQUE

Cette leçon porte sur l'allocation dynamique, qui permet de manipuler des objets dont la durée de vie dépasse leur portée.

ALLOCATIONS DE MÉMOIRE

En C++, il y a deux types d'allocations, l'***allocation statique*** et l'***allocation dynamique***. Lors de la déclaration d'une variable, c'est une allocation statique qui a lieu. Une allocation est dite statique lorsque le besoin en mémoire est connu dès la compilation. Par exemple, si dans un programme on déclare un entier, on sait déjà lors de la compilation que l'on aura besoin d'une zone mémoire contenant un entier. Une allocation est dite dynamique lorsque le besoin en mémoire n'est connu qu'à l'exécution. Par exemple, lors de l'utilisation d'un tableau dynamique, c'est au moment de l'exécution d'une ligne de code contenant un `push_back()` que l'allocation a lieu. En effet, le besoin en mémoire ne peut pas être connu à la compilation car cette ligne peut ne pas être exécutée.

ALLOCATION AVEC POINTEURS

Dans le cas des pointeurs, l'allocation dynamique permet de ***réserver*** une zone mémoire ***indépendamment de toute variable***. Le pointeur pointera alors directement sur la zone mémoire plutôt que sur une variable. Ce sont les opérateurs `new` et `delete` qui permettent d'allouer et de libérer de la mémoire, respectivement. L'opérateur `new` s'utilise comme suit :

```
ptr = new type;
```

où `ptr` est le pointeur dans lequel sera stockée l'adresse de la zone mémoire de type `type` dynamiquement allouée. Par exemple, le code suivant :

```
int* ptr;
ptr = new int;
```

alloue statiquement un pointeur sur un entier, mais la zone mémoire liée à ce pointeur est elle allouée dynamiquement. Cependant, comme lors de la déclaration d'une variable de type `int`, celle-ci n'est pas initialisée. Pour ce faire, on procède ainsi :

```
ptr = new type(valeur);
```

où `valeur` est la valeur d'initialisation.

LIBÉRATION

Dans le cas d'une allocation statique, une variable ne peut pas survivre à sa portée, c'est-à-dire qu'elle est automatiquement libérée à la fin de celle-ci. Ce n'est pas le cas pour une allocation dynamique, c'est ce qui permet à une variable de survivre à sa portée. Cependant, lorsque les données qui ont été allouées dynamiquement ne sont plus nécessaires dans la suite du déroulement du programme, il convient de les ***libérer***, afin de ménager les ressources de l'ordinateur. C'est l'opérateur `delete` qui permet de libérer une zone mémoire et il s'utilise de la façon suivante :

```
delete ptr;
```

où `ptr` est le pointeur contenant l'adresse de la zone mémoire à libérer. Après l'exécution de cette ligne, on ne doit plus accéder à la zone mémoire dont l'adresse est dans `ptr`.

BONNES PRATIQUES

Si l'on tente d'accéder à une zone mémoire qui a été désallouée, on court le risque de voir le programme s'arrêter subitement à cause d'un *Segmentation Fault*. Pour reprendre l'analogie du carnet d'adresses et des terrains, ce serait comme pénétrer dans un terrain dont on n'est plus propriétaire. C'est pourquoi il est fortement conseillé de faire suivre tous les `delete` de l'instruction `ptr=nullptr;` ce qui permettra de mettre en place des garde-fous afin d'empêcher au programme d'accéder à une zone mémoire dont il n'est plus propriétaire. Cependant, il faut veiller à ne pas affecter la valeur `nullptr` à un pointeur avant que celui-ci n'ait été libéré. La zone mémoire ne pourrait alors plus être atteinte, ni même supprimée. Cela s'appelle une ***fuite de mémoire*** et doit être à tout prix évité. Par ailleurs, il est aussi fortement conseillé de libérer avec l'instruction `delete` toute zone mémoire qui a été allouée à l'aide d'un `new`. En effet, la libération n'étant pas automatique, c'est au programmeur de la faire.

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

FIGURE 1

13:50 18:42

Exemple d'utilisation des pointeurs avec allocation dynamique.

EXEMPLE

La première ligne de cet exemple alloue statiquement un pointeur sur un entier, initialisé à `nullptr`. Lors de la deuxième ligne a lieu l'allocation dynamique d'un entier, l'adresse de la zone mémoire allouée est stockée dans `px`. Ensuite, le contenu sur lequel pointe `px` est initialisé à la valeur `20`, grâce à l'opérateur `*`. Il est possible de remplacer ces trois lignes par ce code, plus compact :

```
int* px(new int(20));
```

Puis, le contenu est affiché, à l'aide de l'opérateur `*`. Enfin, la zone mémoire est libérée grâce à l'opérateur `delete`, car le contenu sur lequel pointe `px` n'est plus utilisé dans la suite du programme. La zone mémoire ayant été libérée, la valeur `nullptr` est affectée à `px`.

SEGMENTATION FAULT

Lors de l'exécution d'un programme, il est possible qu'il s'arrête abruptement à cause de l'erreur *Segmentation Fault*. Cette erreur se produit typiquement lorsque l'on tente d'accéder via un pointeur à une zone mémoire qui n'a pas été allouée. Par exemple, sur la figure 2, le pointeur `px` est déclaré mais n'est pas initialisé, il contient donc une adresse aléatoire. À la deuxième ligne, on tente d'accéder au contenu de `px` alors qu'il est inconnu. Il n'y aura pas d'erreur à la compilation, cependant il y en aura une à l'exécution. Pour résoudre ce problème, il convient d'allouer dynamiquement un emplacement mémoire et de stocker son adresse dans `px`. Il faut toujours initialiser un pointeur. C'est la valeur `nullptr` qui doit être utilisée si la zone mémoire pointée n'est pas encore connue à l'initialisation. Grâce à cette précaution, il est possible de mettre en place des garde-fous. En effet, on peut avant d'accéder au contenu pointé, vérifier si le pointeur ne contient pas `nulltptr`, ce qui signifierait qu'il ne pointe vers rien.

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

FIGURE 2

16:26 18:42

Exemple de programme générant un *Segmentation Fault*.

36. POINTEURS « INTELLIGENTS »

Les pointeurs présentés jusqu'à présent sont les pointeurs « à la C », ils requièrent une grande rigueur à l'utilisation, en particulier pour les allocations et les libérations de mémoire. Depuis C++11, d'autres types de pointeurs sont disponibles dans le cas d'une allocation dynamique, ce sont les ***pointeurs intelligents*** ou *smart pointers*. Leur utilisation est plus confortable que celle des pointeurs « à la C ». En effet, ils libèrent automatiquement la mémoire, au moment opportun.

TYPES DE POINTEURS INTELLIGENTS

Ces pointeurs sont définis dans la bibliothèque `memory`, qu'il faut inclure à l'aide de la ligne `#include <memory>`. Ils facilitent la ***gestion de la mémoire***, grâce à un mécanisme de ramasse-miettes ou *garbage collecting*. C'est-à-dire qu'ils effectuent leur propre ***libération*** au moment adéquat. Il existe depuis C++11 trois types de pointeurs intelligents :

– `unique_ptr`
– `shared_ptr`
– `weak_ptr`

La principale différence est le nombre de pointeurs de même type qui peuvent référencer une même zone mémoire. Il ne peut y avoir qu'***un seul*** `unique_ptr` pointant vers une zone mémoire alors qu'il peut y avoir ***plusieurs*** `shared_ptr` ou `weak_ptr` pointant vers la même zone mémoire.

UNIQUE_PTR

Dans le cas des pointeurs « à la C », il n'y a pas de restriction quant au nombre de pointeurs pouvant pointer sur la même zone mémoire. Cela implique une grande rigueur de la part du programmeur. Par exemple, si deux pointeurs pointent vers une même zone mémoire et que l'un d'eux libère cette zone, alors il faut être sûr que l'autre n'aura plus besoin de celle-ci. L'utilisation de `unique_ptr` permet de garantir que le propriétaire d'une zone mémoire est unique, c'est-à-dire qu'il ne peut y avoir qu'un seul `unique_ptr` qui pointe sur une zone mémoire. Ceci permet d'éviter les problèmes mentionnés ci-dessus, dans le cas où la zone mémoire n'a pas à être partagée. Un `unique_ptr` ne peut évidemment pas être copié, mais il peut être « déplacé », grâce à la *move semantic*.

Un `unique_ptr` se déclare comme ceci :

```
unique_ptr<type> identificateur(new type(valeur));
```

où `identificateur` est le nom du `unique_ptr`, `type` le type sur lequel il pointe et `valeur` la valeur d'initialisation. Il est possible de libérer un `unique_ptr` avant que cela ne soit fait automatiquement grâce à la fonction spécifique `reset()`. Après l'appel à cette fonction, l'adresse stockée dans le `unique_ptr` vaut `nullptr`.

EXEMPLES

Le premier exemple (fig. 1) présente une utilisation basique d'un `unique_ptr`, on peut voir que l'opérateur `*` s'utilise comme dans le cas d'un pointeur « à la C ». Le deuxième exemple (fig. 2) montre qu'il est possible de déclarer un tableau de `unique_ptr`, qui contient les adresses de zones mémoire allouées dynamiquement. Enfin, le dernier exemple (fig. 3) illustre la notion de déplacement. La fonction `naissance()` permet d'allouer une zone mémoire contenant une personne, de l'initialiser et de retourner un `unique_ptr` pointant dessus. Cependant, dans une situation classique, à la dernière ligne, il y aurait une copie du `unique_ptr bb` vers le `unique_ptr adresse_quidam`, puis cette copie effectuée, `bb` serait supprimé. En réalité, c'est un déplacement qui est effectué, de manière implicite. En effet, la valeur qui est dans `bb` est déplacée dans `adresse_quidam`. La variable `bb` n'est plus utilisable en tant que telle. Le compilateur C++11 fait cela implicitement car ce qui est affecté à `adresse_quidam` est en fait une variable transitoire résultant de l'appel d'une fonction.

```cpp
#include <memory>
// ...
unique_ptr<int> px(new int(20));
// ...
cout << *px << endl;
```

FIGURE 1

4:35 12:28

Exemple 1 d'utilisation d'un `unique_ptr`.

```cpp
vector<unique_ptr<string>> noms;
noms.push_back(unique_ptr<string>(new string("Pierre")));
noms.push_back(unique_ptr<string>(new string("Paul")));
```

FIGURE 2

5:12 12:28

Exemple 2 d'utilisation d'un `unique_ptr`.

```cpp
unique_ptr<Personne> naissance(string nom) {
  unique_ptr<Personne> bb(new Personne);
  // .. initialise le contenu pointé par bb
  return bb;
}
// ...
unique_ptr<Personne> adresse_quidam( naissance("Pierre") );
```

FIGURE 3

6:13 12:28

Exemple 3 d'utilisation d'un `unique_ptr`.

SHARED_PTR ET WEAK_PTR

Les `unique_ptr` ne conviennent pas à toutes les situations. En particulier lorsque des zones mémoire doivent être partagées par différents pointeurs. Ce sont les `shared_ptr` qui sont destinés à cet usage. La libération de la zone pointée est aussi automatique, et a lieu lorsque tous les `shared_ptr` qui pointaient sur une zone mémoire ne pointent plus vers celle-ci ou ont été supprimés. Il peut arriver qu'il y ait un problème de dépendance cyclique entre plusieurs zones de code contenant des `shared_ptr` sur la même zone mémoire. Il serait alors impossible de libérer cette zone. Les `weak_ptr` permettent de casser cette dépendance cyclique et donc libérer cette zone mémoire. Ces deux types de pointeurs ont été présentés par souci d'exhaustivité mais sont des concepts avancés par rapport au contenu du cours. Le type de pointeur intelligent à retenir est `unique_ptr` qui permet une utilisation simplifiée par rapport aux pointeurs « à la C » et qui ne comporte pas les subtilités des deux autres types.

Initiation à la programmation en C++
Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

37. PUISSANCE 4 : INTRODUCTION

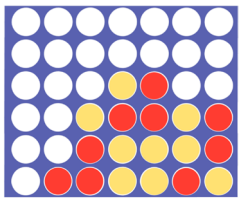


FIGURE 1

0:44 10:47

Jeu de Puissance 4.

Pour conclure ce cours d'introduction à la programmation, nous souhaitons présenter (leçons 37 à 43) un projet un peu plus conséquent que ce que nous avons vu jusqu'à présent. Ce projet consiste en un jeu de ***Puissance 4***.

Les règles du jeu de Puissance 4 sont les suivantes : il se joue à deux joueurs, sur une grille de sept colonnes et six lignes. Chaque joueur a une couleur (rouge ou jaune) et laisse tomber tour à tour un pion de sa couleur dans une colonne, jusqu'à ce que quatre pions de la même couleur soient alignés ou que la grille soit complètement remplie (fig. 1).

```
 | | | | | | | |
 | | | | | | | |
 | | | |X| | | |
 | | |X|O|O|X|O|
 | | |O|X|X|X|O|
 | |O|O|X|X|O|X|
==1=2=3=4=5=6=7==

Joueur O :  entrez un numéro de colonne
5

 | | | | | | | |
 | | | | | | | |
 | | | |X|O| | |
 | | |X|O|O|X|O|
 | | |O|X|X|X|O|
 | |O|O|X|X|O|X|
==1=2=3=4=5=6=7==

Le joueur O a gagne !
```

FIGURE 2

0:53 10:47

Affichage de notre jeu.

NOTRE IMPLÉMENTATION

Nous représenterons notre jeu sans interface graphique, c'est-à-dire dans la console (fig. 2). Lors du développement d'un projet conséquent, certaines règles sont à observer :

- Ne pas tenter d'écrire tout le programme en une seule fois, il faut ***décomposer*** le problème en sous-problèmes, c'est-à-dire développer le programme par étapes. Il est utile de ***tester*** le programme à chaque étape, afin de ne pas continuer sur des bases instables.
- Pour commencer à écrire un programme, il convient d'***identifier*** les ***types*** nécessaires pour représenter les données du programme.
- Ensuite, il faut ***identifier*** les ***fonctions*** qui portent sur ces types, les écrire et les tester au fur et à mesure.
- Lorsqu'une fonction est difficile à écrire, il est possible de la ***décomposer*** elle-même en fonctions plus petites traitant chacune les points difficiles de la fonction.

TYPES DE DONNÉES

Les données principales sont la grille et les pions que celle-ci contient. Nous choisissons de représenter la grille à l'aide d'un tableau statique bidimensionnel, plus précisément un `array` de six `array` représentant les lignes. Ces derniers disposent de sept cases et contiennent les pions. Ce choix du tableau est arbitraire, nous aurions pu aussi choisir un tableau de sept colonnes qui contiennent six cases. Les pions sont représentés pas un nouveau type défini à l'aide d'`enum`, ce qui permet de limiter le nombre de valeurs possibles que peuvent prendre les éléments de ce type, en l'occurrence trois. Ceci se code de cette façon :

```
enum Couleur {vide, rouge, jaune};
```

L'énumération `Couleur` peut être utilisée comme n'importe quel autre type, par exemple pour déclarer des variables ou pour effectuer des comparaisons. Nous pouvons maintenant définir entièrement le type `Grille` :

```
typedef array<array<Couleur, 7>, 6> Grille;
```

Nous pouvons ainsi effectuer les opérations suivantes (par exemple) :

```
Grille grille;
grille[2][3] = jaune;
```

Il reste encore à faire un choix quant à l'orientation du tableau : nous choisissons, arbitrairement, que l'élément d'indices `[0][0]` est en haut à gauche de la grille.

38. PUISSANCE 4 : PREMIÈRES FONCTIONS

Abordons maintenant nos premières fonctions ; les premières tâches à effectuer : construire le jeu et l'afficher.

INITIALISE()

Avant d'écrire une fonction, il faut se demander quelle tâche elle effectuera, et comment elle sera utilisée. Pour `initialise()` nous avons principalement deux possibilités : soit cette fonction prend une grille en argument, l'initialise et ne retourne rien, soit elle ne prend rien en argument mais retourne une grille initialisée. Nous avons choisi ici d'implémenter la première possibilité, `initialise()` aura donc pour prototype :

```
void initialise(Grille& grille);
```

Pour que la grille puisse être modifiée par cette fonction, nous utilisons un passage par référence.

Maintenant que nous avons écrit le prototype, il nous reste à écrire la définition. Celle-ci est assez simple, on utilise deux boucles `for(auto& : )` imbriquées et l'on affecte la valeur `vide` à la case courante. En C++, `case` est un mot-clef du langage, on ne peut donc pas l'utiliser comme nom de variable, c'est pour cela que nous avons utilisé le nom `kase`. Enfin, vu que la grille est modifiée, `ligne` et `kase` doivent être passées par référence dans la boucle. La définition complète est présentée sur la figure 1.

```
void initialise(Grille& grille)
{
  for(auto &ligne : grille) {
    for(auto &kase : ligne) { // "case" est un mot-cle reserve du C++
      kase = vide;
    }
  }
}
```

FIGURE 1

7:55 20:02

Code de la fonction `initialise()`.

AFFICHE()

Là encore il faut se poser les questions concernant l'écriture d'une fonction. Il est logique d'écrire le prototype de la fonction `affiche()` comme ceci :

```
void affiche(Grille grille);
```

A priori, un passage par valeur s'impose ; en effet, rien ne justifie un passage par référence, la grille n'étant pas modifiée par le processus d'affichage. Cependant, un passage par valeur implique la copie de la variable et doit donc être évité lorsque celle-ci est volumineuse. On lui préféra donc un passage par référence constante. Le prototype de `affiche()` devient donc une des deux lignes suivantes :

```
void affiche(const Grille& grille);
void affiche(Grille const& grille);
```

Celles-ci sont totalement équivalentes. L'utilisation d'une référence constante n'est pas une obligation mais une optimisation et une bonne pratique. La référence évite la copie et le `const` permet d'assurer que `grille` ne sera pas modifiée dans le corps de la fonction.

La définition de cette fonction est assez similaire à celle de la fonction `initialise()`. En effet, on parcourt la grille à l'aide de deux boucles imbriquées et suivant la valeur de la case, on affiche soit une espace, soit un `O`, soit un `X`. À la fin de chaque ligne, un saut de ligne est affiché. Dans les boucles, le passage par référence n'est pas nécessaire, car la grille n'est pas modifiée. Enfin il est très utile d'ajouter un commentaire décrivant le comportement d'une fonction, surtout lorsque celle-ci utilise des conventions choisies arbitrairement. La définition complète est présentée sur la figure 2.

```
// affiche O pour une case rouge, X pour une case jaune
void affiche(const Grille& grille)
{
  for(auto ligne : grille) {
    for(auto kase : ligne) {
      if (kase == vide) {
        cout << ' ';
      } else if (kase == rouge) {
        cout << 'O';
      } else {
        cout << 'X';
      }
    }
    cout << endl;
  }
}
```

FIGURE 2

14:03 20:02

Code de la fonction `affiche()`.

TESTS

Comme expliqué dans la leçon 37, lors du développement d'un projet, il est toujours utile d'implémenter des tests, afin de vérifier que les fonctions effectuent correctement les actions qu'elles sont censées exécuter. Pour cela, nous utilisons un petit programme de test présenté sur la figure 3. Le texte de sortie de ce programme est présenté à droite du programme de test. Le caractère ␣ est utilisé ici pour visualiser l'espace. La fonction a affiché les caractères aux endroits attendus, cependant, on remarque que l'affichage est un peu austère et que l'on ne peut pas facilement distinguer une colonne d'une autre. C'est pour cela que l'on se propose de compléter la fonction `affiche()`, afin de pouvoir visualiser les colonnes et leurs numéros.

```
int main()
{
  Grille grille;

  initialise(grille);
  grille[2][3] = jaune;
  grille[2][4] = rouge;
  affiche(grille);

  return 0;
}
```

```
␣␣␣␣␣␣␣
␣␣␣␣␣␣␣
␣␣␣XO␣␣
␣␣␣␣␣␣␣
␣␣␣␣␣␣␣
␣␣␣␣␣␣␣
```

FIGURE 3

15:25 20:02

Programme de test.

AFFICHAGE AMÉLIORÉ

On ajoute un symbole pour délimiter chaque colonne et on affiche les numéros de chaque colonne en dessous de la grille. Pour effectuer cette opération, on utilise une boucle `for` classique. Cependant, cette boucle n'itère pas sur les éléments de la grille, mais sur une variable `i` qui va de 1 à la taille d'une ligne. Le code et l'affichage sont présentés sur la figure 4.

```
// affiche O pour une case rouge, X pour une case jaune
void affiche(const Grille& grille)
{
  cout << endl;

  for(auto ligne : grille) {
    cout << " |";
    for(auto kase : ligne) {
      if (kase == vide) {
        cout << ' ';
      } else if (kase == rouge) {
        cout << 'O';
      } else {
        cout << 'X';
      }
      cout << '|';
    }
    cout << endl;
  }

  cout << '=';
  for(size_t i(1); i <= grille[0].size(); ++i) {
    cout << '=' << i;
  }
  cout << "==";

  cout << endl << endl;
}
```

```
 | | | | | | | |
 | | | | | | | |
 | | | |X|O| | |
 | | | | | | | |
 | | | | | | | |
 | | | | | | | |
==1=2=3=4=5=6=7==
```

FIGURE 4

17:00 20:02

Code de la fonction `affiche()` améliorée.

39. PUISSANCE 4 : FONCTION JOUE PREMIÈRE VERSION

Dans les leçons précédentes nous avons décrit les principaux types de données du programme ainsi que les fonctions de base permettant d'initialiser et d'afficher la grille de jeu. Nous nous intéressons maintenant au jeu lui-même, au déroulement de la partie. Pour jouer, il faut pouvoir :

1. Demander au joueur où il joue.
2. Valider son coup.
3. Demander à l'autre joueur, c'est-à-dire alterner les joueurs.
4. Vérifier si l'un gagne (ou si le jeu est plein).

Cette leçon s'intéresse au second point, l'implémentation de la fonction `joue()`.

PROTOTYPE

La fonction `joue()` recevra le numéro de la colonne dans laquelle le joueur veut placer son pion, puis elle vérifiera si cette action est possible et enfin elle mettra le jeu à jour. Pour déterminer son prototype, nous essayons d'imaginer les appels typiques de cette fonction. Par exemple :

```
joue(grille, 2, rouge);
```

qui placerait dans la colonne 2 de la grille `grille` un pion rouge. Plusieurs éléments sont à prendre en compte pour l'écriture du prototype. Premièrement, la grille sera modifiée, elle doit donc être passée par référence. Ensuite, le 2 correspond à un indice de tableau, il est donc sensé de le représenter par un `size_t`. Enfin, dans cet exemple d'appel typique, nous n'avons pas spécifié de valeur de retour. Donc le type de retour de `joue()` est donc `void`. Le prototype obtenu est alors :

```
void joue(Grille& grille, size_t colonne, Couleur rouge);
```

DÉFINITION

Nous pouvons maintenant coder ce que nous voulons faire. Pour commencer, on écrit un commentaire décrivant la tâche effectuée par la fonction, qui sera utile aux potentiels futurs programmeurs (y compris nous-mêmes) qui reliraient cette fonction. Pour `joue()`, on veut parcourir la colonne de bas en haut jusqu'à trouver une case vide où insérer le pion. Cela sera implémenté par une boucle `while`. Une fois cette case trouvée, on la remplit avec un pion de couleur `couleur`.

```
void joue(Grille& grille, size_t colonne, Couleur couleur)
{
  // on parcourt la colonne en partant du bas...
  size_t ligne(grille.size() - 1);
  // ...jusqu'à trouver une case vide :
  while (grille[ligne][colonne] != vide) {
    --ligne;
  }

  // on remplit la case vide trouvée :
  grille[ligne][colonne] = couleur;
}
```

FIGURE 1

8:52 18:22

Code de la fonction `joue()`.

```
int main()
{
  Grille grille;
  initialise(grille);
  affiche(grille);
  joue(grille, 3, rouge);
  affiche(grille);
  joue(grille, 2, jaune);
  affiche(grille);
  joue(grille, 3, rouge);
  affiche(grille);
  return 0;
}
```

FIGURE 2

9:03 18:22

Premier programme de test de `joue()`.

TEST

La figure 2 présente le programme de test de la fonction `joue()`. En l'exécutant nous obtenons les résultats désirés; cependant, nous n'avons pas effectué tous les tests possibles. Par exemple, lorsque l'on tente d'insérer un pion dans une colonne déjà pleine, on sort du tableau `grille`.

CORRECTION DE LA FONCTION

Pour corriger cette erreur de conception, nous introduisons un booléen qui permet de dire si la colonne est pleine. Nous modifions aussi la boucle pour qu'elle s'arrête lorsque l'on a atteint le haut de la colonne. Nous ajoutons une condition pour n'ajouter le pion que si la colonne n'est pas pleine. Nous pouvons profiter de cette condition pour retourner un booléen indiquant si l'on a pu jouer ou non. Il faut donc changer le type de retour de la fonction de `void` à `bool`.

```
bool joue(Grille& grille, size_t colonne, Couleur couleur)
{
  // on parcourt la colonne en partant du bas...
  size_t ligne(grille.size() - 1);
  // ...jusqu'à trouver une case vide...
  // ...ou jusqu'en haut de la colonne si la colonne est pleine :
  bool pleine(false);
  while ((not pleine) and (grille[ligne][colonne] != vide)) {
    if (ligne == 0) {
      pleine = true;
    } else {
      --ligne;
    }
  }

  // si on n'est pas arrivé jusqu'en haut de la colonne, on remplit la case vide trouvée,
  // sinon c'est que la colonne est pleine et le coup n'est pas valide :
  if (not pleine) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
}
```

FIGURE 3

14:00 18:22

Version corrigée de `joue()`.

TEST

Nous utilisons un deuxième programme de test pour vérifier que l'erreur de conception est résolue. Ce programme essaye d'ajouter trop de pions dans la même colonne et il vérifie que `joue()` ne les ajoute pas à la grille lorsque la colonne est pleine.

```
int main()
{
  Grille grille;

  initialise(grille);
  affiche(grille);

  for(int i(0); i < 10; ++i) {
    bool valide(joue(grille, 3, rouge));

    if (not valide) {
      cout << "impossible d'ajouter un pion sur cette colonne" << endl;
    }

    affiche(grille);
  }

  return 0;
}
```

FIGURE 4

14:15 18:22

Deuxième programme de test de `joue()`.

40. PUISSANCE 4 : FONCTION JOUE VARIANTES ET RÉVISION

Cette leçon présente deux variantes de la fonction `joue()`, qui place un pion d'une certaine couleur dans une grille et qui vérifie la validité du coup. Seule la définition de la fonction `joue()` est modifiée, le prototype reste le même.

PREMIÈRE VARIANTE

Nous nous proposons de changer l'algorithme en vérifiant si la colonne est pleine avant de la parcourir. Pour ce faire, il suffit de vérifier que la case tout en haut n'est pas vide. Si c'est le cas, on peut déjà sortir de la fonction avec un `return false;`. Cette version allège la définition de la fonction `joue()`, en éliminant les tests de plénitude de la colonne dans la boucle et après celle-ci. Il faudrait par ailleurs tester cette variante.

```
bool joue(Grille& grille, size_t colonne, Couleur couleur)
{
  // si la colonne est pleine, le coup n'est pas valide :
  if (grille[0][colonne] != vide) {
    return false;
  }

  // on parcourt la colonne en partant du bas jusqu'à trouver une case vide :
  size_t ligne(grille.size() - 1);
  while (grille[ligne][colonne] != vide) {
    --ligne;
  }

  // on remplit la case vide trouvée :
  grille[ligne][colonne] = couleur;

  return true;
}
```

FIGURE 1

2:28 8:43

Code de la première variante.

DEUXIÈME VARIANTE

Pour cette deuxième variante, nous nous proposons de supprimer le booléen `pleine`. On rappelle que ce booléen a été introduit pour éviter de décrémenter la variable `ligne` lorsqu'elle vaut `0`. Or, `ligne` est de type `size_t`, c'est-à-dire qu'elle ne peut prendre que des valeurs positives ou nulles. En pratique si l'on soustrait `1` à une variable de type `size_t` qui vaut `0`, on obtiendra un très grand nombre. Sa valeur précise importe peu, mais sera plus grande que le nombre de lignes de la grille. Donc pour tester que le numéro de ligne est compris entre `grille.size()-1` et `0`, on peut utiliser la condition `ligne<grille.size()`. Le contrôle des indices devrait d'ailleurs être fait de façon systématique (ou garanti) à chaque accès dans un tableau. La figure 2 présente le code de cette variante.

```
bool joue(Grille& grille, size_t colonne, Couleur couleur)
{
  // on parcourt la colonne en partant du bas...
  size_t ligne(grille.size() - 1);
  // ...jusqu'à trouver une case vide...
  // ...ou jusqu'en haut de la colonne si la colonne est pleine.
  // Si le test (ligne < grille.size()) devient faux, c'est qu'on a
  // soustrait 1 à ligne quand elle valait 0, ce qui arrive quand la
  // colonne est pleine.
  while ((ligne < grille.size()) and (grille[ligne][colonne] != vide)) {
    --ligne;
  }

  // Si ligne < grille.size(), on a trouvé une case vide et on la remplit,
  // sinon c'est que la colonne est pleine et le coup n'est pas valide :
  if (ligne < grille.size()) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
}
```

FIGURE 2

7:25 8:43

Code de la deuxième variante.

Nous avons vu que l'accès à la grille par rapport aux lignes est contrôlé, mais ce n'est pas le cas pour les colonnes ; le code complet de la fonction `joue()` est présenté dans la leçon 41.

41. PUISSANCE 4 : MOTEUR DE JEU

Cette leçon présente l'écriture du moteur de jeu, c'est-à-dire ce qui permet effectivement de jouer. Nous allons d'abord essayer d'écrire tout le code dans la fonction `main()`, puis nous modulariserons.

```
int main()
{
  Grille grille;

  initialise(grille);
  affiche(grille);

  Couleur couleur_joueur(jaune);
```

FIGURE 1

1:10 14:14

Première partie du `main()`, initialisation.

MAIN()

Le `main()` est constitué de deux parties, l'initialisation et la boucle de jeu.

L'initialisation est une partie assez simple, en effet, les fonctions `initialise()` et `affiche()` permettent d'effectuer ces actions de manière concise. La variable `couleur_joueur`, représentant la couleur du joueur dont c'est le tour doit aussi être initialisée.

La deuxième partie du `main()` est plus conséquente, elle comprend la boucle de jeu, c'est-à-dire la boucle qui se répète tant qu'aucun des joueurs n'a gagné ou que la grille n'est pas pleine. Cette boucle est constituée de plusieurs étapes :

1. Demande du numéro de colonne au joueur.
2. Récupération de ce numéro.
3. Évolution du jeu avec la fonction `joue()`.
4. Vérification de la validité du coup.
5. Affichage de la grille.
6. Changement de joueur, c'est-à-dire de couleur.

Ce code est présenté sur la figure 2. Il manque cependant la condition de terminaison de la boucle, qui sera traitée dans la leçon 42.

```
  do {
    if (couleur_joueur == jaune) {
      cout << "Joueur X :  entrez un numéro de colonne" << endl;
    } else {
      cout << "Joueur O :  entrez un numéro de colonne" << endl;
    }

    size_t colonne;
    cin >> colonne;
    --colonne;  // les indices des tableaux commencent par 0 en C++

    bool valide( joue(grille, colonne, couleur_joueur) );
    if (not valide) {
      cout << " > Ce coup n'est pas valide" << endl;
    }

    affiche(grille);

    // on change la couleur pour la couleur de l'autre joueur:
    if (couleur_joueur == jaune) {
      couleur_joueur = rouge;
    } else {
      couleur_joueur = jaune;
    }
  } while(    );

  return 0;
}
```

FIGURE 2

5:30 14:14

Deuxième partie du `main()`, boucle de jeu.

MODULARISATION

Plusieurs critiques peuvent être émises à l'encontre de ce `main()`. Premièrement, il ne peut pas tenir sur une seule figure, il est trop grand et doit donc être modularisé. Nous nous proposons donc de regrouper les quatre premières étapes ci-dessus dans une fonction `demande_et_joue()`. La deuxième partie du `main()` devient alors le code présenté sur la figure 3. Ce n'est évidemment pas la seule façon de modulariser ce code. C'est le fait de modulariser par concept qui est important, c'est-à-dire regrouper les concepts, les grandes fonctionnalités du code dans des fonctions. Le prototype de la fonction `demande_et_joue()` est le suivant :

```
void demande_et_joue(Grille& grille, Couleur couleur_joueur);
```

et sa définition est tout simplement le code qui est remplacé par cette fonction. À ce stade, il est utile de relire le code, de le compiler et de le tester.

Nous remarquons ensuite qu'une partie du code du premier point ci-dessus, la demande du numéro, est recopiée inutilement. C'est pourquoi nous choisissons de n'afficher dans la condition que le texte qui change en fonction de `couleur_joueur`, les parties communes étant affichées avant et après.

Il reste encore une amélioration, en effet, dans l'état actuel du code, lorsque le joueur entre un coup invalide, le programme continue de se dérouler, alors qu'il faudrait qu'il continue de demander au joueur un coup tant que celui-ci n'est pas valide, c'est pour cela que l'on introduit une boucle `do while`. Le code final de `demande_et_joue()` est présenté sur la figure 4.

```
    do {
      demande_et_joue(grille, couleur_joueur);

      affiche(grille);

      // on change la couleur pour la couleur de l'autre joueur:
      if (couleur_joueur == jaune) {
        couleur_joueur = rouge;
      } else {
        couleur_joueur = jaune;
      }
    } while(    );

    return 0;
  }
```

FIGURE 3

7:15 14:14

`main()` avec `demande_et_joue()`.

```
void demande_et_joue(Grille& grille, Couleur couleur_joueur)
{
  bool valide;

  cout << "Joueur ";
  if (couleur_joueur == jaune) {
    cout << 'X';
  } else {
    cout << 'O';
  }
  cout << " :  entrez un numéro de colonne" << endl;

  do {
    size_t colonne;
    cin >> colonne;
    --colonne;  // les indices des tableaux commencent par 0 en C++

    valide = joue(grille, colonne, couleur_joueur);
    if (not valide) {
      cout << " > Ce coup n'est pas valide" << endl;
    }
  } while(not valide);
}
```

FIGURE 4

12:15 14:14

Code de la fonction `demande_et_joue()`.

Dans la leçon 40, nous avions montré que la fonction `joue()` était incomplète, en effet, aucun contrôle n'était fait sur les colonnes. La version corrigée se trouve sur la figure 5.

```
bool joue(Grille& grille, size_t colonne, Couleur couleur)
{
  // si le numéro de colonne n'est pas valide, le coup n'est pas valide :
  if (colonne >= grille[0].size()) {
    return false;
  }

  // on parcourt la colonne en partant du bas...
  size_t ligne(grille.size() - 1);
  // ...jusqu'à trouver une case vide...
  // ...ou jusqu'en haut de la colonne si la colonne est pleine :
  bool pleine(false);
  while ((not pleine) and (grille[ligne][colonne] != vide)) {
    if (ligne == 0) {
      pleine = true;
    } else {
      --ligne;
    }
  }

  // si on n'est pas arrivé jusqu'en haut de la colonne, on remplit la case vide trouvée,
  // sinon c'est que la colonne est pleine et le coup n'est pas valide :
  if (not pleine) {
    grille[ligne][colonne] = couleur;
    return true;
  } else {
    return false;
  }
```

FIGURE 5

13:38 14:14

Code final de la fonction `joue()`.

42. PUISSANCE 4 : FONCTIONS EST_CE_GAGNE ET COMPTE

Cette leçon présente les deux fonctions nécessaires à la condition de terminaison de la boucle `do while` du `main()`. La fonction `est_ce_gagne()` vérifie pour chaque pion du joueur actuel s'il est contenu dans un alignement de quatre pions. La fonction `compte()` permet de compter les pions consécutifs de la même couleur dans une direction donnée.

EST_CE_GAGNE()

Le prototype de la fonction `est_ce_gagne()` est le suivant :

```
bool est_ce_gagne(const Grille& grille, Couleur couleur_joueur);
```

Cette fonction vérifie s'il existe un alignement de quatre pions de couleur `couleur_joueur` dans la grille `grille`. Il existe plusieurs façons d'implémenter cette fonction. Nous avons choisi l'algorithme suivant :

- On parcourt toutes les cases de la grille.
- On vérifie si cette case est de couleur `couleur_joueur`.
- Si c'est le cas, on parcourt la grille dans les huit directions : les deux directions verticales, les deux horizontales, et les quatre diagonales.
- On compte le nombre de pions consécutifs de la couleur `couleur_joueur` dans chacune de ces directions. Si ce nombre est supérieur ou égal à quatre dans l'une des directions, le joueur a gagné.

Il n'est en fait pas nécessaire de parcourir la grille dans les huit directions, en effet, chaque direction peut être parcourue dans les deux sens. Donc seules quatre directions suffisent.

Notre implémentation suit l'algorithme exposé ci-dessus. On commence donc par parcourir la grille case par case, à l'aide de deux boucles `for` imbriquées. Puis, on teste si la case est de la couleur `couleur_joueur`. Le cas échéant, on compte les pions dans les quatre directions à l'aide de la fonction `compte()`, décrite ci-dessous. Cette fonction prend comme arguments la grille, les coordonnées de la case et la direction selon laquelle les pions sont comptés. En réalité, la case est identifiée par l'indice de sa ligne et l'indice de sa colonne et la direction est représentée par deux nombres qui définissent l'incrément ajouté à la ligne et à la colonne. Par exemple le couple (0, +1) définit un déplacement vers la droite, le couple (+1, 0) vers le bas. On vérifie donc les quatre directions données par les couples (-1, +1), (0, +1), (+1, +1) et (+1, 0). Si l'une d'elles contient plus de quatre pions consécutifs de la même couleur, la fonction retourne true, sinon le parcours des cases continue.

```
// gagne = est_ce_gagne(grille, couleur_joueur);
bool est_ce_gagne(const Grille& grille, Couleur couleur_joueur)
{
  for(size_t ligne(0); ligne < grille.size(); ++ligne) {
    for(size_t colonne(0); colonne < grille[ligne].size(); ++colonne) {
      Couleur couleur_case(grille[ligne][colonne]);

      if (couleur_case == couleur_joueur) {
        if (// en diagonale, vers le haut et la droite:
            compte(grille, ligne, colonne, -1, +1) >= 4 or
            // horizontalement, vers la droite:
            compte(grille, ligne, colonne,  0, +1) >= 4 or
            // en diagonale, vers le bas et la droite:
            compte(grille, ligne, colonne, +1, +1) >= 4 or
            // verticalement, vers le bas:
            compte(grille, ligne, colonne, +1,  0) >= 4) {
          return true;
        }
      }
    }
  }

  return false;
}
```

FIGURE 1

Code de la fonction `est_ce_gagne()`.

COMPTE()

La fonction compte a pour prototype

```
unsigned int compte(
const Grille& grille, size_t ligne_depart,
size_t colonne_depart, int dir_ligne, int dir_colonne);
```

où `ligne_depart` et `colonne_depart` représentent la position de la case à partir de laquelle on compte ; `dir_ligne` et `dir_colonne` définissent la direction dans laquelle on se déplace. Cette fonction est assez simple, elle utilise une variable `compteur` qui est incrémentée dans une boucle `while`. Cette boucle incrémente par ailleurs les variables `ligne` et `colonne` avec les valeurs de `dir_ligne` et `dir_colonne`, respectivement. Elle continue tant que la couleur de la case à la position `ligne` et `colonne` est la même que celle de la case de départ et que les indices `ligne` et `colonne` ne sont pas hors de la grille.

```
// if (compte(grille, ligne, colonne, -1, +1) >= 4 ...
unsigned int compte(const Grille& grille,
                    size_t ligne_depart, size_t colonne_depart,
                    int dir_ligne, int dir_colonne)
{
  unsigned int compteur(0);

  size_t ligne(ligne_depart);
  size_t colonne(colonne_depart);

  while (grille[ligne][colonne] == grille[ligne_depart][colonne_depart]) {

    ++compteur;
    ligne   = ligne   + dir_ligne;
    colonne = colonne + dir_colonne;

  }

  return compteur;
}
```

FIGURE 2

13:15 15:51

Code de la fonction `compte()`.

OPTIMISATION DE EST_CE_GAGNE()

La fonction `est_ce_gagne()` effectue parfois des tests inutiles. Par exemple, lorsqu'un pion est en haut à droite de la grille, il est inutile de compter les pions vers le haut ou vers la droite. Plus généralement, lorsqu'un pion est à moins de quatre cases d'un bord, il est inutile de compter dans cette direction. Nous introduisons donc de nouveaux tests permettant de vérifier ces conditions.

```
            if (couleur_case == couleur_joueur) {
              const size_t ligne_max(grille.size() - 4);
              const size_t colonne_max(grille[ligne].size() - 4);

              if (// en diagonale, vers le haut et la droite:
                  (ligne >= 3 and
                   compte(grille, ligne, colonne, -1, +1) >= 4) or

                  // horizontalement, vers la droite:
                  (colonne <= colonne_max and
                   compte(grille, ligne, colonne,  0, +1) >= 4) or

                  // en diagonale, vers le bas et la droite:
                  (ligne <= ligne_max and colonne <= colonne_max and
                   compte(grille, ligne, colonne, +1, +1) >= 4) or

                  // verticalement, vers le bas:
                  (ligne <= ligne_max and
                   compte(grille, ligne, colonne, +1,  0) >= 4)) {
                    return true;
              }
            }
          }
        }
```

FIGURE 3

15:34 15:51

Optimisation des conditions de `est_ce_gagne()`.

43. PUISSANCE 4 : FINALISATION

Le programme de notre projet est presque terminé, il manque seulement l'affichage du joueur qui a gagné et la gestion du match nul, c'est-à-dire de la grille pleine.

FINALISATION DU MAIN()

Nous ajoutons d'abord l'affichage du vainqueur, à l'aide d'une simple condition à la suite de la boucle. À la sortie de celle-ci, `couleur_joueur` contient la couleur du perdant ; en effet, la couleur est changée *avant* de sortir de la boucle. Il convient donc d'intervertir les messages. La figure 1 présente le code de cette partie.

Enfin, il reste à traiter le cas où la grille est pleine. Pour cela on introduit une fonction `plein()`, décrite ci-dessous, qui prend la grille en argument et qui retourne un booléen. On ajoute la négation de cette fonction à la condition d'arrêt de la boucle et l'on modifie aussi l'affichage du vainqueur, pour qu'en cas de match nul, un message soit affiché.

```
// attention, on a change la couleur pour la couleur de l'autre joueur !
if (couleur_joueur == jaune) {
  cout << "Le joueur O a gagne !" << endl;
} else {
  cout << "Le joueur X a gagne !" << endl;
}
```

FIGURE 1

2:25 8:07

Affichage du vainqueur.

```
  // on change la couleur pour la couleur de l'autre joueur:
  if (couleur_joueur == jaune) {
    couleur_joueur = rouge;
  } else {
    couleur_joueur = jaune;
  }
} while(not gagne and not plein(grille));

if (gagne) {
  // attention, on a change la couleur pour la couleur de l'autre joueur !
  if (couleur_joueur == jaune) {
    cout << "Le joueur O a gagne !" << endl;
  } else {
    cout << "Le joueur X a gagne !" << endl;
  }
} else {
  cout << "Match nul !" << endl;
}

return 0;
}
```

FIGURE 2

4:08 8:07

Version finale du `main()`.

PLEIN()

Le prototype de la fonction déterminant si le jeu est plein est assez naturellement le suivant :

```
boule plein(const Grille& grille);
```

Il n'est pas nécessaire d'itérer sur toutes les cases de la grille et de les tester, il suffit de vérifier le contenu de la première ligne de la grille. En effet, la grille se remplit de bas en haut, il ne peut donc y avoir de case vide sous un pion. On itère donc sur la ligne du haut, on retourne `false` si l'on trouve une case vide et l'on retourne `true` si l'on n'en trouve aucune.

```
// plein(grille)
bool plein(const Grille& grille)
{
  // Si on trouve une case vide sur la premiere ligne, la grille n'est pas pleine :
  for(auto kase : grille[0]) {
    if (kase == vide) {
      return false;
    }
  }

  // Sinon, la grille est pleine :
  return true;
}
```

FIGURE 3

6:05 8:07

Code de la fonction `plein()`.

CONCLUSION DU PROJET

Le programme de ce Puissance 4 est maintenant complet. La principale chose à retenir de cette étude de cas sont les règles de développement d'un projet, qui ont été présentées à la leçon 37, et qui sont, pour mémoire :

- Ne pas tenter d'écrire tout le programme en une seule fois, il faut ***décomposer*** le problème en sous-problèmes, c'est-à-dire développer le programme par étapes. Il est utile de ***tester*** le programme à chaque étape, afin de ne pas continuer sur des bases instables.
- Pour commencer à écrire un programme, il convient d'***identifier*** les ***types*** nécessaires pour représenter les données du programme.
- Ensuite, il faut ***identifier*** les ***fonctions*** qui portent sur ces types, les écrire et les tester au fur et à mesure.
- Lorsqu'une fonction est difficile à écrire, il est possible de la ***décomposer*** en fonctions plus petites traitant chacune les points difficiles de la fonction.

Le code complet de l'étude de cas peut être téléchargé **ici**.

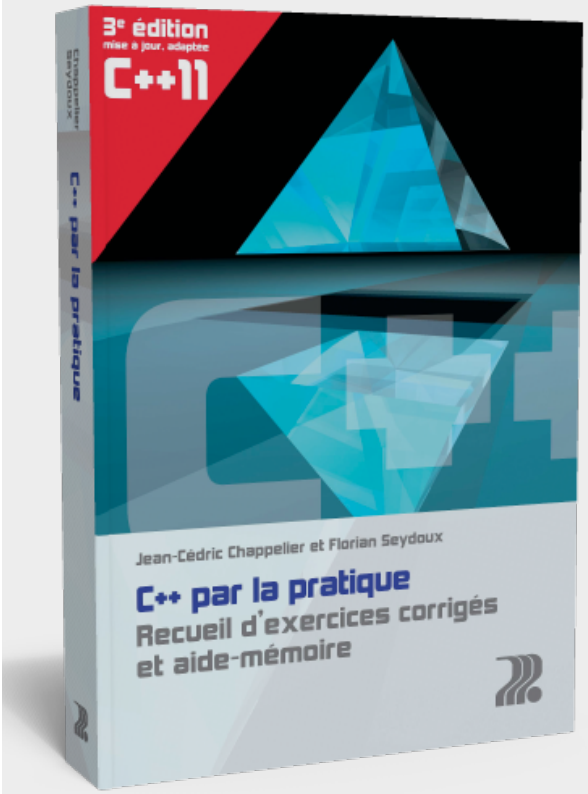
IMPRESSUM

Graphisme :
Emphase Sàrl, Lausanne

Résumé : Chang Su Yeon et Arbez-Gindre Félix

Développés par EPFL Press, les BOOCs (Book and Open Online Courses) sont le support compagnon des MOOCs proposés par l'École polytechnique fédérale de Lausanne. Valeur ajoutée aux MOOCs, ils rassemblent l'essentiel à retenir pour l'obtention du certificat et constituent un atout pédagogique. Learn faster, learn better. Bonne révision !

ISBN 978-2-88914-396-2



OUVRAGE DE RÉFÉRENCE DU MÊME AUTEUR

Cet ouvrage a pour objectif d'offrir la pratique nécessaire à tout apprentissage de la programmation : un cadre permettant au débutant de développer ses connaissances sur des cas concrets. Il se veut un complément pédagogique à un support de cours. Avec près d'une centaine d'exercices gradués de programmation en C++, accompagnés d'une solution complète et souvent détaillée, l'ouvrage est structuré en deux parties : la première présente la programmation procédurale, tandis que la seconde aborde le paradigme de la programmation objet. Chacune de ces parties se termine par un chapitre regroupant des exercices généraux. Chaque chapitre contient un exercice pas à pas et une série d'exercices classés par niveaux de difficulté.