



## Chapitre 3 Fonctions

Auteur : Marco Lavoie  
Adaptation : Sébastien Bois, Hiver 2021

Langage C++  
25908 IFM

La plupart des programmes informatiques qui résolvent des problèmes du monde réel sont beaucoup plus imposants que ceux présentés dans les chapitres précédents. L'expérience a démontré que la meilleure façon de développer et de maintenir un gros programme consiste à le bâtir à partir de petits éléments ou composants, car ceux-ci sont plus maniables qu'un gros programme. De plus, ces composants sont conséquemment réutilisables puisqu'ils peuvent ultérieurement être intégrés à d'autres programmes.

On qualifie cette technique de programmation par l'expression « diviser pour régner ».



### Objectifs

- Créer des nouvelles fonctions
- Comprendre les mécanismes de communication de données entre fonctions
- Comprendre la portée des identificateurs
- Se familiariser avec les stratégies de débogage

En C++, les modules de code source sont désignés par les termes *fonctions* et *classes*. La conception d'un programme en C++ consiste donc à combiner les nouvelles fonctions écrites par le programmeur avec des fonctions déjà disponibles dans la *bibliothèque standard du C++* et à combiner les nouvelles classes créées par le programmeur aux classes disponibles dans différentes bibliothèques de classes.

Ce chapitre porte sur les fonctions; les classes seront analysées en détails au chapitre 6.

Nous étudions aussi le rôle du débogueur de *Visual Studio* qui permet d'exécuter un programme pas à pas afin d'en identifier et corriger les erreurs de logique.

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++



### Aperçu

- Fonctions mathématiques
- Définition de fonctions
  - Prototypes
  - Format de déclaration d'une fonction
- Fichiers d'en-tête standards
- Énumérations
- Portée et classes de stockage
- Fonctions, paramètres et arguments
- *Microsoft Visual Studio* - le débogueur

La bibliothèque standard du C++ offre une riche collection de fonctions pour effectuer les calculs mathématiques courants, la manipulation de chaînes, la manipulation de caractères, les entrées/sorties, la vérification d'erreurs et nombre d'autres opérations utiles. Dans ce chapitre nous présentons les fonctions mathématiques fournies par cette bibliothèque.

Nous présentons aussi comment un programmeur peut définir ses propres fonctions afin de configurer un programme en modules. Il existe plusieurs raisons de « fonctionnaliser » un programme. L'approche « diviser pour mieux régner » améliore la maniabilité du développement du programme. Un autre motif est la *réutilisation du logiciel*, à savoir : l'utilisation de fonctions existantes comme blocs de construction pour créer de nouveaux programmes.

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

## Fonctions mathématiques

- Inclure le fichier d'en-tête **<cmath>**  
`#include <cmath>`

Fonction	Description	Exemple
<code>ceil( x )</code>	Arrondit $x$ au plus petit entier $\geq x$	<code>ceil( 9.2 )</code> donne 10.0
<code>cos( x )</code>	Cosinus trigonométrique ( $x$ en radians)	<code>cos( 0.0 )</code> donne 1.0
<code>exp( x )</code>	Fonction exponentielle $e^x$	<code>exp( 1.0 )</code> donne 2.718282
<code>fabs( x )</code>	Valeur absolue de $x$	<code>fabs( -5.1 )</code> donne 5.1
<code>floor( x )</code>	Arrondit $x$ au plus grand entier $\leq x$	<code>floor( 9.2 )</code> donne 9.0
<code>log( x )</code>	Logarithme naturel de $x$ (base $e$ )	<code>log( 2.718282 )</code> donne 1.0
<code>log10( x )</code>	Logarithme de $x$ (base 10)	<code>log10( 100.0 )</code> donne 2.0
<code>pow( x, y )</code>	$x$ à la puissance $y$ ( $x^y$ )	<code>pow( 2, 7 )</code> donne 128
<code>sin( x )</code>	Sinus trigonométrique ( $x$ en radians)	<code>sin( 0.0 )</code> donne 0.0
<code>sqrt( x )</code>	Racine carrée de $x$	<code>sqrt( 9.0 )</code> donne 3.0
<code>tan( x )</code>	Tangente trigonométrique ( $x$ en radians)	<code>tan( 0.0 )</code> donne 1.0

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Les fonctions de la bibliothèque mathématique permettent au programmeur d'effectuer certains calculs mathématiques courants. On *invoque* généralement les fonctions en écrivant leur nom suivi d'une parenthèse gauche, suivi de l'*argument* (ou par une liste d'arguments séparés par des virgules) de la fonction, suivi d'une parenthèse droite. Par exemple, un programmeur désirant calculer la racine carrée de 900.0 écrira

```
cout << sqrt( 900.0 );
```

Lorsque cette instruction est exécutée, la fonction **sqrt** de la bibliothèque **<cmath>** est appelée à calculer la racine carrée de l'argument fourni (900.0). L'instruction ci-haut affichera 30.

Toutes les fonctions de la bibliothèque mathématique renvoient un résultat de type **double**.

## Définition d'une fonction

- Exemple

```
// Exemple de définition de fonction
#include <iostream>

int carre( int ); // Prototypé de fonction

int main() {
    for (int x = 1; x <= 10; x++)
        std::cout << carre( x );

    return 0;
}

// Définition de la fonction
int carre( int y ) {
    return y * y;
}
```

Prototypé de la fonction

Invocation de la fonction

Définition de la fonction

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Chacun des programmes vus à date se résumait à une fonction nommée **main** appelant des fonctions de la bibliothèque standard pour accomplir des tâches. Le code ci-contre présente la définition et invocation d'une nouvelle fonction définie par le programmeur.

La fonction **carre** est invoquée (ou appelée) dans **main** avec l'appel

```
carre( x )
```

La fonction **carre** reçoit une copie de la valeur de **x** dans le *paramètre* **y**. Ensuite, **carre** calcule **y \* y**. Le résultat est transmis à l'endroit où la fonction **carre** a été invoquée dans **main**, puis est affiché. Notez que la valeur de **x** n'est pas changée par l'appel de la fonction. Ce processus est répété dix fois en utilisant la structure répétitive **for**.

## Prototype

- Permet au compilateur de valider les appels à la fonction avant d'en connaître la définition
  - Indique les informations suivantes
    - Le type de la valeur de retour
    - Le nom de la fonction
    - Le nombre de paramètres et le type de chacun

```
// Exemple de fonction
#include <iostream>

int carre( int ); // Prototypé

int main() {
    for (int x = 1; x <= 10; x++)
        std::cout << carre( x );

    return 0;
}

// Définition de la fonction
int carre( int y ) {
    return y * y;
}
```

La ligne

```
int carre( int ); // Prototypé de fonction
```

est le *prototypé de fonction*. Le type de données **int** entre parenthèses informe le compilateur que la fonction **carre** attend une valeur d'entier de la fonction d'appel. Le type de données **int** à gauche du nom de la fonction **carre** avise le compilateur que **carre** renvoie un résultat entier à la fonction d'appel. Le compilateur se réfère au prototype de fonction pour vérifier si les invocations de **carre** contiennent le bon type de données renvoyées, le bon nombre et le bon type d'arguments et si les arguments sont dans le bon ordre.

La portion d'un prototype de fonction incluant le nom de la fonction et les types de ses arguments s'appelle la *signature de la fonction*, ou simplement la *signature*. La signature d'une fonction n'inclut pas le type de retour de la fonction.

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

## Prototype (suite)

- Le prototype n'est pas requis si la définition de la fonction est positionnée avant toute invocation

- Ce n'est pas toujours possible
  - Nous verrons des exemples plus tard
- Il est préférable de toujours positionner la fonction principale (`main()`) en premier

```
// Exemple de fonction
#include <iostream>

// Définition de la fonction
int carre( int y ) {
    return y * y;
}

int main() {
    for (int x = 1; x <= 10; x++ )
        std::cout << carre( x );

    return 0;
}
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Le prototype de la fonction n'est pas requis si la définition de la fonction apparaît avant la première invocation de celle-ci dans le programme. Le cas échéant, la définition de la fonction agit également comme prototype de la fonction.

Comme mentionné précédemment, le rôle du prototype est d'informer le compilateur du format d'invocation requis par la fonction `carre`. Lorsque le compilateur compile la ligne d'invocation de `carre` dans `main`, il peut valider cette invocation même s'il n'a pas encore compilé la définition de la fonction, puisque celle-ci apparaît plus loin dans le fichier. Si la définition de la fonction `carre` est déplacée avant la fonction `main`, le prototype de `carre` n'est plus requis puisque le compilateur aura déjà compilé la définition de `carre` lorsqu'il compilera son invocation dans `main`, étant alors apte à valider l'invocation sans nécessiter un prototype de `carre`.

## Déclaration de la fonction

- Syntaxe

```
type_valeur_de_retour nom_de_fonction( liste_parametres )
{
    suite_instructions ;
}
```

- Si aucune valeur de retour n'est requise
  - Le mot-clé `void` est spécifié comme type de retour
  - L'instruction `return` ne fournit pas de valeur
- Exemple

```
// Affiche "Bonjour"
void bonjour() {
    std::cout << "Bonjour" << std::endl;
    return;
}
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Le *nom\_de\_fonction* peut être un identificateur valide. Le *type\_valeur\_de\_retour* est le type de données du résultat retourné par la fonction lors de son invocation. Le type de la valeur renvoyée `void` indique qu'une fonction ne retourne aucune valeur. Lorsque le type de valeur renvoyée n'est pas spécifié, le compilateur prend pour acquis qu'il est de type `int`.

## Déclaration de la fonction (suite)

- Syntaxe

```
type_valeur_de_retour nom_de_fonction( liste_parametres ) {
    suite_instructions ;
}
```

- Le type de chaque paramètre doit être fourni individuellement
  - Conforme à la syntaxe des prototypes
  - Exemple : `(int x, int y)` et non pas `(int x, y)`
- À l'invocation, les parenthèses doivent accompagner le nom de la fonction même si celle-ci n'a aucun paramètre

```
int main() {
    bonjour(); // Message d'accueil
    return 0;
}
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

La *liste\_parametres* est une liste séparée par des virgules qui contient les déclarations des paramètres reçus par la fonction lorsqu'elle est appelée. Si une fonction ne reçoit aucune valeur, cette liste est `void` ou simplement laissée vide. On doit explicitement spécifier un type pour chacun des paramètres de la liste de paramètres d'une fonction.

Il existe trois façons de renvoyer le contrôle à l'endroit précis où une fonction a été invoquée. Si la fonction ne retourne par de résultat, le contrôle est simplement renvoyé après l'exécution de la dernière instruction du corps de la fonction (i.e. de *suite\_instructions*), ou en exécutant l'instruction

```
return;
```

Si la fonction retourne un résultat, l'instruction

```
return expression;
```

retourne la valeur de l'*expression* à l'appelant.

# Fonctions avec paramètres

## • Deuxième exemple

```
// Exemple de définition de fonction
#include <iostream>

int maximum( int, int, int ); // Prototypé

// Programme principal
int main() {
    int a, b, c, plusGrand;

    std::cout << "Entrez trois entiers: ";
    std::cin >> a >> b >> c;

    // Trouver et afficher le maximum
    plusGrand = maximum( a, b, c );
    std::cout << "La valeur maximale est "
              << plusGrand << std::endl;

    return 0;
}
```

```
// Maximum de trois valeurs
int maximum( int x, int y, int z ) {
    int max = x;

    if ( y > max )
        max = y;

    if ( z > max )
        max = z;

    return max;
}
```

Ce deuxième exemple utilise une fonction définie par le programmeur, **maximum**, pour déterminer et renvoyer le plus grand de trois entiers. Les trois entiers sont entrés dans **main** puis sont passés vers **maximum** qui détermine le plus grand entier. Cette valeur est retournée à **main** par l'instruction **return** à l'intérieur de **maximum**. La valeur retournée est affectée à la variable **plusGrand** et ensuite affichée.

Voici une exécution de ce programme :

```
Entrez trois entiers: 22 85 17
La valeur maximale est 85
```

# Exercice #3.1

- Solutionnez l'exercice distribué par l'instructeur
  - Créer un nouveau projet console *Visual Studio*
  - Solutionner le problème tel que décrit
  - N'oubliez pas les conventions d'écriture
  - Soumettez votre projet selon les indications de l'instructeur

Les deux fonction à définir reçoivent en paramètre une température (valeur entière) et retournent la température convertie (aussi une valeur entière). Déterminez les formules de conversion de Celsius à Fahrenheit et vice-versa à partir d'une recherche sur Internet.

Le programme principal doit exploiter une boucle **for** pour produire chaque tableau de conversions.

# Fichiers d'en-tête standards

- Plusieurs fichiers d'en-tête standards sont fournis avec le compilateur C++, dont

Fichier d'en-tête	Description
<iostream>	Prototypes de fonctions d'entrées/sorties standards
<iomanip>	Prototypes de fonctions de formatage de flux d'entrées/sorties
<ctype>	Prototypes de fonctions de manipulation de caractères
<climits>	Fournie diverses limites de valeurs entières
<cmath>	Fournie diverses limites de valeurs flottantes
<cmath>	Prototypes de fonctions mathématiques
<stdio>	Prototypes de fonctions d'entrées/sorties standards
<stdlib>	Prototypes de fonctions utilitaires telles que de conversion, de gestion de mémoire et de nombres aléatoire
<time>	Prototypes de fonctions de manipulation de dates et d'heures
<string>	Prototypes de fonctions de traitement de chaînes de caractères
<fstream>	Prototypes de fonctions d'entrées/sorties aux fichiers

Chaque bibliothèque standard possède un *fichier d'en-tête* correspondant qui contient les prototypes de fonction pour toutes les fonctions de cette bibliothèque et les définitions de différents types de données et constantes requis par ces fonctions.

## Définition de fichiers d'en-tête

13

- Le programmeur peut définir ses propres fichiers d'en-tête
  - Nous verrons plus tard comment définir des fichiers d'en-tête
  - #include** permet d'intégrer un tel fichier dans notre programme, mais avec une syntaxe modifiée
    - Exemple :
 

```
#include <iostream>
#include "carre.h"
```
    - L'utilisation des guillemets ("") indique que ce n'est pas un fichier d'en-tête standard
 

```
// Programme principal
int main() {
```
    - Les fichiers d'en-tête non-standards doivent obligatoirement inclure l'extension **.h**

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Le programmeur peut créer des fichiers d'en-tête personnalisés. Les fichiers d'en-tête définis par le programmeur doivent se terminer par l'extension **.h**. On peut inclure un fichier d'en-tête défini par le programmeur en utilisant la directive de précompilation **#include**.

Notez que les fichiers d'en-tête créés par le programmeur sont inclus par **#include** en encerrant le nom du fichier de *guillemets doubles* (p.ex. **#include "carre.h"**) alors que les fichiers d'en-tête de la bibliothèque standard sont inclus par **#include** en encerrant le nom du fichier de *crochets* (p.ex. **#include <iostream>**).

## Exemple : nombres aléatoires

14

- Exploiter les fonctions de manipulation de nombres aléatoires
  - Définies dans le fichier **<cstdlib>**
  - La fonction **rand()** retourne un entier au hasard entre 0 et **RAND\_MAX** (constante définie dans **<cstdlib>**)
  - Pour reporter l'échelle [0...**RAND\_MAX**] à [a...b]: **a + rand() % b**
    - Le modulo ramène l'échelle entre 0 et b-1
    - L'addition de a décale l'échelle de 0 à a

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

La fonction **rand** choisit un nombre au hasard entre 0 et **RAND\_MAX** (constante symbolique dont la valeur dépend du système d'exploitation de l'ordinateur, mais d'un minimum 32767). Chaque entier entre 0 et **RAND\_MAX** possède une chance égale d'être choisi à chaque appel de **rand** (ce qui signifie que, théoriquement, un même entier pourrait être retourné par deux appels consécutifs à **rand**).

L'échelle des valeurs directement produite par **rand** diffère souvent de ce qui est requis dans un programme. Pour modifier cette échelle, on exploite habituellement l'opérateur arithmétique modulo (%) pour réduire l'échelle, et l'addition (+) pour la décaler. Par exemple dans l'expression

$$1 + \text{rand}() \% 6$$

L'expression **rand() % 6** ramène l'échelle 0...**RAND\_MAX** à 0...5, puis l'addition de 1 ramène cette échelle à 1...6, soit la valeur d'un dé de jeu à six faces.

## Exemple : nombres aléatoires (suite)

15

- Programme lançant deux dés et affichant les résultats

```
C:\>des.exe
1, 4
5, 5
6, 2
3, 1
6, 5
4, 3
6, 6
3, 1
4, 2
5, 2
C:\>
```

```
#include <iostream>
#include <cstdlib>

int lancerDe(); // Prototypage de fonction

// Programme lançant deux dés à répétition
int main() {
    for (int i = 0; i < 10; i++) {
        int de1 = lancerDe();
        int de2 = lancerDe();

        std::cout << de1 << ", " << de2
                  << std::endl;
    }

    return 0;
}

// Simule le lancer d'un dé
int lancerDe() {
    return 1 + rand() % 6;
}
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Comme le démontre l'exemple ci-contre, la fonction **rand** permet d'introduire les notions de hasard dans un programme.

La fonction **rand** génère en réalité des *nombres pseudo-aléatoires*. Un appel répété de la fonction **rand** produit une séquence de nombres qui semble aléatoire. Toutefois, la même séquence se répète à chaque exécution du programme. C'est parce qu'il est pratiquement impossible pour un ordinateur de générer des nombres de façon complètement aléatoire, alors **rand** exploite plutôt une fonction mathématique sophistiquée pour générer une séquence de nombres qui « semble » aléatoire, mais ne l'est pas en réalité (puisque c'est une fonction mathématique qui est appliquée).



## Exemple : nombres aléatoires (suite)

16

- Si vous exécutez le programme précédent à répétition, les mêmes réponses sont affichées
  - Est-ce vraiment aléatoire ?
    - **Non** : une fonction mathématique produit la séquence *pseudo-aléatoire*
    - Cette séquence débute toujours à la même valeur de départ, appelée le *germe*
  - Pour modifier le germe de départ, utilisez la fonction **srand( unsigned int )**

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++



## Exemple : nombres aléatoires (suite)

17

- On utilise l'heure courante comme germe
  - L'heure étant différente à chaque exécution

```
#include <iostream>
#include <cstdlib>
#include <ctime>

int lancerDe(); // Prototypage de fonction

// Programme lançant deux dés à répétition
int main() {
    // Initialiser le germe selon l'heure courante
    srand( time( 0 ) );

    for ( int i = 0; i < 10; i++ ) {
        int de1 = lancerDe();
        int de2 = lancerDe();
    }
}
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++



## Exercice #3.2

18

- Solutionnez l'exercice distribué par l'instructeur
  - Créer un nouveau projet console *Visual Studio*
  - Solutionner le problème tel que décrit
  - N'oubliez pas les conventions d'écriture
  - Soumettez votre projet selon les indications de l'instructeur

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Comme on vient de le mentionner, la même séquence de nombres se répète à chaque exécution du programme car c'est une fonction mathématique qui la génère. On peut cependant conditionner la fonction **rand** de façon qu'elle génère une séquence différente de nombres aléatoires à chaque exécution. Ce processus s'appelle *randomisation* et s'accomplit avec la fonction **srand** de la bibliothèque standard. La fonction **srand** prend un argument entier et *donne une valeur de départ* à la fonction **rand** afin qu'elle produise une séquence de nombres aléatoires différente à chaque exécution du programme.

Si nous désirons randomiser la fonction **rand** avec une valeur de départ différente à chaque exécution du programme, nous pouvons utiliser une instruction comme

```
srand( time( 0 ) )
```

Cette instruction dit à l'ordinateur de lire son horloge afin d'obtenir automatiquement une valeur de départ. La fonction **time** (avec l'argument 0 écrit comme ci-dessus) renvoie la valeur de l'heure courante au calendrier, en secondes. Cette valeur est convertie en un entier puis utilisée comme valeur de départ pour produire les nombres aléatoires.

Le prototype de la fonction **time** se trouve dans **<ctime>**.

Au *jeu de la barbotte* (vous verrez comment se joue ce jeu dans le cadre du prochain devoir), deux dés sont lancés et les gains du joueur sont déterminés selon les résultats obtenus. La probabilité d'obtenir une valeur spécifique sur un dé est 1/6, mais celle d'obtenir une valeur spécifique entre 1 et 12 sur la somme de deux dés n'est pas 1/12 puisque certaines valeurs résultent de plus d'une combinaison de dés. Par exemple, seule la combinaison 1+1 permet d'obtenir un total de 2, mais les combinaisons 1+4, 2+3, 3+2 et 4+1 permettent toutes d'obtenir un total de 5.

Puisque le jeu de la barbotte est basé en bonne partie sur la probabilité d'obtenir un total de 7 aux dés, quelle est cette probabilité?



## Énumérations

19

- Le mot-clé **enum** permet de définir un ensemble de constantes entières (**int**) liées à un nom d'ensemble

- Exemple

```
enum Jour { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI,
            SAMEDI, DIMANCHE };
```

- On peut ensuite définir des variables de ce nouveau type

```
Jour j;
j = DIMANCHE;
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Une énumération, introduite par le mot-clé **enum** et suivi par un nom de type (dans l'exemple ci-contre, **Jour**), est une série de constantes d'entiers représentées par des identificateurs. Ces identificateurs doivent être uniques.

Les variables du type de l'énumération ne peuvent être affectées qu'à l'une des constantes déclarées dans l'énumération.

## Énumérations (suite)

20

- À moins d'indication contraire, la première constante de l'énumération se voit attribuer la valeur 0

```
enum Jour { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI,
            SAMEDI, DIMANCHE };
Jour j = DIMANCHE;

std::cout << LUNDI << ", " << j; // Affiche 0,6
```

- On peut changer la valeur de départ de l'énumération

```
enum Jour { LUNDI = 1, MARDI, MERCREDI, JEUDI, VENDREDI,
            SAMEDI, DIMANCHE };

std::cout << LUNDI << ", " << DIMANCHE; // Affiche 1,7
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Comme la première valeur de l'énumération au bas de la page ci-contre est spécifiquement réglée à 1, toutes les valeurs suivantes sont incrémentées de 1, pour donner des valeurs résultantes comprises entre 1 et 7. Toute constante d'énumération peut être affectée d'une valeur d'entier dans la définition de l'énumération et toutes les constantes d'énumération subséquentes posséderont une valeur plus élevée que la constante précédente par une valeur de 1.

L'avantage d'exploiter un type d'énumération plutôt que de directement définir une valeur constantes est en rapport à la lisibilité du code source : il est beaucoup plus facile pour un programmeur de « lire » un code source contenant

```
Jour vidanges = MARDI;
```

que

```
int vidanges = 1;
```

## Règles de portée

21

- Portée** : accessibilité d'un identificateur (variable, prototype, fonction, fichier d'entête, ...)
- Un identificateur déclaré en dehors de toute fonction a une **portée de fichier** : accessible à partir de sa déclaration jusqu'à la fin du fichier
- Un identificateur déclaré à l'intérieur d'un bloc (`{ }`) a une **portée de bloc** : accessible à partir de sa déclaration jusqu'à la fin du bloc
  - Les paramètres d'une fonction ont une portée associée à tout le bloc de la fonction

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

La portion de programme où un identificateur est significatif représente sa *portée*. Par exemple, lorsque nous déclarons une variable locale dans un bloc, elle ne peut être consultée qu'à l'intérieur de ce bloc ou dans les blocs imbriqués au sein de ce bloc.

Les quatre portées d'un identificateur sont la *portée de fonction*, la *portée de fichier*, la *portée de bloc* et la *portée de prototype de fonction*. Il existe aussi une *portée de classe* qui sera étudiée au chapitre 6.

Les étiquettes de **case** et de **goto** (identificateurs suivis d'un deux-points, par exemple **depart :**) sont les seuls identificateurs à avoir une *portée de fonction*.

Les seuls identificateurs dotés d'une *portée de prototype de fonction* sont ceux que l'on utilise dans une liste de paramètres d'un prototype de fonction. Comme nous l'avons vu, le compilateur ignore les noms des paramètres dans ces prototypes, donc si on y déclare un identificateur, celui-ci peut être redéfini sans ambiguïté ailleurs dans le programme.

## Règles de portée : exemple

22

Portée de l'identificateur

```
#include <iostream>

void fonc1( int ); // Prototype

void fonc2( int v ) {
    int w = 2; // Variable locale
    std::cout << w * ++v;
}

int GLOBALVAR = 3; // Variable globale

int main() {
    int x = 5; // Variable locale
    fonc1( x );
    fonc2( x * GLOBALVAR );
    return 0;
}

void fonc1( int x ) {
    std::cout << x * 6;
    int y = 2; // Variable locale
    std::cout << x * y;
}
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois 25908 IFM - Langage C++

L'identificateur **fonc1** a une portée de fichier débutant au prototype jusqu'à la fin du fichier. Il en est de même avec l'identificateur **fonc2** dont la portée débute à la définition de la fonction. Cependant les variables **v** et **w** ont une portée limitée à la fonction **fonc2** (portée du bloc de la fonction).

La variable globale **GLOBALVAR** a aussi une portée de fichier, tout comme la fonction **main**. La variable **x** définie dans **main** a une portée de bloc correspondant au corps de la fonction **main**. Notez que la fonction **fonc1** dispose aussi d'une variable locale **x** ayant une portée correspondant au corps de la fonction **fonc1**. Même si **main** et **fonc1** disposent tous deux d'une variable locale **x**, celles-ci sont distinctes et le compilateur n'y voit aucune ambiguïté puisque ces deux variables ont des portées qui ne se recoupent pas.

## Classes de stockage

- Déterminent la période d'existence de l'identificateur (i.e. variable) en mémoire
  - Ne pas confondre avec le principe de classe en programmation orientée objet
  - Classes de stockage : **auto**, **register**, **static**, **mutable** et **extern**
    - **mutable** et **extern** seront étudiées plus tard
- Certaines classes de stockages impactent aussi la *portée* de l'identificateur

En plus de sa portée, un identificateur (p.ex. une variable ou une fonction) possède aussi une *classe de stockage* qui détermine la période durant laquelle l'identificateur existe en mémoire. Certains identificateurs existent brièvement, quelques-uns sont créés et détruits à maintes reprises, d'autres encore existent pendant toute l'exécution du programme.

Les spécifications de classes de stockage peuvent être divisées en deux catégories : classes de stockage *automatiques* et classes de stockage *statiques*. Les mots-clés **auto** et **register** sont utilisés pour déclarer des variables de classe de stockage automatique. De telles variables sont créées à l'entrée du bloc où elles sont déclarées; elles existent pendant que le bloc est en exécution et détruites lorsque l'exécution du bloc est terminée.

### Classes de stockage **auto** et **register**

- 24
- Classe **auto** (pour *automatique*) : classe de stockage par défaut pour les variables locales
 

```
{ auto int i; } équivalent à { int i; }
...
{ }
```
  - Classe **register** : suggère au compilateur de réserver un registre du microprocesseur pour la variable
    - Généralement pour accélérer le traitement d'un compteur
 

```
for ( register int i = 0; i < 10; i++ ) {
    ...
}
```
    - Le compilateur peut ne pas respecter cette directive

Auteur : Marco Lavoie | Adaptation : Sébastien Bois 25908 IFM - Langage C++

Seules les variables peuvent être de la classe de stockage automatique. Les paramètres et les variables locales d'une fonction sont par défaut de cette classe de stockage. C'est pourquoi le mot-clé **auto** est rarement utilisé.

Les données du programme compilé en langage machine doivent être chargées dans les *registres* du microprocesseur afin d'être manipulées. La spécification de classe de stockage **register** peut être placée avant la déclaration d'une variable automatique pour suggérer au compilateur de conserver la variable dans l'un des registres haute vitesse du microprocesseur plutôt qu'en mémoire vive. S'il est possible de conserver des variables à utilisation intensive, comme des compteurs ou des totaux, dans des registres, on peut alors éliminer la surcharge causée par les transferts répétitifs de ces variables entre les registres du microprocesseur et la mémoire vive de l'ordinateur.



## Classe de stockage `static`

25

- Classe `static` : variable globale à portée limitée
  - Variable globale : existe durant toute l'exécution du programme et conserve sa valeur
  - Portée limitée : accessible seulement dans le bloc où elle est déclarée
- Principale utilisation : *variable locale de fonction conservant sa valeur entre les invocation de la fonction*

Marco Lavoie

14728 ORD - Langage C++

Les mots-clés `extern` et `static` sont utilisés pour déclarer des identificateurs pour les variables et fonctions de classe de stockage statique. De telles variables existent dès le début de l'exécution du programme, et ce jusqu'à la fin de son exécution. Pour ces variables, le stockage n'est alloué et initialisé qu'une seule fois, au démarrage du programme (avant l'exécution de la première instruction de `main`).

Toutefois, même si ces variables et fonctions existent dès le début de l'exécution du programme, ces identificateurs ne pourront pas nécessairement être utilisés à travers tout le code source du programme car ceux-ci sont soumis aux *contraintes de portée*. Ainsi, les variables locales déclarées avec le mot-clé `static` ne sont encore connues qu'à l'intérieur du bloc (p.ex. corps de fonction) dans lequel elles sont définies.

## Classe de stockage `static` (suite)

26

### Exemple

```
int main() {
    for ( int i = 0; i < 10; i++ )
        fonce();

    std::cout << std::endl;

    return 0;
}
```

```
void fonce() {
    int v = 0;
    std::cout << v++ << ' ';
}
```

– Qu'affiche ce programme ?

```
C:\>fonce.exe
0 0 0 0 0 0 0 0 0 0
C:\>
```

```
int main() {
    for ( int i = 0; i < 10; i++ )
        fonce();

    return 0;
}
```

```
void fonce() {
    static int v = 0;
    std::cout << v++ << ' ';
}
```

– Qu'affiche ce programme ?

```
C:\>fonce.exe
0 1 2 3 4 5 6 7 8 9
C:\>
```

Marco Lavoie

14728 ORD - Langage C++

Contrairement aux variables automatiques, les variables locales statiques conservent leur valeur après l'exécution du bloc dans lequel elles sont définies. Lors d'un appel ultérieur de la fonction, les variables locales `static` possèdent les mêmes valeurs qu'au moment de la dernière sortie de la fonction.

Si une variable locale de classe de stockage statique est initialisée lors de sa déclaration, cette initialisation est appliquée uniquement lors de la première exécution du bloc où elle est définie. Lors des exécutions subséquentes du bloc, l'initialisation n'est pas appliquée afin que la variable conserve la valeur lui ayant été attribuée lors de l'exécution précédente du bloc.

Toutes les variables numériques de la classe de stockage statique sont initialisées à zéro si elles ne sont pas initialisées explicitement par le programmeur lors de leur déclaration.

## Fonctions inline

27

- L'invocation d'une fonction consomme des ressources (processeur et mémoire), ralentissant l'exécution
- **Exemple** : `ex2.cpp` plus rapide que `ex1.cpp` :

**ex1.cpp**

```
int cube( const int y ) {
    return y * y * y;
}

int main() {
    int somme = 0;

    for ( int i = 1; i <= 1000; i++ )
        somme += cube( i );

    std::cout << somme;

    return 0;
}
```

**ex2.cpp**

```
int main() {
    int somme = 0;

    for ( int i = 1; i <= 1000; i++ )
        somme += i * i * i;

    std::cout << somme;

    return 0;
}
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Mettre en œuvre un programme sous forme d'une série de fonctions est une bonne pratique du point de vue de la conception de logiciels, mais les appels de fonctions impliquent une surcharge du temps d'exécution. Le C++ offre les fonctions `inline`, qui aident à réduire la surcharge en mémoire des appels de fonctions, surtout pour les petites fonctions.



## Fonctions `inline` (suite)

28

- On peut "suggérer" au compilateur de remplacer les invocations d'une fonction par le code de cette fonction

```

inline int cube( const int y ) {
    return y * y * y;
}

int main() {
    int somme = 0;

    for ( int i = 1; i <= 1000; i++ )
        somme += cube( i );

    std::cout << somme;

    return 0;
}

```

→ Le compilateur transforme ce code à :

```

int main() {
    int somme = 0;

    for ( int i = 1; i <= 1000; i++ )
        somme += i * i * i;

    std::cout << somme;

    return 0;
}

```

Marco Lavoie

14728 ORD - Langage C++

Le qualificatif **inline** placé avant le type de renvoi d'une fonction dans la définition de fonction « recommande » au compilateur de produire une copie du code de la fonction en place (au moment opportun) pour éviter un appel de fonction. Ce compromis permet l'insertion dans le programme de multiples copies du code de la fonction en place de ses invocations, augmentant d'autant la taille du programme, plutôt qu'une copie unique de la fonction vers laquelle le contrôle est refilé à chaque invocation de celle-ci.

L'emploi de fonctions **inline** peut réduire le temps d'exécution d'un programme, mais peut aussi augmenter la taille de celui-ci.



## Fonctions `inline` (suite)

29

- Le compilateur va ignorer la directive `inline` si
  - Un des paramètres n'a pas l'attribut **const**
  - La fonction `inline` est trop longue
    - Généralement seules les fonctions de quelques lignes devraient être déclarées `inline`
  - Le code source contient beaucoup d'invocations à `inline`
    - La duplication du code de la fonction augmentera trop la taille du code exécutable

Marco Lavoie

14728 ORD - Langage C++

Le compilateur peut ignorer le qualificatif **inline** attribué à une fonction. En fait, c'est ce qu'il fait habituellement pour toutes les fonctions sauf les plus petites.

Le mot-clé **const** dans la liste de paramètres d'une fonction indique au compilateur que la fonction ne modifie pas la valeur du paramètre. Cela permet d'assurer que la valeur du paramètre n'est pas changée par la fonction. Le mot-clé **const** est présenté en détails dans les chapitres subséquents.



## Fonctions `inline` (suite)

30

- Bonne pratique de programmation
  - Associer l'attribut `inline` à une fonction seulement si elle peut être écrite sur une seule et même ligne

```

inline int cube( const int y ) { return y * y * y; }

int main() {
    int somme = 0;

    for ( int i = 1; i <= 1000; i++ )
        somme += cube( i );

    std::cout << somme;

    return 0;
}

```

Marco Lavoie

14728 ORD - Langage C++

Comme mentionné précédemment, la désignation **inline** « recommande » au compilateur de substituer les invocations de la fonction par le code source du corps de celle-ci. Pour limiter l'augmentation de taille du programme résultant, seules les fonctions à une ligne devraient être désignées **inline**.

## Variables références

31

- **Variable référence** : permet de déclarer un *alias* (i.e. autre nom) pour une variable existante

```
int main() {
    int a = 0, b = a;

    std::cout << a << " ";
    b = 1;
    std::cout << a << "\n";

    return 0;
}
```

```
C:\>exemple1.exe
0 0
C:\>
```

*b est un alias de la variable a*

```
int main() {
    int a = 0, &b = a;

    std::cout << a << " ";
    b = 1;
    std::cout << a << "\n";

    return 0;
}
```

```
C:\>exemple1.exe
0 1
C:\>
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

L'*appel par référence* et l'*appel par valeur* sont deux façons d'invoquer des fonctions dans de nombreux langages de programmation. Lorsqu'un argument est passé par un appel par valeur, une *copie* de la valeur de l'argument est passée à la fonction appelée. Les changements appliqués par la fonction à cette copie n'affectent ainsi en rien la valeur de la variable indiquée en argument. Cela permet d'éviter les effets de bord accidentels qui font grandement obstacle au développement de logiciels justes et fiables.

L'appel par référence requiert l'utilisation de l'*opérateur référence* (&). Cet opérateur permet de définir un alias de variable pour une variable existante, c'est-à-dire un deuxième nom pour une variable existante.

## Variables références (suite)

32

- Une variable référence peut changer d'alias en tout temps

```
int main() {
    int a = 0, b = 1, &r = a;

    std::cout << setw( 2 );
    std::cout << a << b << r << std::endl;

    r = 1;
    std::cout << a << b << r << std::endl;

    &r = b;
    r++;
    std::cout << a << b << r << std::endl;

    return 0;
}
```

```
C:\>exemple1.exe
0 1 0
1 1 1
1 2 2
C:\>
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Une fois que la variable référence est déclarée alias d'une autre variable, toutes les opérations qu'on croit effectuer sur l'alias (c'est-à-dire la référence) sont en fait exécutées sur la variable d'origine. L'alias ne représente qu'un autre nom de la variable d'origine.

## Paramètres de référence

33

- Un paramètre de fonction peut aussi être alias de l'argument d'invocation

```
// Transmutation des arguments
void swap( int &x, int &y ) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 10, b = 20;

    std::cout << setw( 3 );
    std::cout << a << b << std::endl;
    swap( (a), (b) );
    std::cout << a << b << std::endl;

    return 0;
}
```

- Les paramètres *x* et *y* sont dits *paramètres de référence*
- Lors de l'appel : `&x = a` et `&y = b`
- Prototype de la fonction :  
`void swap( int&, int& );`

```
C:\>ex_swap.exe
10 20
20 10
C:\>
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Avec l'appel par référence, l'appelant donne à la fonction appelée la possibilité d'accéder directement aux données de l'appelant et de modifier ces données si la fonction appelée le désire.

Un paramètre par référence est donc un alias de la variable correspondante spécifiée en argument lors de l'invocation de la fonction. Ensuite, le fait de mentionner le paramètre dans le corps de la fonction fait référence à la variable initiale de la fonction appelante; la variable initiale peut alors être modifiée directement par la fonction appelée. Ce comportement impose toutefois une contrainte à la fonction appelante : elle doit absolument fournir une variable correspondant à un paramètre référence, et non une constante ou une expression mathématique.

Comme toujours, le prototype de fonction et l'en-tête de fonction doivent concorder.

## Paramètres de référence (suite)

34

- L'exemple précédent ne serait pas fonctionnel avec des *paramètres de valeur*

```
// Transmutation des arguments
void swap( int x, int y ) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 10, b = 20;

    std::cout << setw( 3 );

    std::cout << a << b << std::endl;
    swap( a, b );
    std::cout << a << b << std::endl;

    return 0;
}
```

```
C:\>eg_swap.exe
10 20
10 20
C:\>
```

- Car les paramètres `x` et `y` sont locales à `swap()`
  - Donc aucun impact sur les arguments `a` et `b` dans `main()`

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

L'emploi de paramètres références (i.e. l'appel par référence) occasionne ce qu'on nomme communément un *effet de bord*, où l'invocation d'une fonction cause des modifications aux variables de la fonction appelante. Ces effets de bords sont généralement une façon pour une fonction de *retourner plus d'un résultat*.

Sans les paramètres références, la seule façon pour une fonction de retourner un résultat est via son type de retour, à l'aide de l'instruction `return`. Si la fonction doit cependant retourner plus d'un résultat, l'utilisation de l'instruction `return` n'est plus appropriée. L'emploi de paramètre références permet alors à la fonction de stocker des résultats directement dans les variables spécifiées par la fonction appelante lors de l'invocation de la fonction appelée. Dans l'exemple ci-contre, la fonction `swap` peut directement modifier le contenu des variables `a` et `b` de la fonction `main`.

## Arguments par défaut

35

- Une fonction peut définir des paramètres optionnels
  - Si l'invocation ne fournit pas d'argument correspondant, une *valeur par défaut* est utilisée

```
// Volume d'un cube
int volume( int lo = 1, int la = 1, int ha = 1 ) {
    return lo * la * ha;
}

int main() {
    int long, larg, haut;

    std::cout << "Dimensions? ";
    std::cin >> long >> larg >> haut;

    std::cout << volume() << std::endl
              << volume( long ) << std::endl
              << volume( long, larg ) << std::endl
              << volume( long, larg, haut ) << std::endl;

    return 0;
}
```

```
C:\>volume.exe
Dimensions? 5 3 4
1
5
15
60
C:\>
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Les appels de fonction peuvent couramment passer la valeur particulière d'un argument. Le programmeur peut spécifier qu'un tel argument est un *argument par défaut* et lui attribuer une valeur par défaut. Lorsque l'argument par défaut est omis dans un appel de fonction, la valeur par défaut de cet argument est insérée automatiquement par le compilateur et passée dans l'invocation.

## Arguments par défaut (suite)

36

- Restrictions d'utilisation #1

- Les valeurs par défaut doivent être spécifiées dans le prototype s'il vient avant la définition de la fonction

```
int volume( int = 1, int = 1, int = 1 );

int main() {
    int long, larg, haut;

    std::cout << "Dimensions? ";
    std::cin >> long >> larg >> haut;

    std::cout << volume() << std::endl
              << volume( long ) << std::endl
              << volume( long, larg ) << std::endl
              << volume( long, larg, haut ) << std::endl;

    return 0;
}

// Volume d'un cube
int volume( int lo, int la, int ha ) {
    return lo * la * ha;
}
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Il est obligatoire de spécifier les arguments par défaut avec la première occurrence du nom de la fonction, habituellement dans le prototype. Les valeurs par défaut peuvent être des constantes, des variables globales ou des appels de fonctions. On peut également utiliser les arguments par défaut avec des fonctions `inline`.

## Arguments par défaut (suite)

37

- Restriction d'utilisation #2
  - Si un paramètre de fonction dispose d'une valeur par défaut, tous les paramètres subséquents doivent en avoir une aussi
    - Sinon le compilateur ne saura pas comment associer les arguments aux paramètres

```
int fonc1(int a = 1, float b = 10.0, char c = 'a') ✓
int fonc2(int a, float b = 10.0, char c = 'a') ✓
int fonc3(int a, float b, char c = 'a') ✓
int fonc4(int a = 1, float b, char c = 'a') ✗
int fonc5(int a = 1, float b, char c) ✗
int fonc6(int a, float b = 10.0, char c) ✗
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Les arguments par défaut doivent être les arguments les plus à droite (de suite) dans une liste de paramètres de fonction. Lors de l'appel d'une fonction avec deux arguments par défaut ou plus, si un argument omis n'est pas l'argument le plus à droite dans la liste, tous les arguments situés à sa droite doivent être aussi omis.

## Exercice #3.3

38

- Solutionnez l'exercice distribué par l'instructeur
  - Créer un nouveau projet console *Visual Studio*
  - Solutionner le problème tel que décrit
  - N'oubliez pas les conventions d'écriture
  - Soumettez votre projet selon les indications de l'instructeur

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Puisque la fonction à implanter doit retourner plus d'un résultat (i.e. la racine carrée et la racine cubique du nombre donné), des paramètres références doivent être exploités par la fonction afin de lui permettre de modifier le contenu des variables arguments fournies par la fonction appelante (i.e. dans **main**) lors de l'invocation.

Puisqu'il est impossible de calculer la racine d'une valeur négative (p.ex. aucune valeur négative au carré pourrait donner -4), la fonction **facteurs** doit retourner une valeur de type **bool** : elle doit retourner **true** si elle a pu calculer les racines de l'argument **valeur**, **false** sinon). Ainsi, le programme principal peut déterminer (dans la condition du **if**) si **facteurs** a réussi à stocker les racines dans ses arguments **racine2** et **racine3**.

La librairie **<cmath>** dispose d'une fonction pour calculer la racine carrée d'une valeur (**sqrt**), mais ne dispose pas d'une telle fonction pour la racine cubique. Conséquemment vous devez utiliser la fonction **pow** pour calculer la racine cubique (i.e. calculer  $x^{1/3}$ ).

## Surcharge de fonction

39

- Le C++ permet à plusieurs fonctions d'avoir le même identificateur (i.e. même nom)
  - En autant que chacune ait un prototype distinct
  - Cette capacité s'appelle la *surcharge de fonction*
- Lors de l'invocation, le compilateur sélectionne la fonction appropriée en fonction des arguments

On utilise couramment la surcharge d'une fonction pour créer plusieurs fonctions portant le même nom qui effectuent des tâches similaires mais sur des types de données différentes.

On distingue les fonctions surchargées par *leurs signatures*; une signature combine le nom de fonction et ses types de paramètres. Le compilateur code chaque identificateur de fonction avec le nombre et les types de ses paramètres pour permettre la *liaison à vérification de type*. Lors de l'invocation d'une fonction disposant de plusieurs surcharges, le compilateur identifie la surcharge ayant une signature conforme à l'invocation et fait le lien en conséquence lors de l'exécution du programme.

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

## Surcharge de fonction : exemple

- Le compilateur identifie la fonction à exécuter selon

- Le nombre d'arguments
- Le type de chaque argument
- Important : les deux fonctions doivent avoir une signature distincte

```
// Carré d'un entier
int carre( int x ) {
    return x * x;
}

// Carré d'un flottant
double carre( double x ) {
    return x * x;
}

int main() {
    std::cout << carre( 7.5 ) << std::endl;
    std::cout << carre( 7 ) << std::endl;

    return 0;
}

int carre( int );
double carre( double );
```

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Dans l'exemple ci-contre, la fonction surchargée **carre** calcule le carré d'un **int** et le carré d'un **double**. Lors des invocations de la fonction dans `main`, le compilateur est apte à faire la liaison entre chaque invocation et la surcharge correspondante grâce aux signatures de fonctions et d'invocation.

Même si deux fonctions portent le même nom d'identificateur (i.e. **carre**), il n'y a aucune confusion puisque ces deux fonctions ont une signature distincte.

## Microsoft Visual Studio

- Le débogueur de *Visual Studio*
  - Exécution pas à pas du code source
    - Permet de suivre l'évolution de l'exécution
    - Permet d'inspecter le contenu des variables après l'exécution de chaque instruction
      - Et de changer le contenu de celles-ci
  - Outil indispensable pour corriger les erreurs de logique dans un projet C++
- La plupart des environnements de développement (e.g. *Eclipse*) offrent un débogueur

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

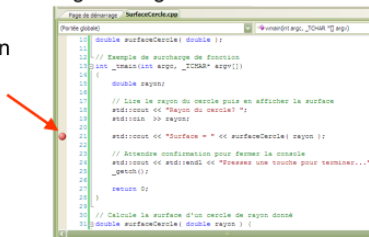
25908 IFM - Langage C++

Beaucoup de programmeurs, principalement débutants, n'utilisent très peu voire pas du tout cet indispensable outil qu'est le débogueur. Et ceux qui l'utilisent passent bien souvent à côté des fonctionnalités les plus intéressantes. Non pas parce que son utilisation est difficile, bien au contraire, mais parce qu'on ne connaît bien souvent pas l'étendue des possibilités qu'il nous offre.

*Visual Studio* fournit l'un des débogueurs les plus performants du marché, possédant beaucoup de fonctionnalités, et surtout très intuitif à utiliser. Nous présentons ici les principales fonctionnalités de ce débogueur. Vous apprendrez rapidement à maîtriser cet outil en l'utilisant fréquemment, comme le font les programmeurs d'expérience.

## Visual Studio : Point d'arrêt

- Pour activer le débogueur, il faut premièrement définir un point d'arrêt (*breakpoint* en anglais)
  - En cliquant dans la marge à la ligne d'arrêt désirée
  - Ou en pressant la touche **F9** afin de basculer le point d'arrêt à la ligne au curseur
- On peut activer plus d'un point



Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Les points d'arrêt (*breakpoints*) sont un élément essentiel du débogueur : à chaque fois que le débogueur en rencontre un, il met en pause l'exécution à cet endroit. C'est très pratique notamment pour inspecter la valeur de certaines variables à un moment précis, ou encore pour s'assurer que le programme passe bien par certains endroits.

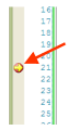
Pour placer un point d'arrêt, il suffit de cliquer sur la ligne choisie, dans la colonne située à gauche du code. Vous pouvez également utiliser le raccourci **F9**. Vous verrez un point rouge apparaître : votre point d'arrêt est maintenant placé. Notez que *Visual Studio* peut déplacer vos points d'arrêt, notamment si ceux-ci se trouvent sur des portions ne correspondant pas à des instructions (un commentaire par exemple).



## Visual Studio : Atteinte du point d'arrêt

43

- L'exécution est suspendu à chaque point d'arrêt atteint
  - La flèche jaune indique la prochaine instruction à être exécutée



```

14 // Lire le rayon du cercle puis en afficher la surface
15 std::cout << "Rayon du cercle? ";
16 std::cin >> rayon;
17
18 std::cout << "Surface = " << surfaceCercle( rayon );
19
20 // Attendre confirmation pour fermer la console
21 std::cout << std::endl << "Appuyez une touche pour terminer..." << std::endl;
22 _getch();
23

```




- Le mode d'exécution pas à pas est alors activé

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

## Visual Studio : Exécution pas à pas

44

- Permet d'exécuter une instruction à la fois
  - **Pas à pas détaillée** (  ) : si l'instruction invoque une fonction, celle-ci est exécutée pas à pas
  - **Pas à pas principal** (  ) : si l'instruction invoque une fonction, celle-ci est exécutée en continu
    - Le pas à pas est réactivé une fois la fonction exécutée
  - **Pas à pas sortant** (  ) : exécuter en continue la fonction courante jusqu'à sa complétion
    - Le pas à pas est réactivé à la sortie de la fonction

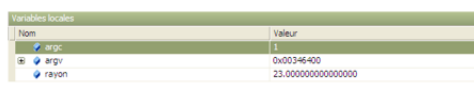
Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

## Visual Studio : Inspection des variables

45

- Le contenu des variables est affiché
  - en plaçant la souris sur la variable visée, ou
  - dans la fenêtre d'inspection des variables



Variables locales	
Nom	Valeur
argc	1
argv	0x00346400
rayon	23.000000000000000000

- La valeur d'une variable peut y être modifiée en double cliquant sur le champ de la colonne Valeur

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Lancez l'exécution en mode débogage : si tout se passe bien, votre programme va s'arrêter sur le premier point d'arrêt rencontré. Vous pouvez maintenant si vous le souhaitez inspecter la pile des appels à cet endroit, ou encore la valeur de variables locales. Vous pourrez ensuite continuer l'exécution jusqu'au prochain point d'arrêt, ou jusqu'à la fin s'il n'y en a plus.

A noter que si vous vous perdez dans les points d'arrêt, l'option du menu Déboguer->Supprimer tous les points d'arrêt (Ctrl+Maj+F9) permet de retirer tous les points d'arrêt du programme.

Parfois, il arrive que l'on ait besoin de voir comment se comporte un programme ligne par ligne. Pas question de poser des points d'arrêt partout, bien sûr. *Visual Studio* propose un mode d'exécution *pas à pas*, qui permet d'avancer d'une (ou plus) instruction à la fois, vous permettant ainsi d'observer en détail le parcours du programme, ainsi que de suivre l'évolution de quelques variables si vous le souhaitez.

L'exécution pas à pas peut être démarrée après l'arrêt du programme sur un point d'arrêt, mais également directement à partir du menu Déboguer->Pas à pas détaillé (F11). L'exécution pas à pas se poursuit également avec F11, ou avec F10 (Pas à pas principal). F11 va s'engouffrer dans le code, et entrer dans les appels de fonctions par exemple, alors que F10 ne va jamais quitter la fonction courante même s'il croise un autre appel de fonction.

Vous pouvez également démarrer le pas à pas à partir d'un certain endroit du code, soit en posant un point d'arrêt, soit en utilisant la commande du clic droit Exécuter jusqu'au curseur (Ctrl + F10).

Une première fonctionnalité primordiale du débogueur, est la possibilité de scruter la valeur des variables, et ce à tout moment. Il existe plusieurs façons de visualiser ces valeurs, lorsque le programme est stoppé en mode débogage.

La première manière de visualiser une variable est de simplement placer le pointeur de la souris sur celle-ci : une *info bulle* vous indiquera sa valeur. Pour les structures ou les tableaux, vous pouvez même dérouler et inspecter plus en profondeur éléments, et ce sur plusieurs niveaux de profondeur.

Si vous souhaitez suivre la valeur d'une variable sans avoir à pointer dessus, ou si vous voulez évaluer des expressions plus complexes, alors vous pouvez utiliser les *fenêtres espions* (Watch windows) prévues à cet effet. Si elles ne sont pas visibles par défaut, vous pouvez les afficher via le menu Déboguer->Fenêtres->Espion.



## Erreurs de programmation

46

- Oublier d'inclure le fichier en-tête requis
- Oublier de spécifier le type de la valeur de retour d'une fonction (spécifiez `void` si aucune valeur n'est retournée par votre fonction)
- Oublier de retourner une valeur lorsque la fonction spécifie un type de valeur de retour
- Oublier les `()` lors de l'invocation d'une fonction
- Définir une fonction à l'intérieur d'une autre fonction
- Oublier de définir la fonction (ou son prototype) avant son invocation
  - Et le prototype doit correspondre à la signature de la fonction

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

En C++, les `()` dans un appel de fonction sont en fait un opérateur qui provoque l'appel de la fonction. Oublier les `()` dans un appel de fonction qui ne requiert pas d'arguments n'est pas une erreur de syntaxe, mais la fonction n'est pas invoquée.



## Erreurs de programmation (suite)

47

- Confondre une variable locale avec une variable globale du même nom
  - D'où l'importance de respecter les conventions d'écriture (variables globales en lettres majuscules)
- Confondre un paramètre de référence avec un paramètre de valeur
- Ne pas définir adéquatement les arguments par défaut d'une fonction
- Surcharger une fonction avec une autre ayant la même signature
  - Changer uniquement le nom des paramètres ne produit pas une signature distincte

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Confondre l'utilisation d'une variable locale avec une variable globale du même nom (ou toute variable de même nom dont la portée recoupe celle de la variable locale) constitue généralement une erreur de logique qui occasionne des effets de bords inattendus et difficiles à identifier. C'est pourquoi il est important de s'assurer que deux variables ayant une portée commune n'aient pas le même identificateur.



## Bonnes pratiques de programmation

48

- Toujours fournir un prototype en début de fichier pour les fonctions définies dans celui-ci
- Utiliser uniquement des majuscules pour le nom des constantes, incluant celles dans les énumérations
- Utiliser le qualificatif `inline` uniquement pour les fonctions courtes (une ou deux lignes)
- Exploiter judicieusement les arguments par défaut
- Éviter d'utiliser des variables de même nom dans les différents blocs du programme

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

Il est important de se familiariser rapidement avec la riche collection de fonctions et de classes offertes par la bibliothèque standard du C++. Le programmeur débutant va occasionnellement implanter une fonction sans savoir qu'il en existe déjà une de la bibliothèque qui offre la fonctionnalité visée.

Il est aussi inutile de récrire des fonctions existantes de la bibliothèque standard afin de les rendre plus efficaces. Dans la plupart des cas, vous ne pourrez pas améliorer leur performance.

## Devoir #2

49

- Solutionnez le problème distribué par l'instructeur
  - Créer un nouveau projet console *Visual Studio*
  - Solutionner le problème tel que décrit, avec la spécification supplémentaire suivante
    - À mesure que la partie progresse, affichez un message de bavardage avec une probabilité de 30% (i.e. 70% des itérations n'affichent aucun message de bavardage)
  - N'oubliez pas les conventions d'écriture
  - Respectez l'échéance imposée par l'instructeur
  - Soumettez votre projet selon les indications de l'instructeur
    - **Attention** : respectez à la lettre les instructions de l'instructeur sur la façon de soumettre vos travaux, *sinon la note EC sera attribuée à ceux-ci*

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

La meilleure façon de solutionner ce devoir est de transférer le code source du programme principal (i.e. fonction **main**) dans une nouvelle fonction, appelée par exemple **lancerBarbotte**, qui joue une partie de barbotte, c'est-à-dire du premier lancer des dés jusqu'à la détermination du victorieux. Votre nouvelle fonction peut retourner un booléen indiquant si le joueur a gagné. Son prototype sera alors :

```
bool lancerBarbotte();
```

La fonction **main** de votre programme n'aura alors qu'à s'occuper de gérer les fonds, demander le montant misé par le joueur à chaque partie et invoquer la fonction **lancerBarbotte** à chaque mise, et ce jusqu'à ce que le joueur ait perdu tous ses fonds ou qu'il ait décidé de quitter la partie.

## Pour la semaine prochaine

50

- Vous devez relire le contenu de la présentation du chapitre 3
  - Il y aura un **quiz** sur ce contenu au prochain cours
    - À livres et ordinateurs fermés

Auteur : Marco Lavoie | Adaptation : Sébastien Bois

25908 IFM - Langage C++

En début de classe la semaine prochaine vous aurez à répondre à des questions sur le C++ sans consultation du matériel pédagogique. Vous êtes donc fortement encouragé à relire les notes de cours du chapitre 3. Profitez-en aussi pour réviser le contenu des chapitres précédents.