Pour un grand nombre de problématiques, vous pourrez utiliser des combinaisons d'un ou plusieurs algorithmes de la bibliothèques standard, en combinaison ou pas avec des prédicats spécifiques. Pratiquer la bibliothèque standard est donc indispensable pour être efficace en C++. Et comprendre comment sont implémenter ces fonctions et pourquoi elles sont implémentées comme elles le sont, vous permettra de créer plus facilement vos propres algorithmes.

Mais malgré la diversité et la généricité des algorithmes de la bibliothèque standard, ceux-ci ne peuvent pas non plus répondre à tous les besoins possibles et imaginables. Vous serez forcément amené à ecrire vos propres algorithmes a un moment donné.

Jusqu'à maintenant, le terme “algorithme” a été pour désigner les fonctions fournies dans le fichier d'en-tête `<algorithm>` de la bibliothèque standard. Mais c'est un concept plus générique : l'algorithmique est l'étude des structures de données et des suites d'instructions qui permettent de résoudre un problème. L'algorithmique n'est donc pas spécifique d'un langage de programmation en particulier, et ne se limite pas au traitement des collections de données (comme c'est la cas avec les algorithmes de la bibliothèque standard).

L'algorithmique constitue un champs de connaissance et de recherche très riche, son étude sort du cadre de ce cours. Quelques éléments seront introduits dans ce cours et plusieurs exercices auront pour objectif d'étudier les principaux algorithmes, mais vous devez également étudier un cours d'algorithmique.

Notez que si l'algorithmique vise à trouver une solution à une problématique, la conception vise de son côté à déterminer comment implémenter au mieux cette solution, en tenant compte des contraintes de la qualité logicielle (en premier lieu la fiabilité, la maintenance et l'évolutivité du code).

Les instructions conditionnelles

Les flux non-linéaires d'instructions

Dans les chapitres précédents, les codes que vous avez écrit étaient des suites séquentielles d'instructions, séparées par des points-virgules. Cela signifie que les instructions sont exécutées dans l'ordre où elles apparaissent dans le code et qu'une instruction est exécutée uniquement lorsque l'instruction précédente est terminée.

Le code suivant :

```
instruction1;
instruction2;
instruction3;
```

Doit être lu de la façon suivante : “l'instruction 1 est exécutée. Puis quand elle est finie, l'instruction 2 est exécutée. Puis quand elle est finie, l'instruction 3 est exécutée”.

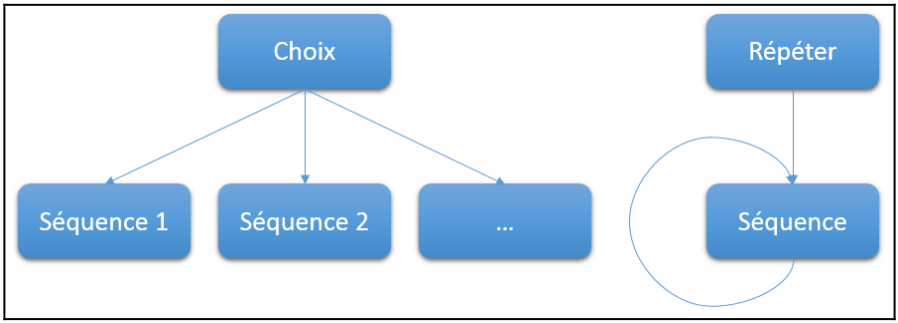
C'est le fonctionnement de la programmation impérative, qui décrit un programme sous forme d'une séquence d'instructions. Pour rappel, voir le chapitre Le programme "hello world".

Mais pour ecrire des algorithmes, cette approche linéaire n'est pas suffisant. Il est nécessaire d'ajouter deux nouvelles façons non séquentielles de suivre le flux d'instructions :

- les conditions (ou embranchements ou tests, *selection statements* en anglais). Elles permettent de choisir entre plusieurs chemins possibles pour suivre le flux d'instruction.
- les boucles (ou iterations, *iteration statements* en anglais). Elles permettent de répéter plusieurs fois une même séquence d'instructions. Elles seront vues dans le chapitre suivant.

Graphiquement, il est classique de représenter les conditions et les boucles de la façon suivante :

mettre a jour l'image



La partie de gauche de la figure représente une condition. Elle doit être lue de la façon suivante : “l'instruction 1 est exécutée, puis l'instruction 2. Si le test est vrai, l'instruction 3 puis l'instruction 4 sont exécutée. Si le test est faux, l'instruction 5 puis l'instruction 6 est exécutée”.

La partie de droite représente une boucle. Elle doit être lu de la façon suivante : ” les instructions 1 puis 2 sont exécutée. Puis les instruction 3 et 4 sont exécutée. Puis encore les instructions 3 et 4. Puis encore les instructions 3 et 4. Puis les instructions 5 et 6 sont exécutées”.

Les conditions permettent de prendre des décisions, selon la valeur prise par une expression. Il existe deux types de conditions, selon le type de la valeur :

- les tests `if-else` prennent une valeur booléenne et permettent de choisir entre deux chemins (“si le test est vrai, faire quelque chose, sinon faire autre chose”) ;
- les tests `switch` prennent une valeur entière et permettent de choisir entre plusieurs valeurs possibles (“selon la valeur donnée, faire quelque chose, ou quelque chose d'autre, ou quelque chose d'autre”).

Les tests if-else

Les instructions if

Une instruction `if` prend une expression retournant un booléen et exécute des instructions si l'expression est vraie. La syntaxe de base est la suivante :

```
if (EXPRESSION) {
    INSTRUCTIONS
}
```

Par exemple :

```
if (i < 5) {
   std::cout << "i est plus petit que 5" << std::endl;
}
```

Ce code peut se lire de la façon suivante : “si la valeur de la variable i est inférieure à 5, alors afficher le texte”.

Les accolades

Les accolades permettent de délimiter un bloc de code constituée de plusieurs instructions. Lors qu'il n'y a qu'une seule ligne d'instruction à exécuter lorsque l'expression booléenne est vraie, il est possible de ne pas mettre les accolades et d'écrire par exemple :

```
if (i < 5)
   std::cout << "i est plus petit que 5" << std::endl;
```

Mais avec cette syntaxe, il est possible d'oublier d'ajouter les accolades si on ajoutes des lignes de code. Beaucoup de guides de codage recommandent de toujours mettre les accolades.

Dans ce cours, les accolades seront toujours utilisées.

L'expression booléen peut être n'importe quoi qui s'évaluent comme étant une valeur booléenne. Cela peut être tout simplement `true` ou `false` (dans ce cas, le compilateur peut optimiser et supprimer complètement l'instruction `if`), une expression logique contenant des

opérateurs de comparaison (`<`, `>`, etc), une fonction qui retourne un booléen. Et vous pouvez combiner plusieurs expressions booléennes en utilisant les opérateurs logiques vus dans le chapitre Logique binaire et calcul booléen.

```
if (true) ...              // une littérale booléenne

if (i < j) ...             // une expression booléenne
utilisant
                           // un opérateur de comparaison

if (v.empty()) ...         // une fonction qui retourne un
                           // booléenne (std::vector::empty)

if (0 < i && i < N) ...    // un expression complexe utilisant
                           // un opérateur logique
```

Faites bien attention en écrivant vos expressions logiques de bien comprendre les opérateurs logiques et de ne pas les inverser. Par exemple, le dernière ligne dans le code précédent permet de tester si un nombre est compris entre 0 et N (“si i est supérieur à 0 ET i est inférieur à N”). Si vous vous trompez d'opérateur logique et utilisez `||` (OU), alors le test devient “si i est superieur a 0 OU i est inférieur a N”, ce qui sera toujours vrai (si N > 0).

Les instructions if-else

L'instruction `else` s'utilise dans un test avec l'instruction `if` et permet d'écrire du code à exécuter lorsque l'expression booléenne dans l'instruction `if` est fausse.

```
if (EXPRESSION) {
    INSTRUCTIONS
} else {
    INSTRUCTIONS
}
```

Par exemple :

```cpp
if (i < 5) {
   std::cout << "i est plus petit que 5" << std::endl;
} else {
   std::cout << "i est plus grand que 5" << std::endl;
}
```

Evaluation paresseuse des opérateurs logiques

Vous avez vu dans le chapitre sur les opérateurs logiques que ceux-ci utilisaient l'évaluation paresseuse. Pour rappel, cela signifie que dans une expression logique contenant `&&` (ET) :

```
A && B
C || D
```

Si `A` est faux, la première expression sera toujours fausse, quelque soit la valeur de `B`. Et si `C` est vrai, la seconde expression sera toujours vraie, quelque soit la valeur de D. Et dans ce cas, l'expression `B` et `D` ne sont pas évaluée.

Il est donc important que les expressions `B` et `D` ne changent pas le comportement du programme. Par exemple, si vous écrivez une fonction qui retourne un booléen et réaliser d'autres opérations :

main.cpp

```cpp
#include <iostream>

bool f() {
    std::cout << "hello" << std:: endl;
    return true;
}

int main() {
    if (true || f()) {
        std::cout << "world" << std:: endl;
    }
}
```

Affiche :

```
world
```

Dans ce code, vous pouvez voir que la fonction `f` n'est pas du tout exécutée et le message qu'elle contient n'est pas affiché.

Ce comportement des opérateurs logiques peut paraître étrange, mais cela permet . Imaginez par exemple que vous souhaitez tester que le résultat d'une division entière soit supérieur a une valeur. Vous pourriez écrire :

```
if ((i / j) > 5) ...
```

Ce code peut sembler correct, mais en fait, il présente une erreur critique : si la variable `j` est nulle, la division est invalide et le programme plante.

Il est donc nécessaire de tester la valeur de `j` avant de faire la division. Une solution peut être d'écrire deux tests `if` de la façon suivante :

```
if (j != 0) {
    if ((i / j) > 5) ...
}
```

Mais il est également possible d'utiliser l'évaluation paresseuse pour simplifier le code :

```
if ((j != 0) && (i / j) > 5) ...
```

Dans ce code, le compilateur commence par évaluer l'expression à gauche de l'opérateur `&&` (`j != 0`). Si cette expression est vraie (donc si `j` n'est pas nul), alors l'expression a droite est évaluée, sans provoquer d crash. Si `j` est nulle, alors la division n'est pas évaluée et cela ne provoque pas non plus de crash.

L'instruction switch

Lorsque vous avez plusieurs valeurs à tester, vous pouvez utiliser des suites de clause `if-else` :

```
if (i == 1) {
    ...
} else if (i ==2) {
    ...
} else if (i ==3) {
    ...
```

Lorsque vous voulez tester de cette façon si une variable entière prend des valeurs en particulier, vous pouvez utiliser une instruction `switch`. La syntaxe est la suivante :

```
switch (EXPRESSION) {
case VALUE1:
    INSTRUCTIONS
    break;
case VALUE2:
    INSTRUCTIONS
    break;
...
default:
    INSTRUCTIONS
}
```

L'expression doit correspondre a un type entier ou équivalent (la norme C++ parle de type “intégrable”) :

- un type entier ;
- une énumération.

Vous ne pouvez donc pas utiliser un chaîne ou un tableau par exemple dans une instruction `switch`, il faut utiliser des instructions `if-else` dans ce cas.

Lors de l'évaluation de la clause `switch`, les différentes clause `case` sont testées une par une et si l'une d'elle correspond à la valeur de la variable, les instructions correspondantes sont exécutées jusqu'à atteindre le mot-clé `break`. Celui-ci interrompre l'exécution de l'instruction `switch` et le programme continue après l'accolade fermante du `switch`.

La clause `default` est exécutée si aucune clause `case` ne correspond à

la valeur de la variable.

Voici un exemple plus concret :

main.cpp

```
#include <iostream>

void print(int i) {
    switch(i) {
        case 1:
            std::cout << "un";
            break;
        case 2:
            std::cout << "deux";
            break;
        case 4:
            std::cout << "quatre";
            break;
        default:
            std::cout << "inconnu";
            break;
    }
    std::cout << "." << std::endl;
}

int main() {
    print(1);
    print(5);
}
```

affiche :

```
un.
inconnu.
```

Dans ce code, la fonction `print` est appelée une première fois avec la valeur 1. Le paramètre de fonction `i` est utilisé directement dans la clause `switch`. Il existe une clause `case` correspondant à cette valeur, les instructions suivantes cette clause sont exécutées :

- l'affichage du texte “un” avec l'instruction `std::cout` ;

- l'exécution du mot-clé `break`, qui termine le `switch`.

Le code continue ensuite avec l'instruction juste après l'accolade fermante du `switch`, c'est a dire l'instruction `std::cout` qui affiche un point et passe a la ligne suivante.

Lors du second appel à la fonction `print`, la valeur passée en argument (5) n'existe pas dans les clauses `case` du `switch`. Dans ce cas, la clause `default` est exécutée et afficher le texte “inconnu”. Puis l'instruction suivant le `switch` est exécutée, ce qui affiche un point et retourne a la ligne.

La clause default

Il n'est pas obligatoire de mettre la clause `default` dans un `switch`. Le mettre systématiquement permet d'éviter qu'une valeur soit oubliée dans un `switch`.

Mais utiliser systématiquement `default` peut avoir une conséquence en termes d'évolutivité du code. Imaginez que vous créez une énumération `Color` qui peut prendre les valeurs `Black` et `White`. Dans une autre partie de votre code, vous avez un `switch` qui définit des clauses `case` pour ces deux valeurs possibles et une clause `default`.

Quelques temps après, vous modifiez votre code pour ajouter la couleur `Red`. Il faut dans ce cas penser à vérifier toutes les instructions `switch` qui utilisent `Color`, pour ajouter une clause `case` pour cette valeur.

Si vous ne mettez pas de clause `default` et que vous “oubliez” de mettre une clause `case` pour une valeur de l'énumération, le compilateur pourra vous signaler ce problème, ce qui facilite l'évolutivité du code.

main.cpp

```
enum class Color { Black, White, Red };

int main() {
    Color c {};
```

```
    switch(c) {
        case Color::Black:
            break;
        case Color::White:
            break;
    }
}
```

affiche l'avertissement suivant, qui indique clairement que la valeur `Red` a été oubliée :

```
main.cpp:5:12: warning: enumeration value 'Red' not
handled in switch [-Wswitch]
    switch(c) {
           ^
```

Si le `switch` contient une clause `default`, ce message d'avertissement n'est pas affiché.

Dans ce cours, la clause `default` sera systématiquement utilisée pour les types entiers, et systématiquement absente pour les énumérations.

Le mot-clé break

Le mot-clé `break` est optionnel dans une clause `case`. Si celui-ci est absent, l'exécution continue sans prendre en compte les clauses `case` suivantes, jusqu'à rencontrer un `break` ou d'arriver à la fin du `switch`.

Voyez par exemple le code suivant :

main.cpp

```
#include <iostream>

void print(int i) {
    switch(i) {
        case 1:
            std::cout << "un"; // pas de break
        case 2:
            std::cout << "deux";
            break;
```

```
        case 4:
            std::cout << "quatre";
            break;
        default:
            std::cout << "inconnu";
            break;
    }
    std::cout << "." << std::endl;
}

int main() {
    print(1);
}
```

affiche :

```
undeux.
```

En effet, la clause `case` correspondant a la valeur 1 est exécutée (ce qui affiche le texte “un”), mais il n'y a pas de mot-clé `break` entre cette clause et la clause suivante. La clause suivante (correspondant à la valeur 2) est alors exécutée et affiche le texte “deux”. Cette clause contient un `break`, ce qui termine l'exécution du `switch` et affiche le point et le retour à la ligne.

Il faut faire particulièrement attention de ne pas oublier de `break`, sous peine d'avoir un comportement étrange lors de l'exécution. Et si l'absence du `break` est volontaire, il est préférable de mettre un commentaire, pour bien indiquer qu'il ne s'agit pas d'un oubli.

fallthrough [C++17]

Dans la future norme du C++, un attribut a été ajoutée pour indiquer clairement dans une clause `case` que le `break` est volontairement absent. Cet attribut `fallthrough` s'écrit entre double crochets droits et se place à la place du `break` qu'il remplace.

```
case 1:
    std::cout << "un"; // pas de break
    [[fallthrough]];
```

```
case 2:
    std::cout << "deux";
    break;
```

Cette fonctionnalité est déjà implémentée dans certains compilateurs, qui indique un avertissement lorsqu'il n'y a ni `break`, ni attribue `fallthrough`. Cette fonctionnalité est activée en utilisant la directive de compilation `-Wimplicit-fallthrough` et l'attribue se nomme `fallthrough`.

Le code d'exemple sans l'attribut `fallthrough` affiche alors le message suivant :

```
main.cpp:8:1: warning: unannotated fall-through between
switch labels [-Wimplicit-fallthrough]
case 2:
^
main.cpp:8:1: note: insert '[[clang::fallthrough]];' to
silence this warning
case 2:
^
[[clang::fallthrough]];
main.cpp:8:1: note: insert 'break;' to avoid fall-through
case 2:
^
break;
1 warning generated.
```

Ce message indique qu'il faut ajouter un `break` ou `[[clang::fallthrough]]`.

Portée dans un switch

L'instruction `switch` correspond a une seule portée. Cela implique que vous ne pouvez pas déclarer une même variable dans deux clauses `case` différentes, puisque cela provoquera un conflit de noms.

main.cpp

```
int main() {
```

```
    switch(1) {
        case 1:
            int i { 123 };
            break;
        case 2:
            int i { 123 };
            break;
        default:
            break;
    }
}
```

affiche les messages suivants :

```
main.cpp:7:17: warning: declaration shadows a local variable
[-Wshadow]
            int i { 123 };
                ^
main.cpp:4:17: note: previous declaration is here
            int i { 123 };
                ^
main.cpp:7:17: error: redefinition of 'i'
            int i { 123 };
                ^
... // d'autres message ensuite
```

Pour éviter cela, il faut ajouter des accolades, pour créer un bloc d'instruction dans les clauses `case`, de façon à réduire la portée des variables et éviter les conflits.

main.cpp

```
int main() {
    switch(1) {
        case 1: {
            int i { 123 };
            break;
        }
        case 2: {
            int i { 123 };
            break;
        }
```

```
        default:
            break;
    }
}
```

L'opérateur conditionnel ternaire

L'opérateur conditionnel ternaire est similaire à une instruction `if-else` et permet de choisir entre deux expression, selon le résultat d'une condition booléenne.

La syntaxe est la suivante :

```
EXPRESSION_BOOLEENNE ? EXPRESSION_SI_VRAI :
EXPRESSION_SI_FAUX
```

Opérateurs unaire, binaire et ternaire
Un opérateur unaire est un opérateur qui prend une seule valeur. Par exemple les opérateurs `++i` (opérateur d'incrémentation) et `--i` (opérateur de décrémentation).

Un opérateur binaire prend deux valeurs. Par exemple `x + y` (opérateur arithmétique), `A && B` (opérateur logique) ou `i < j` (operateur de comparaison).

Un opérateur ternaire prend trois valeurs. En C++, il n'existe qu'un seul opérateur ternaire, c'est l'opérateur conditionnel qui est présenté dans ce chapitre. Par abus de langage et comme il n'y a pas d'ambiguïté possible en C++, l'expression “opérateur ternaire” est souvent utilisée pour parler de l'opérateur conditionnel ternaire.

La particularité de cette syntaxe est que cela forme une seule expression, qui peut donc être utilisée n'importe où une expression est acceptée. Pour bien comprendre, voici un exemple simple : pour initialiser une variable, vous avez vu qu'il existait plusieurs syntaxes possibles. En particulier, il est possible d'initialiser en utilisant une expression :

```
const int i { EXPRESSION };
```

Cette expression peut être vide, être une littérale, un appel de fonction, un calcul arithmétique, etc.

```
const int i {};
const int i { 123 };
const int i { foo() };
const int i { 1 + 2 };
```

Si vous souhaitez initialiser une variable en fonction d'une condition booléenne, vous ne pouvez pas utiliser directement une structure de contrôle `if-else` a la place de l'expression :

```
const int i { if (CONDITION) ... }; // erreur de syntaxe
```

Il est possible d'utiliser `if-else`, mais cela implique de “sortir” la déclaration de la variable en dehors du `if-else`.

```
int i {};
if (CONDITION) {
    i = 123;
} else {
    i = 456;
}
```

Mais cela implique que la déclaration et l'initialisation de la variable sont séparés et qu'il n'est pas possible d'utiliser `const`.

L'opérateur conditionnel ternaire est une expression et peut donc être utilisée pour initialiser directement une variable lors de la déclaration.

```
const int i { CONDITION ? 123 : 456 };  // ok
```

Pour lire cette ligne de code, vous devez procéder en deux temps :

- évaluer l'expression, pour déterminer sa valeur (123 ou 456) ;
- initialiser la variable en utilisant cette valeur.

Exemples d'utilisation

Un opérateur conditionnelle ternaire peut être utilisée aussi avec une instruction `return` pour un retour de fonction ou comme argument dans un appel de fonction.

```
// Retourne "min" si "i" est inférieur a "min", retourne
"max"
// si "i" est supérieur à "max", ou sinon retourne "i".

int clamp(int i, int min, int max) {
    return (i < min ? min : (i > max ? max : i));
}
```

Note : cette fonction utilise deux opérateurs ternaires imbriquées, prenez le temps de la comprendre.

Le premier opérateur ternaire compare `i` et `min` : si `i` est inférieur à `min`, l'opérateur ternaire retourne `min` ; si `i` est supérieur à `min`, le second opérateur ternaire est évaluée.

Le second opérateur ternaire compare `i` et `max` : si `i` est supérieur à `max`, l'opérateur ternaire retourne `max` ; si `i` est inférieur à `max`, l'opérateur ternaire retourne `i`.

La fonction `clamp` permet donc de retourner une valeur toujours comprise entre `min` et `max`. (Note : cette fonction a été ajoutée dans la prochaine norme du C++17).

Contrainte et inconvénient

L'opérateur ternaire présente une contrainte syntaxique : les sous-expressions utilisées doivent retourner le même type. Il n'est pas possible par exemple d'écrire :

```
condition ? 123 : "hello";
```

Puisque la sous-expression a gauche est de type entière (`int`) alors que

celle de droite est une chaîne de caractères (`const char[]`).

L'inconvénient majeur de l'opérateur conditionnel est son manque de lisibilité, en particulier lorsque plusieurs opérateurs ternaires sont imbriqués (comme c'est le cas avec l'exemple précédent de la fonction `clamp`) ou lorsque les sous-expressions sont complexes.

Pour éviter cela, il n'existe pas beaucoup de méthodes : faites un effort de présentation de votre code, et n'utilisez pas l'opérateur ternaire lorsque cela nuit à la lisibilité. Et n'hésitez pas a utiliser les parenthèses pour bien encadrer l'opérateur ternaire.

Chapitre précédent	Sommaire principal	Chapitre suivant