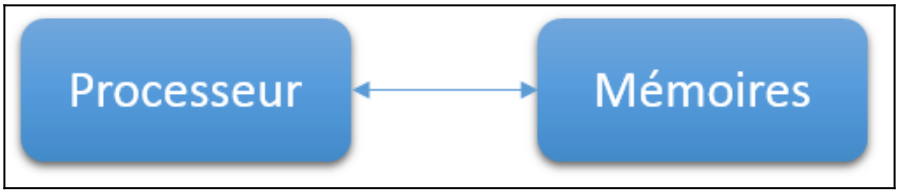
Utiliser la mémoire : les variables

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 (comme les disques durs) et les mémoires de travail (mémoire vive ou RAM *Random Access Memory*). Cette dernière est utilisée pour conserver les données et les résultats des calculs des programmes.

Pour manipuler les données en mémoire en C++, vous allez utiliser des variables. Une variable est un emplacement en mémoire que vous allez pouvoir lire ou écrire directement. Chaque variable dans un programme est désigné par un nom unique, appelé identifiant.

Créer, modifier et lire une variable

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

main.cpp

```cpp
#include <iostream>

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

Ce code permet de créer une variable appelée `i`, qui peut contenir des nombres entiers (`int` correspond à “integer”, qui signifie “entier”) et qui est initialisée avec la valeur `123`.

Typage fort

Vous avez vu dans les chapitres précédents que le C++ accorde une importance particulière aux types des données. Une même valeur, par exemple 3, pourra signifier différentes choses selon le type : `3` pour un nombre entier, `3.0` pour une nombre réel, `“3”` pour une chaîne de caractères, '3' pour un caractère.

Ce principe de typage fort s'applique également pour les variables : une variable est créée avec un type défini et ne pourra pas changer de type. Si vous créer une variable de type entier par exemple (ie pouvant contenir des nombres entiers), vous ne pourrez pas mettre un autre type dedans.

Si vous essayez, vous pouvez obtenir différents résultats. Par exemple, si vous essayer de mettre dedans une chaîne de caractères, vous aurez une erreur de compilation. Si vous essayez de mettre un nombre réel, celui-ci sera arrondi en valeur entière.

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

La syntaxe générale pour créer une variable peut donc être résumée par le schéma suivant :

```
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).

Vous pouvez créer autant de variables que vous en avez besoin dans votre programme :

```
int x { 123 + 456 };
int y { x * 789 };
cout << x << ' ' << y << endl;
```

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

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'apporte rien par rapport à la syntaxe utilisée dans ce cours (au contraire, dans certains cas, elles peuvent être ambiguës). (4) est appelé inférence de type et sera étudiée dans le prochain chapitre.

Le type

Vous n'avez pas encore vu ce terme, mais vous avez déjà manipulé les types. En effet, “nombre entier”, “nombre réel” ou “chaîne de caractères” sont des types. En C++, les types ne s'écrivent pas de cette manière, il

faut utiliser des mots-clés correspondant à des types. Pour les types que vous connaissez déjà :

- `int` (abréviation de *integer*, “entier” français) correspond à un nombre entier ;
- `double` correspond à un nombre réel (vous verrez par la suite pourquoi on utilise ce terme en C++) ;
- `string` correspond aux chaînes de caractères ; oups, j'ai oublié de parlé de string
- `char` correspond à un caractère ;
- `bool` correspond aux booléens.

Vous avez vu également dans les chapitres précédents comment s'écrivent les littérales correspondantes à chaque type :

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

Il existe bien sûr d'autres types, plus ou moins complexes, et vous pourrez créer vos propres types par la suite.

Le type d'une variable en C++ est définitif, il n'est pas possible de le changer une fois que vous avez défini une variable, vous ne pouvez donc pas écrire :

```
int x { 123 };  // x correspond à une entier
x = "Bonjour";  // erreur : vous ne pouvez pas écrire une
chaîne de caractères
                // dans un entier
```

Lorsque vous ne modifiez pas une variable après l'avoir initialisée, on peut considérer que cette variable est constante. Vous pouvez indiquer cette information au compilateur en ajoutant le mot-clé `const` (“constant”) comme modificateur de type. Ainsi :

- `int` : représente un type entier ;
- `int const` (ou `const int`) : représente un type entier constant.

Indiquer cette information permet au compilateur de faire certaines optimisations (ce que vous ne verrez pas forcément sur un petit programme, mais cela peut avoir un impact sur un programme complexe) et surtout cela permet au compilateur de vérifier que vous ne modifiez pas cette variable par la suite.

```
int x { 123 };
x = 456; // ok, x n'est pas constant

int const y { 123 };
y = 456; // erreur, y est constant
```

L'utilisation de `const` apporte une garantie plus forte sur votre code, vous devez systématiquement réfléchir aux rôles de vos variables et si elles doivent être modifiées durant l'exécution de votre programme ou non. Et donc utiliser le mot-clé `const` aussi souvent que nécessaire.

Notion de contrat
Le typage des variables et l'utilisation de `const` est une forme de contrat que vous passez avec le compilateur. Vous lui dites que vous allez respecter un certain nombre de contraintes (les valeurs seront d'un type défini, les variables ne seront pas modifiées), celui-ci pourra alors vérifier que vous respectez ces contraintes et fera éventuellement des optimisations.

La programmation par contrat est une approche qui permet d'améliorer la qualité de votre code et qui est plus complet que ce qui est présenté ici. Vous verrez dans la suite du cours comment utiliser efficacement la programmation par contrat en C++, en particulier pour créer vos propres types.

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 :

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

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 :

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

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 ;
- `variable1`, `variable2`, etc. : idem ;

- `UnEvArIaBlE` : peu lisible ;
- `une_variable_qui_contient_le_resultat_du_premier_calcul` : trop long.

Bonne pratique de codage : règles que vous vous imposez ainsi qu'aux développeurs qui participent à un projet, pour faciliter la lecture du code par tous. Le but est d'avoir des noms homogènes, simples et informatifs.

Il existe déjà des “règles de codage” toutes faites, vous pouvez utiliser vos propres règles. Les conventions de nommage les plus connues : `une_variable` (STL, Boost), `uneVariable` (Qt)

Pour le moment, 3 sources de règles pour écrire du code :

- le langage C++, imposé par le compilateur ;
- les règles de conception, imposées par la qualité logicielle ;
- les règles de codage, que vous vous imposez.

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 aléatoires. Nous n'allons pas voir toutes les syntaxes possibles pour initialiser une variable, mais uniquement celle qui sont recommandées.

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 };`.

Le signe `=` utilisé pour attribuer une valeur à une variable s'appelle l'opérateur d'affectation.

Plus concrètement, avec du code :

main.cpp

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

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

    // Initialisation avec une valeur
    int const x_value { 123 };
    double const y_value { 123.456 };
    std::string const s_value { "hello, world!" };
    char const c_value { 'a' };
    bool const b_value { true };

    // Initialisation avec une expression
    int const x_expression { 123 + 456 };
    double const y_expression { 12.34 + 56.78 };
    bool const b_expression { 123 > 456 };
}
```

Comme vous avez vu dans les codes précédents, il est possible d'afficher la valeur d'une variable directement avec `cout`. Celui-ci est capable de connaître le type de la variable et d'afficher correctement la valeur, comme si vous aviez écrit une littérale directement avec `cout`. Vous pouvez également utiliser directement une variable dans un calcul.

```
int const x { 123 };
std::cout << "La valeur de x est : " << x << std::endl;
```

```
int const y { x * 45 };
```

Portée et durée de vie

Une variable existe à partir de moment où vous la créez, pas avant. Vous ne pouvez pas utiliser une variable dans une ligne de code et la définir ensuite.

```
cout << i << endl;  // erreur : la variable x est inconnue à cette ligne, elle est
                    // définie uniquement à partir de la ligne suivante

int i { 123 };      // création de i
```

De nouveaux opérateurs arithmétiques

Modifier une variable non const : utilisation de =

Combinaison opérateur et affectation :

```
a = a +b;
a += b;

a = a * b;
a *= b;

etc.
```

Variable temporaire et non temporaire

En pratique, lorsque l'on écrit :

```
int x = y + z;
```

Que se passe-t-il en réalité ?

1. chargement de y et z depuis la mémoire dans le processeur;
2. calcul et résultat dans le processeur;
3. retourne résultat du processeur vers mémoire.

Le résultat du processeur est une variable non nommée (type, valeur) temporaire (= intermédiaire de calcul, pas en mémoire). Cette variable temporaire s'appelle une *rvalue* (le nom vient du fait qu'une rvalue ne peut être qu'a droite du signe d'affectation).

Au contraire, x, y, et z sont des variables nommées non temporaires. Cela s'appelle un lvalue (left value, qui peut être à gauche).

Bien faire attention à la différence : c'est l'expression (y+z) qui est une rvalue, y et z en eux même sont des lvalue.

Note sur bool
Certains types sont convertissable automatiquement en booléen, pour pouvoir tester s'ils sont valide ou non. C'est le cas par exemple des littérales chaînes de caractères. Il est possible d'écrire le code suivant sans que cela ne produise d'erreur :

```
bool b { "hello, world" };
```

Lecture complémentaire

Sur les anciennes syntaxes pour initialiser : A case against direct initialisation

Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++