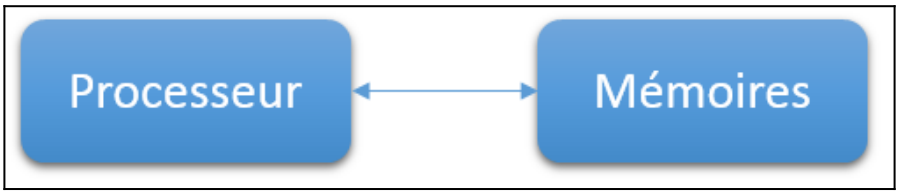
Utiliser la mémoire avec les variables

La mémoire et les variables

Très schématiquement, un ordinateur peut être décomposé en deux éléments :

- le processeur, qui réalise les calculs et les opérations logiques ;
- les mémoires, qui contiennent les informations (programmes à exécuter, données à traiter, résultats des calculs).



On distingue les mémoires de stockage sur le long terme (comme les disques durs) et les mémoires de travail à court terme (mémoire vive ou RAM, *Random Access Memory*). Cette dernière est utilisée pour conserver les données et les résultats des calculs des programmes.

En C++, les données sont manipulées en utilisant des variables. Ces variables peuvent être utilisées pour réaliser des calculs et diverses opérations logiques. Chaque variable dans un programme est désignée par un nom unique, appelé identifiant.

Créer une variable

Pour commencer, voyons un exemple de code utilisant une variable :

main.cpp

```
#include <iostream>

int main() {
    int i { 123 };                // création de i
    std::cout << i << std::endl;  // utilisation de i
}
```

affiche :

```
123
```

Ce code permet de créer une variable appelée `i`, qui peut contenir un nombre entier (`int` correspond à “integer”, qui signifie “entier”) et qui est initialisée avec la valeur `123`. Cette variable `i` est ensuite affichée en utilisant `std::cout`.

Lors de l'exécution de ce programme, `std::cout` ne va pas afficher le caractère `i`, mais la valeur contenue dans la variable `i`.

Faites bien attention à la syntaxe, même si un nom de variable est composé de caractères, ce n'est pas une littérale caractère ou une littérale chaîne de caractères :

```
std::cout <<  i  << std::endl;  // variable i
std::cout << 'i' << std::endl;  // caractère 'i', entre guillemets simples
std::cout << "i" << std::endl;  // chaîne de caractères "i", entre guillemets doubles
```

En pratique, la confusion est rare, la majorité des éditeurs de code utilisent des couleurs différentes pour les variables et les littérales.

La syntaxe générale pour créer une variable peut être résumée par la syntaxe suivante :

```
TYPE IDENTIFIANT { VALEUR };
```

Pour créer une variable, vous devez donc donner plusieurs informations, dans l'ordre :

- un **type** (par exemple `int` dans le code précédent) ;
- un **identifiant** (par exemple `i` dans le code précédent) ;
- une **valeur** (par exemple `123` dans le code précédent).

Vocabulaire

- On **déclare** un identifiant.
- On **définit** une variable avec un type et un identifiant.
- On **initialise** une variable avec une valeur.

Vous pouvez créer autant de variables que vous le souhaitez dans vos programmes (en fonction des capacités de votre ordinateur. Mais même un ordinateur de bureau basique de nos jours peut contenir sans problème plusieurs milliards d'entiers en mémoire) :

main.cpp

```
#include <iostream>

int main() {
    int x { 123 };
    int y { 456 };
    int z { 789 };
    std::cout << x << std::endl;
    std::cout << y << std::endl;
    std::cout << z << std::endl;
}
```

affiche :

```
123
456
789
```

Nous allons voir en détail chaque élément de la définition et l'initialisation d'une variable.

Syntaxes alternatives
Il existe en réalité plusieurs syntaxes possibles pour créer une variable. Voici quelques exemples :

```
int x;          // (1)
int x = 123;    // (2)
int x(123);     // (3)
auto x = 123;   // (4)
```

- (1) permet de créer une variable sans l'initialiser. Cette syntaxe est moins sûre que la syntaxe avec initialisation et ne sera pas utilisée dans ce cours.
- (2) et (3) sont des anciennes syntaxes qui sont encore très utilisées, mais n'apportent rien par rapport à la syntaxe utilisée dans ce cours (au contraire, dans certains cas, elles peuvent être ambiguës).
- (4) est appelée *inférence de type* et sera étudiée dans le prochain chapitre.

Vous rencontrez probablement ce type de syntaxe dans des codes existants, par exemple dans des tutoriels en ligne ou dans des livres. Ce cours se focalise sur les syntaxes recommandées en C++ moderne, ces syntaxes ne seront donc pas détaillées par la suite. Mais vous apprendrez sans problème ces syntaxes dans les exercices d'apprentissage que vous réaliserez.

Modifier la valeur d'une variable

L'intérêt d'une variable est que vous allez pouvoir la réutiliser dans des expressions. A chaque fois qu'une expression contenant une variable est évaluée, la variable est remplacé par sa valeur lors du calcul.

main.cpp

```
#include <iostream>

int main() {
```

```
    int x { 123 };
    int y { 456 };
    std::cout << x * 2 << std::endl; // affiche le résultat
du calcul 123 * 2
    std::cout << x + y << std::endl; // affiche le résultat
du calcul 123 + 456
}
```

affiche :

```
246
579
```

Une expression peut également être utilisée pour initialiser une autre variable.

main.cpp

```
#include <iostream>

int main() {
    int x { 123 };
    int y { x * 2 };
    std::cout << y << std::endl; // affiche le résultat du
calcul 123 * 2
}
```

affiche :

```
246
```

Une variable permet donc de retenir le résultat d'un calcul complexe, qui serait pénible de devoir réécrire plusieurs fois.

Modifier une variable

Une variable est définie par un type, un identifiant et une valeur. Même si pour être rigoureux, il faut dire “modifier la valeur d'une variable”, on simplifie souvent en disant “modifier une variable”. Il n'y a pas de confusion possible, puisque le type et l'identifiant d'une variable ne peuvent être définis que lors de la création et ne plus être modifié

ensuite.

Il est également possible de modifier la valeur d'une variable, en utilisant l'opérateur d'affectation `=`. La syntaxe est la suivante :

```
IDENTIFIANT = VALEUR;
```

En pratique, cela donne :

main.cpp

```
#include <iostream>

int main() {
    int x { 123 };
    std::cout << x << std::endl;
    x = 456;
    std::cout << x << std::endl;
}
```

affiche :

```
123
456
```

Confusion possible

Attention à ne pas confondre l'opérateur d'affectation pour modifier une variable `=` avec l'opérateur de comparaison d'égalité `==`.

De plus, dans les syntaxes alternatives, il est possible d'écrire le code suivant pour créer une variable.

```
int x = 123; // initialisation
    x = 456; // affectation
```

Notez la différence : une initialisation contient le type puis l'identifiant d'une variable, une affectation contient uniquement l'identifiant.

Pour éviter la confusion, il est recommandé d'utiliser la syntaxe avec des accolades pour l'initialisation et celle avec `=` pour l'affectation.

Dans de nombreux cas, vous n'aurez pas besoin de modifier la valeur d'une variable. Dans ce cas, on parle de constante. Pour indiquer cela dans le code, vous pouvez utiliser le mot-clé `const` (*constant*) devant le type de la variable lors de l'initialisation. De plus, cela permet au compilateur de vérifier que vous ne modifier effectivement pas cette variable et de réaliser certaines optimisations.

main.cpp

```
#include <iostream>

int main() {
    const int x { 123 };
    x = 456; // erreur
}
```

affiche :

```
main.cpp: In function 'int main()':
main.cpp:5:7: error: assignment of read-only variable 'x'
     x = 456; // erreur
       ^
```

qui peut se traduire par “affectation sur une variable en lecture seule”.

Il est important d'utiliser `const` aussi souvent que possible, c'est-à-dire à chaque fois que vous ne modifiez pas une variable. Dans la suite de ce cours, nous utiliserons systématiquement `const` dans les codes d'exemple.

Position de `const`

Le mot-clé `const` peut se placer avant ou après le type. Selon sa position, cela peut changer l'interprétation de ce qui est constant et de ce qui ne l'est pas. Mais dans les cas simples présentés ici, la position ne change rien à la signification de `const`. Les codes d'exemple utiliseront indifféremment les deux écritures.

```
const int i { 123 };
int const j { 456 };
```

La position de `const` sera importante dans certains cas, comme par exemple les pointeurs, qui seront vus à la fin de ce cours.

Le type d'une variable

Vous avez déjà rencontré la notion de type dans les chapitres précédents :

```
2    // littérale entière
2.0  // littérale réelle
'2'  // littérale caractère
"2"  // littérale chaîne
```

Chaque littérale précédente possède un type défini (“entier”, “réel”, “caractère”, “chaîne”, mais il y en a beaucoup d'autres). C'est également le cas avec les variables, elles possèdent toutes un type défini, qui ne peut pas être changé (seule la valeur qu'elles contiennent peut changer).

En C++, les types de base s'écrivent avec des mots-clés définis dans le langage. Vous avez vu que le type `int` correspond aux entiers. Il existe beaucoup de types définis en C++ (et il est possible de définir ses propres types, il peut donc potentiellement exister une infinité de types différents), mais retenez pour le moment les types correspondants aux littérales que vous avez déjà manipulées :

- `int` (abréviation de *integer*, “entier” français) correspond à un nombre entier ;
- `double` correspond à un nombre réel (vous verrez par la suite pourquoi le C++ utilise ce terme) ;
- `std::string` correspond aux chaînes de caractères ;
- `char` correspond à un caractère ;
- `bool` correspond aux booléens.

Il existe également des mots-clés permettant de modifier un type de base. Par exemple, `signed` et `unsigned` permettent respectivement de spécifier un type **entier** signée (qui accepte des valeurs négatives et positives) et non-signée (qui acceptent uniquement des valeurs

positives).

```
const int i { 123 };            // signed par défaut
const signed int j { 123 };     // signed explicite
const unsigned int k { 123 };   // unsigned
const unsigned int l { -123 };  // erreur, littérale signed
dans une variable unsigned
```

Pour rappel, voici comment s'écrivent les littérales correspondant à chaque type :

- pour un `int` : par exemple `123` ou `456` ;
- pour un `double` : par exemple `123.456` ou `123.456e789` ;
- pour un `std::string` : par exemple `“hello, world!”` ou `“bonjour!”` ;
- pour un `char` : par exemple 'a' ou 'z' ;
- pour un `bool` : uniquement `true` ou `false`.

Conversion de types

Essayons de voir ce qui se passe si on utilise un type de littérale différent du type de variable (conversion).

main.cpp

```
int main() {
    int a { 1 };    // [1]
    int b { 1.2 };  // [2]
    int c { '1' };  // [3]
    int d { "1" };  // [4]
}
```

Ce code va produire les erreurs suivantes :

```
main.cpp:3:13: error: type 'double' cannot be narrowed to
'int' in initializer list [-Wc++11-narrowing]
    int b { 1.2 };  // [2]
            ^~~
```

```
main.cpp:3:13: note: insert an explicit cast to silence this
issue
    int b { 1.2 };  // [2]
            ^~~
            static_cast<int>( )
main.cpp:3:13: warning: implicit conversion from 'double' to
'int' changes value from 1.2 to 1 [-Wliteral-
conversion]
    int b { 1.2 };  // [2]
          ~ ^~~
main.cpp:5:13: error: cannot initialize a variable of type
'int' with an lvalue of type 'const char [2]'
    int d { "1" };  // [4]
            ^~~
1 warning and 2 errors generated.
```

Prenons chaque ligne en détail et les erreurs produites.

Pas de conversion

La ligne [1] initialise une variable de type `int` a partir d'une littérale de type `int`. Dans ce cas, pas de problème, les types correspondent parfaitement.

Conversion avec arrondi

Le ligne [2] produit plusieurs messages.

Un message d'erreur “type 'double' cannot be narrowed to 'int'” (“le type 'double' ne peut pas être restreint en type 'int'”) indique que la conversion de types peut produire une perte d'information, c'est a dire que le type `double` peut contenir des valeurs que le type `int` ne peut pas contenir.

Une note permet d'aider le développeur a corriger ce problème : “insert an explicit cast to silence this issue” (“insérer une conversion explicite pour faire taire ce problème”).

Le message suivant est un avertissement : “implicit conversion (...) changes value from 1.2 to 1” (“la conversion implicite change la valeur 1.2 en 1”). Sans surprise, puisque les types ne sont pas directement

convertibles sans perte potentielle d'information, les valeurs doivent être arrondies (dans ce cas “1.2” en “1”).

Conversion implicite

La ligne [3] est beaucoup plus surprenante. Elle ne produit pas de message d'erreur ! Cependant, si on affiche la valeur de la variable `c`, le résultat est encore plus surprenant.

main.cpp

```
#include <iostream>

int main() {
    int c { '1' };  // [3]
    std::cout << c << std::endl;
}
```

affiche :

```
49
```

En fait, pour des raisons historiques, le type `char`, qui représente un caractère, est considéré comme un type entier et peut donc être converti automatiquement par le compilateur (conversion implicite). La valeur 49 correspond au caractère `1` dans la norme ASCII.

Généralement, la conversion de `char` en `int` ne posera pas de problème, mais dans d'autres cas, ce type de conversion implicite peut réellement produire des comportements non prévus par le développeur et être assez difficile a identifier et corriger.

Conversion impossible

La ligne [4] produit un message d'erreur plus simple : “cannot initialize a variable of type 'int' with an lvalue of type 'const char [2]'” (impossible d'initialiser une variable de type `int` avec une lvalue de type `const char [2]`). Le compilateur ne sait pas convertir une chaîne de caractères en entier et le signale.

Cela peut sembler ennuyeux que le compilateur bloque le processus, on

aimerait parfois qu'il se débrouille pour trouver une solution et réussisse toujours a créer le programme. Mais il faut bien comprendre que c'est en fait une aide, pas une punition. Il est préférable que le compilateur dise “je ne sais pas, aide moi”, plutôt que suivre un comportement que le développeur n'a pas prévu.

Typage fort

Avoir un contrôle sur les types par le compilateur permet de garantir leur utilisation correcte (*type safety*). Plus la prise en compte des types est importante, plus vous aurez de garantie sur le code. Ce typage fort est une des forces du C++ et il est intéressant de permettre au compilateur de faire un maximum de vérifications.

Dans tous les cas, il est important d'accorder une attention particulière aux types des variables et des données dans vos codes C++.

L'identifiant

L'identifiant d'une variable est le nom de cette variable. Vous pouvez utiliser cet identifiant dans vos codes en remplacement d'une valeur dans un calcul par exemple. Si vous utilisez plusieurs variables, chaque identifiant doit être unique, vous ne pouvez pas définir plusieurs variables utilisant le même nom :

```
#include <iostream>

int main() {
    int const x { 123 }; // x correspond à un entier
    int const x { 456 }; // erreur : l'identifiant x est
déjà utilisé
}
```

affiche :

```
main.cpp: In function 'int main()':
main.cpp:6:15: error: redeclaration of 'const int x'
     int const x { 456 }; // erreur : l'identifiant x est
déjà utilisé
```

```
                ^
main.cpp:5:15: note: 'const int x' previously declared here
     int const x { 123 }; // x correspond à un entier
                ^
```

Pour écrire un identifiant, vous pouvez utiliser les caractères alphanumériques minuscules et majuscules (`a` à `z`, `A` à `Z` et `0` à `9`) et le tiret bas `_` (*underscore*, correspond à la touche 8 sur un clavier français). De plus, un identifiant doit obligatoirement commencer par une lettre.

Par exemple, les noms suivants sont des identifiants valides :

- `x` ;
- `y` ;
- `unevariable` ;
- `uneVariable` ;
- `une_variable` ;
- `UnEvArIaBlE`.

En revanche, les identifiants suivants ne sont pas valides ou sont déconseillés :

- `_une_variable` : commence par un tiret bas ;
- `123variable` : commence par un chiffre ;
- `variable_réelle` : contient un caractère interdit (`é`).

Un identifiant qui commence par un tiret bas, comme `_une_variable`, est en fait autorisé par la norme C++, mais son usage est restreint. Pour éviter les problèmes, il est d'usage de ne pas utiliser ce type d'identifiant.
Comme vous le voyez, le langage C++ laisse de grandes libertés pour choisir un identifiant... ce qui peut poser des problèmes. Exemple de mauvais identifiant :

- `jjfndsfkjgukzv` : ne veut rien dire, n'apporte pas d'information sur le rôle de cette variable ;
- `une_variable` : trop générique, ne dit pas quel est le rôle de cette

variable ;
- `variable1`, `variable2`, etc. : trop générique aussi ;
- `UnEvArIaBlE` : le mélange de minuscule et majuscules rend ce nom peu lisible ;
- `une_variable_qui_contient_le_resultat_du_premier_calcul` : trop long.

Dans les projets réalisés en équipe, une bonne pratique est de définir des règles (“bonnes pratiques de codage”) qui s'imposent à tous les développeurs du projet, pour faciliter la lecture du code. Le but est d'avoir des noms homogènes, simples et informatifs.

Il existe déjà des règles de bonnes pratiques de codage toutes faites, mais vous pouvez aussi utiliser vos propres règles. Les conventions de nommage les plus connues écrivent les noms en minuscules séparées par un tiret bas (`une_variable`, par exemple dans la bibliothèque standard ou Boost) ou en écrivant les noms avec des majuscule au début de chaque mot et sans séparateur (`uneVariable`, par exemple dans la bibliothèque Qt).

Le nom des variables participe à la qualité d'un code. Plus le nom d'une variable est informatif sur le rôle de cette variable, moins vous aurez de risque de mal utiliser cette variable.

La valeur

Une variable contient obligatoirement une valeur. Il est possible de définir une variable sans l'initialiser, mais cette variable pourra alors contenir une valeur aléatoire. Cependant, vous imaginez bien qu'un programme ne va pas forcément fonctionner correctement si certaines variables sont initialisées avec des valeurs non déterminées.

Une variable peut être initialisée avec une valeur par défaut (*value initialization*), avec une littérale (*direct initialization*) ou avec une expression (*copy initialization*).

- initialisation par défaut : `Type Identifiant {};` ;

- initialisation avec une littérale : `Type Identifiant { Valeur };` ;
- initialisation avec une expression : `Type Identifiant { Expression };`.

Plus concrètement, avec du code :

main.cpp

```cpp
#include <iostream>
#include <string>

int main() {
    // Initialisation par défaut
    const int         i_default {};
    const double      d_default {};
    const std::string s_default {};
    const char        c_default {};
    const bool        b_default {};

    // Initialisation avec une valeur
    const int         i_default { 123 };
    const double      d_default { 123.456 };
    const std::string s_default { "hello, world!" };
    const char        c_default { 'a' };
    const bool        b_default { true };

    // Initialisation avec une expression
    const int         i_default { 123 + 456};
    const double      d_default { 12.34 + 56.78 };
    const std::string s_default { std::string{ "hello, " } +
"world!" };
    const char        c_default { 'a' };
    const bool        b_default { 123 > 456};
}
```

Note : l'expression avec `std::string` est un peu plus complexe, du fait de certaines règles de manipulation des chaînes de caractères. Vous verrez cela plus en détail dans le chapitre sur les chaînes.

Portée d'une variable

Une variable existe à partir du moment où elle est déclarée et jusqu'à la fin du bloc contenant sa déclaration. Cela s'appelle la portée de la variable. Utiliser une variable avant de l'avoir définie produit donc une erreur.

main.cpp

```
#include <iostream>

int main() {
    std::cout << x << std::endl;
    const int x { 123 };
}
```

affiche :

```
main.cpp: In function 'int main()':
main.cpp:4:18: error: 'x' was not declared in this scope
     std::cout << x << std::endl;
                  ^
```

Ce qui signifie “x n'est pas déclarée dans cette portée”.

Il est également possible d'ajouter des blocs supplémentaires, pour limiter la portée des variables.

main.cpp

```
#include <iostream>

int main() {
    const int x { 123 };              // début de portée de
x    //
    {                                 //
//
        const int y { x + 456 };      //
// début de portée de y
        std::cout << x << std::endl;  // ok, x existe
//
        std::cout << y << std::endl;  //
```

```
// ok, y existe
    }                                   //
// fin de portée de y
    std::cout << x << std::endl;        // ok, x existe
//
    // std::cout << y << std::endl;     //
// erreur, y est hors de portée
}                                       // fin de portée de x
//
```

Note : dans la partie sur les identifiants, il est dit qu'un identifiant devait être unique. En fait, la règle est plus précisément “il ne faut pas avoir deux identifiants identiques dans la même portée”. Il est donc possible d'avoir plusieurs variables portant le même identifiant, si leur portée est différente.

main.cpp

```cpp
#include <iostream>

int main() {
    {
        const int x { 123 };
        std::cout << x << std::endl;
    }
    {
        const int x { 456 };
        std::cout << x << std::endl;
    }
}
```

Il faut bien comprendre ce qui se passe ici. Une première variable nommée `x` est créée dans le premier bloc et initialisée avec la valeur 123. Puis celle-ci est détruite à la fin du bloc et une nouvelle variable est créée. Cette variable s'appelle aussi `x`, mais c'est bien une variable différente.

Chapitre précédent	Sommaire principal	Chapitre suivant