

Programmation C++

Une version à jour et éditable de ce livre est disponible sur Wikilivres,
une bibliothèque de livres pédagogiques, à l'URL :
https://fr.wikibooks.org/wiki/Programmation_C%2B%2B

Vous avez la permission de copier, distribuer et/ou modifier ce document selon les termes de la Licence de documentation libre GNU, version 1.2 ou plus récente publiée par la Free Software Foundation ; sans sections inaltérables, sans texte de première page de couverture et sans Texte de dernière page de couverture. Une copie de cette licence est incluse dans l'annexe nommée « Licence de documentation libre GNU ».

Sections

- [1 Avant propos](#)
- [2 Introduction](#)
 - [2.1 Quelques repères historiques](#)
 - [2.2 Présentation du C++](#)
 - [2.3 Les apports du C++ par rapport au C](#)
- [3 Compilation](#)
 - [3.1 Définition de la compilation](#)
 - [3.1.1 Phases de la compilation](#)
 - [3.2 Les fichiers sources](#)
 - [3.2.1 Les fichiers d'en-tête](#)
 - [3.2.2 Les fichiers .cpp](#)
 - [3.3 L'édition de liens \(linking\)](#)
 - [3.4 Les projets](#)
 - [3.4.1 Les makefiles](#)
 - [3.4.2 Les environnements de développement intégrés](#)
 - [3.5 Quelques exemples](#)
 - [3.5.1 Multi-plateformes](#)
 - [3.5.2 Sous windows](#)
 - [3.5.3 Sous Linux](#)
 - [3.5.4 Sous MacOS X](#)
- [4 Un premier programme](#)
 - [4.1 Le fichier source](#)
 - [4.2 Explications](#)
 - [4.2.1 La directive #include](#)
 - [4.2.2 L'espace de nommage standard](#)
 - [4.2.3 La fonction main](#)
 - [4.2.4 L'objet cout](#)
 - [4.2.5 return 0](#)
 - [4.3 Exécution](#)
- [5 Les commentaires](#)
 - [5.1 Bloc de commentaire](#)
 - [5.2 Commentaire de fin de ligne](#)
- [6 Le préprocesseur](#)
 - [6.1 Inclusion de fichiers](#)
 - [6.2 #define, #undef](#)
 - [6.3 #ifdef, #ifndef, #if, #endif et #else](#)
 - [6.3.1 Présentation](#)
 - [6.3.2 Exemples](#)
 - [6.3.2.1 Exemple 1](#)
 - [6.3.2.2 Exemple 2](#)
 - [6.3.2.3 Exemple 3](#)
 - [6.3.2.4 Autre solution](#)
 - [6.4 Les macros](#)
 - [6.4.1 Présentation](#)
 - [6.4.2 Exemple](#)
 - [6.4.3 Bonnes pratiques](#)
- [7 Les types de base et les déclarations](#)

- 7.1 Déclarations, types et identificateurs
 - 7.1.1 Les variables
 - 7.1.2 Les types en C++ et les systèmes de représentation
 - 7.1.3 Les déclarations
 - 7.1.4 Identificateurs valides
 - 7.1.5 Les types entiers
 - 7.1.6 Les types réels
 - 7.1.7 Les caractères
 - 7.1.7.1 Le type char
 - 7.1.7.1.1 Transformation de majuscule en minuscule
 - 7.1.7.2 Les types signed char et unsigned char
 - 7.1.7.3 Les types char16_t and char32_t
 - 7.1.7.4 Le type wchar_t
 - 7.1.7.5 API exceptionnellement Unicode
 - 7.1.8 Les booléens
 - 7.1.9 L'opérateur const
 - 7.1.10 L'opérateur sizeof
 - 7.1.11 Définir un alias de type
- 7.2 Voir aussi
- 8 Les opérations de base
 - 8.1 L'affectation
 - 8.1.1 Syntaxe
 - 8.1.2 Sémantique
 - 8.1.3 Exemple
 - 8.1.3.1 Exécution
 - 8.1.3.2 Explications
 - 8.1.4 Affectation à la déclaration
 - 8.1.5 Affectation en série
 - 8.1.6 Remarque importante
 - 8.2 Opérations arithmétiques
 - 8.2.1 sur les entiers
 - 8.2.2 sur les réels
 - 8.2.3 Exemple
 - 8.2.3.1 Exécution :
 - 8.3 Opérations binaires
 - 8.3.1 Les décalages
 - 8.3.1.1 Exemple
 - 8.3.1.2 Exécution
 - 8.3.2 Le ET binaire
 - 8.3.2.1 Syntaxe
 - 8.3.2.2 Sémantique
 - 8.3.2.3 Exemple
 - 8.3.2.4 Exécution
 - 8.3.2.5 Explications
 - 8.3.3 Le OU binaire
 - 8.3.3.1 Syntaxe
 - 8.3.3.2 Sémantique
 - 8.3.3.3 Exemple
 - 8.3.3.4 Exécution
 - 8.3.3.5 Explications

- 8.3.4 Le OU exclusif binaire
 - 8.3.4.1 Syntaxe
 - 8.3.4.2 Sémantique
 - 8.3.4.3 Exemple
 - 8.3.4.4 Exécution
 - 8.3.4.5 Explications
- 8.3.5 Le NON binaire
 - 8.3.5.1 Syntaxe
 - 8.3.5.2 Sémantique
 - 8.3.5.3 Exemple
 - 8.3.5.4 Exécution
 - 8.3.5.5 Explications
- 8.4 Opérations booléennes
 - 8.4.1 Comparaisons usuelles
 - 8.4.2 Le ET logique
 - 8.4.2.1 Syntaxe
 - 8.4.2.2 Sémantique
 - 8.4.3 Le OU logique
 - 8.4.3.1 Syntaxe
 - 8.4.3.2 Sémantique
 - 8.4.4 Le NON logique
 - 8.4.4.1 Syntaxe
 - 8.4.4.2 Sémantique
 - 8.4.5 Exemples d'utilisation des opérateurs booléens
- 8.5 Affectation avec opérateur
 - 8.5.1 L'opérateur +=
 - 8.5.1.1 Syntaxe
 - 8.5.1.2 Sémantique
 - 8.5.1.3 Exemple
 - 8.5.1.4 Exécution
 - 8.5.2 Autres opérateurs
- 8.6 Priorité des opérateurs
- 8.7 La liste complète des opérateurs du C++
- 9 Les entrées-sorties
 - 9.1 Classes de gestion des flux
 - 9.2 Flux standards
 - 9.2.1 Exemple
 - 9.3 Autres types de flux
 - 9.3.1 Flux de fichier
 - 9.3.2 Flux de chaîne de caractères
 - 9.4 Manipulateurs
 - 9.4.1 Manipulateur endl
 - 9.4.2 Manipulateur hex
 - 9.4.3 Manipulateur dec
 - 9.4.4 Manipulateur setbase(*base*)
 - 9.4.5 Manipulateur setw(*width*)
 - 9.4.6 Manipulateur setfill(*char*)
 - 9.4.7 Manipulateur setprecision(*digits*)
 - 9.4.8 Manipulateurs setiosflags et resetiosflags
- 10 Les pointeurs

- [10.1 Introduction](#)
- [10.2 Déclaration](#)
- [10.3 L'opérateur &](#)
- [10.4 L'opérateur *](#)
- [10.5 Comparaison avec une variable classique](#)
- [10.6 Exemple de programme](#)
 - [10.6.1 Exécution](#)
 - [10.6.2 Explications](#)
- [10.7 Opérations arithmétiques sur les pointeurs](#)
 - [10.7.1 Exemple 1](#)
 - [10.7.2 Exemple 2](#)
- [10.8 Opération utilisant deux pointeurs](#)
- [10.9 Pointeur constant et pointeur vers valeur constante](#)
 - [10.9.1 Pointeur vers une valeur constante](#)
 - [10.9.2 Pointeur constant](#)
 - [10.9.3 Pointeur constant vers une valeur constante](#)
 - [10.9.4 Position du mot clé const](#)
- [10.10 Voir aussi](#)
- [11 Les références](#)
 - [11.1 Présentation des références](#)
 - [11.1.1 Déclaration](#)
 - [11.1.2 Sémantique](#)
 - [11.1.3 Exemple de programme](#)
 - [11.1.3.1 Exécution](#)
 - [11.1.3.2 Explications](#)
 - [11.1.4 Pourquoi utiliser une référence ?](#)
 - [11.2 Les références et leur lien avec les pointeurs](#)
 - [11.3 Exercices](#)
 - [11.3.1 Exercice 1](#)
 - [11.3.2 Exercice 2](#)
 - [11.4 Tests](#)
 - [11.4.1 Test 1](#)
 - [11.4.1.1 Cas 1](#)
 - [11.4.1.2 Cas 2](#)
 - [11.4.1.3 Cas 3](#)
 - [11.4.1.4 Cas 4](#)
 - [11.4.1.5 Cas 5](#)
 - [11.4.1.6 Solution](#)
 - [11.4.2 Test 2](#)
 - [11.4.2.1 Cas 1](#)
- [12 Les tableaux](#)
 - [12.1 Les tableaux à une dimension](#)
 - [12.1.1 Les tableaux statiques](#)
 - [12.1.1.1 Syntaxe](#)
 - [12.1.1.2 Sémantique](#)
 - [12.1.1.3 Pointeurs et tableaux](#)
 - [12.1.1.4 Exemple](#)
 - [12.1.1.4.1 Exécution](#)
 - [12.1.2 Les tableaux dynamiques](#)

- [12.1.2.1 L'opérateur new](#)
 - [12.1.2.2 L'opérateur delete\[\]](#)
 - [12.1.2.3 Exemple](#)
 - [12.1.2.3.1 Exécution 1](#)
 - [12.1.2.3.2 Exécution 2](#)
 - [12.1.2.4 Tableaux multidimensionnels](#)
 - [12.1.3 Les vectors](#)
 - [12.1.4 Parcours d'un tableau](#)
 - [12.2 Les tableaux à deux dimensions](#)
 - [12.2.1 Statiques](#)
 - [12.2.1.1 Exemple](#)
 - [12.2.1.2 Exécution](#)
 - [12.2.2 Dynamiques](#)
 - [12.2.2.1 Exemple](#)
 - [12.2.2.2 Exécution 1](#)
 - [12.2.2.3 Exécution 2](#)
 - [12.3 Les tableaux de caractères](#)
 - [12.3.1 Les tableaux de char](#)
 - [12.3.2 Les tableaux de char16_t and char32_t](#)
 - [12.3.3 Les tableaux de wchar_t](#)
 - [12.3.4 Portabilité](#)
 - [12.4 Voir aussi](#)
- [13 Les structures de contrôles](#)
 - [13.1 Le if](#)
 - [13.1.1 Syntaxe :](#)
 - [13.1.2 Sémantique :](#)
 - [13.1.3 Exemple :](#)
 - [13.1.3.1 Exécution 1](#)
 - [13.1.3.2 Exécution 2](#)
 - [13.1.4 Condition multiple contenant des opérateurs logiques](#)
 - [13.2 Le if...else](#)
 - [13.2.1 Syntaxe](#)
 - [13.2.2 Sémantique](#)
 - [13.2.3 Exemple :](#)
 - [13.2.3.1 Exécution 1](#)
 - [13.2.3.2 Exécution 2](#)
 - [13.2.4 Plusieurs instructions par condition](#)
 - [13.3 Le switch](#)
 - [13.3.1 Syntaxe](#)
 - [13.3.2 Sémantique](#)
 - [13.3.3 Exemple 1](#)
 - [13.3.4 Exemple 2](#)
 - [13.4 Le for "classique"](#)
 - [13.4.1 Syntaxe](#)
 - [13.4.2 Sémantique](#)
 - [13.4.3 Exemples](#)
 - [13.4.4 Instructions multiples](#)
 - [13.4.5 Boucle infinie](#)
 - [13.5 le for "moderne"](#)

- [13.5.1 Syntaxe](#)
 - [13.5.2 Sémantique](#)
 - [13.5.3 Exemple](#)
- [13.6 Le while](#)
 - [13.6.1 Syntaxe](#)
 - [13.6.2 Sémantique](#)
 - [13.6.3 Exemple de programme](#)
 - [13.6.3.1 Exécution](#)
 - [13.6.3.2 Explications](#)
- [13.7 Le do ... while](#)
 - [13.7.1 Syntaxe](#)
 - [13.7.2 Sémantique](#)
 - [13.7.3 Exemple](#)
- [13.8 Le goto](#)
- [13.9 Le break](#)
 - [13.9.1 Syntaxe](#)
 - [13.9.2 Sémantique](#)
 - [13.9.3 Exemple](#)
 - [13.9.3.1 Exécution](#)
 - [13.9.4 Note importante](#)
- [13.10 Le continue](#)
 - [13.10.1 Syntaxe](#)
 - [13.10.2 Sémantique](#)
 - [13.10.3 Exemple](#)
 - [13.10.3.1 Exécution](#)
- [13.11 Voir aussi](#)
- [14 Les fonctions](#)
 - [14.1 Utilisation des fonctions](#)
 - [14.1.1 Compléments indispensables pour les débutants](#)
 - [14.1.2 Prototype d'une fonction](#)
 - [14.1.2.1 syntaxe](#)
 - [14.1.2.2 Exemple](#)
 - [14.1.2.3 Rôle](#)
 - [14.1.3 Définition d'une fonction](#)
 - [14.1.3.1 Syntaxe](#)
 - [14.1.3.2 Exemple](#)
 - [14.1.3.3 Exemple avec prototype](#)
 - [14.1.4 Portée des variables](#)
 - [14.1.4.1 Présentation](#)
 - [14.1.4.2 Exemple](#)
 - [14.1.5 Passage de paramètres par pointeur](#)
 - [14.1.6 Passage de paramètres par référence](#)
 - [14.1.7 Pointeur de fonction](#)
 - [14.1.8 Passage de fonctions en paramètre](#)
 - [14.2 Exercices](#)
 - [14.2.1 Exercice 1](#)
 - [14.2.2 Exercice 2](#)
 - [14.2.3 Exercice 3](#)
 - [14.2.4 Exercice 4](#)

- [14.3 Voir aussi](#)
- [15 Les structures](#)
 - [15.1 Présentation](#)
 - [15.2 Syntaxe](#)
 - [15.3 Exemple](#)
 - [15.4 Pointeur vers une structure](#)
 - [15.5 Copier une structure](#)
 - [15.6 Voir aussi](#)
- [16 Les classes](#)
 - [16.1 La notion de classe en programmation orientée objet](#)
 - [16.2 L'encapsulation en C++](#)
 - [16.3 Les fonctions membres](#)
 - [16.3.1 Déclaration](#)
 - [16.3.2 Définition](#)
 - [16.4 Constructeurs et destructeurs](#)
 - [16.4.1 Constructeurs](#)
 - [16.4.2 Destructeurs](#)
 - [16.5 Exemples de classes](#)
 - [16.6 Les opérateurs new et delete](#)
 - [16.7 Surcharge d'opérateurs](#)
 - [16.7.1 Présentation](#)
 - [16.7.2 Opérateurs surchargeables](#)
 - [16.7.3 Syntaxe](#)
 - [16.7.4 Exemple](#)
 - [16.8 Héritage](#)
 - [16.8.1 Présentation](#)
 - [16.8.2 Syntaxe](#)
 - [16.8.3 Exemple](#)
 - [16.8.4 Méthodes virtuelles](#)
 - [16.8.5 Destructeur virtuel](#)
 - [16.8.6 Héritage multiple](#)
 - [16.8.7 Voir aussi](#)
 - [16.9 Classes abstraites](#)
 - [16.9.1 Présentation](#)
 - [16.9.2 Syntaxe](#)
 - [16.9.3 Exemple](#)
 - [16.10 Pointeur de membre](#)
 - [16.10.1 Exemple](#)
 - [16.10.2 Exemple](#)
 - [16.11 Voir aussi](#)
- [17 Les espaces de noms](#)
 - [17.1 Introduction](#)
 - [17.2 Créer un espace de noms](#)
 - [17.3 Utiliser un espace de noms](#)
 - [17.4 Alias d'espace de noms](#)
 - [17.5 En plusieurs fois](#)
- [18 Les exceptions](#)
 - [18.1 Exceptions en C++](#)
 - [18.2 Lancer une exception](#)
 - [18.3 Attraper une exception](#)
 - [18.4 Attraper toutes les exceptions](#)

- [18.5 Déclaration des exceptions lancées](#)
 - [19 Les templates](#)
 - [19.1 Définitions des templates](#)
 - [19.2 Patron de fonction](#)
 - [19.3 Patron de classe](#)
 - [19.4 Voir aussi](#)
 - [20 Conventions d'appel](#)
 - [20.1 Convention d'appel](#)
 - [20.2 Conventions d'appels sous Windows](#)
 - [20.2.1 Spécifier la convention d'appel](#)
 - [20.2.2 Différences entre les conventions d'appel](#)
 - [20.3 Nommage des fonctions](#)
 - [20.4 Liens](#)
 - [21 Les fichiers](#)
 - [21.1 Les fichiers](#)
 - [21.2 Généralité sur les fichiers](#)
 - [21.3 Fichiers textes ou binaires](#)
 - [21.4 cstdio ou fstream](#)
 - [21.5 Utilisation de cstdio](#)
 - [21.5.1 La fonction FILE * fopen\(const char * filepath, char * mode\)](#)
 - [21.5.2 La fonction fclose\(FILE *\)](#)
 - [21.5.3 Les fichiers binaires](#)
 - [21.5.4 Les fichiers textes](#)
 - [21.5.5 Encodage](#)
 - [21.6 Utilisation de fstream](#)
 - [21.6.1 Les fichiers textes](#)
 - [21.6.2 Les fichiers binaires](#)
 - [22 La librairie standard](#)
 - [22.1 La STL](#)
 - [22.1.1 Les conteneurs](#)
 - [22.1.1.1 Les conteneurs séquentiels](#)
 - [22.1.1.2 Les conteneurs associatifs](#)
 - [22.1.1.3 Les adaptateurs de conteneurs](#)
 - [22.1.2 Les algorithmes](#)
 - [22.1.2.1 Les algorithmes de séquences non modifiants](#)
 - [22.1.2.2 Les algorithmes de séquences modifiants](#)
 - [22.1.2.2.1 copies](#)
 - [22.1.2.2.2 échanges](#)
 - [22.1.2.2.3 transformations](#)
 - [22.1.2.2.4 remplacements](#)
 - [22.1.2.2.5 remplissages](#)
 - [22.1.2.2.6 générations](#)
 - [22.1.2.2.7 suppressions](#)
 - [22.1.2.2.8 éléments uniques](#)
 - [22.1.2.2.9 ordre inverse](#)
 - [22.1.2.2.10 rotations](#)
 - [22.1.2.2.11 permutations aléatoires](#)
 - [22.1.2.2.12 répartitions](#)
 - [22.1.2.3 Les algorithmes de tri et les opérations apparentées](#)
 - [22.1.2.3.1 tris](#)

- [22.1.2.3.2 recherches dichotomiques](#)
- [22.1.2.3.3 fusions](#)
- [22.1.2.3.4 opérations d'ensemble](#)
- [22.1.2.3.5 opérations de tas](#)
- [22.1.2.3.6 minimum et maximum](#)
- [22.1.2.3.7 permutations](#)
- [22.2 Les chaînes de caractères](#)
 - [22.2.1 La classe string](#)
 - [22.2.1.1 Différentes opérations sur la classe string](#)
 - [22.2.1.2 Exemple 1 : la classe string](#)
 - [22.2.1.2.1 Explications](#)
 - [22.2.1.2.2 Exécution](#)
 - [22.2.1.3 Séparateurs](#)
 - [22.2.1.4 Exemple 2 : string avec des espaces](#)
 - [22.2.1.4.1 Explications](#)
 - [22.2.1.4.2 Exécution](#)
 - [22.2.1.5 Analyse de chaînes](#)
 - [22.2.1.6 Exemple 3 : analyse de chaînes](#)
 - [22.2.1.6.1 Explications](#)
 - [22.2.1.6.2 Exécution](#)
 - [22.2.1.7 Compatibilité avec les char * et les tableaux de char](#)
 - [22.2.2 Exemple 4 : compatibilité avec les tableaux de char et les char *](#)
 - [22.2.2.1 Explications](#)
 - [22.2.2.2 Exécution](#)
 - [22.2.2.3 Transformation d'une chaîne en int ou double](#)
 - [22.2.2.4 Exemple 5 : transformation de string en int](#)
 - [22.2.2.4.1 Explications](#)
 - [22.2.2.4.2 Exécution 1](#)
 - [22.2.2.4.3 Exécution 2](#)
 - [22.2.3 Les chaînes de caractères de type C](#)
 - [22.2.3.1 memcpy](#)
 - [22.2.3.2 memmove](#)
 - [22.2.3.3 strcpy](#)
 - [22.2.3.4 strncpy](#)
 - [22.2.3.5 strcat](#)
 - [22.2.3.6 strncat](#)
 - [22.2.3.7 memcmp](#)
 - [22.2.3.8 strcmp](#)
 - [22.2.3.9 strcoll](#)
 - [22.2.3.10 strncmp](#)
 - [22.2.3.11 strxfrm](#)
 - [22.2.3.12 strcspn](#)
 - [22.2.3.13 strspn](#)
 - [22.2.3.14 strtok](#)
 - [22.2.3.15 memset](#)
 - [22.2.3.16 strerror](#)
 - [22.2.3.17 strlen](#)
 - [22.2.3.18 memchr](#)
 - [22.2.3.19 memchr\(void* p, int c, size_t n\)](#)
 - [22.2.3.20 inline char* strchr\(char* s1, int __n\)](#)
 - [22.2.3.21 inline char* strpbrk\(char* s1, const char* s2\)](#)

- [22.2.3.22 inline char* strrchr\(char* s1, int n\)](#)
- [22.2.3.23 inline char* strstr\(char* s1, const char* s2\)](#)
- [22.3 Voir aussi](#)
- [23 Expressions rationnelles](#)
 - [23.1 Références](#)
- [24 Interfaces graphiques](#)
- [25 Bibliographie et liens](#)
 - [25.1 Bibliographie](#)
 - [25.2 Liens sur les autres projets](#)
 - [25.3 Liens Externes](#)
- [26 Dessiner des formes : les bibliothèques graphiques](#)
 - [26.1 Téléchargement de la SFML](#)
 - [26.2 Mise en place de la SFML](#)
 - [26.3 Utilisation de la SFML](#)
- [27 Opérateurs](#)
- [28 Héritage](#)
- [29 Polymorphisme](#)
 - [29.1 Polymorphisme](#)
 - [29.2 Exemple](#)
 - [29.3 Problème](#)
 - [29.4 Solution](#)

Avant propos

Ce livre est destiné à présenter les différents aspects du langage de programmation [C++](#). L'objectif est d'en faire un livre référence sur les différents aspects de ce langage, en étant le plus exhaustif possible. Nous essayerons de proposer pour chaque concept un voire plusieurs exemples illustratifs complets.

```
/* C */
#include <stdio>
Main()
{
    printf("Hallo Welt.");
}

// C++
#include <iostream>
using namespace std;

int main(void)
{
    cout << "Hallo Welt." << endl;
    return 0;
}
```

Exemple de codes C et C++.

Introduction

Quelques repères historiques

À l'origine, un ordinateur ne comprenait que le langage binaire : un programme était constitué d'une suite de 0 et de 1. Le programmeur devait lui-même traduire son programme sous cette forme. En 1950, Alan Turing et Maurice V. Wilkes de l'université de Cambridge branchent un clavier à un ordinateur, il est dorénavant possible de rentrer des mots associés au langage machine. Par exemple : mov, load, sub... Ces mots ne sont pas compris par l'ordinateur qui ne comprend que le langage binaire. Il faut alors un « assembleur » qui transforme le code en langage binaire.

La nécessité de créer un langage de haut niveau se fait sentir, le FORTRAN (FORMula TRANslator, traducteur de formules) est créé en 1957. Il est utilisé pour des programmes mathématiques et scientifiques.

En 1970, Ken Thompson, créa un nouveau langage : Le B, descendant du BCPL (Basic Combined Programming Language, créé en 1967 par Martin Richards). Son but était de créer un langage simple, malheureusement, son langage fut trop simple et trop dépendant de l'architecture utilisée...

En 1971 Dennis Ritchie commence à mettre au point le successeur du B, le C. Le résultat est convaincant : Le C est totalement portable (il peut fonctionner sur tous les types de machines et de systèmes), il est de bas niveau (il peut créer du code aussi rapide que de l'assembleur) et il permet de traiter des problèmes de haut niveau. Le C permet de quasiment tout faire, de driver au jeu.

Le C devient très vite populaire, tout le monde veut créer sa version. Beaucoup de compilateurs qui voient le jour sont incompatibles entre eux et la portabilité est perdue. Il est décidé qu'il faut un standard pour le C. L'ANSI (American National Standards Institute) s'en charge en 1983. La plupart des compilateurs se conforment à ce standard. Un programme écrit en C ANSI est compatible avec tous les compilateurs.

Il manque la programmation orientée objet au C. C'est pourquoi Bjarne Stroustrup, des laboratoires Bell, crée le C++, dont le développement a commencé au début de années 1980. Il construit donc le C++ sur la base du C. C++ est capable de compiler un programme C, et garde donc une forte compatibilité avec le C.

Les langages C et C++ sont les langages les plus utilisés dans le monde de la programmation.

Présentation du C++

Le C++ est un langage multiparadigme. Il supporte essentiellement les paradigmes suivants :

- programmation procédurale : il reprend essentiellement les concepts du langage C, notamment la notion de fonction (une procédure étant une fonction avec un retour de type 'void') ;
- programmation structurée : il reprend la notion struct du langage C. Cette notion est considérée en C++ aussi comme des classes dont l'accès par défaut est public ;
- programmation orientée-objet : il implémente la notion de classe (dont l'accès par défaut est privé), d'encapsulation grâce aux restrictions d'accès (publique, protégé, privé), d'héritage (simple ou multiple) à l'aide du mécanisme de dérivation, d'abstraction grâce aux classes de base abstraites pures (on peut parler d'interface bien que les héritages multiples sur une interface C++ lèvent des difficultés montrant que leur implémentation est plus proche d'une classe abstraite que d'une vraie interface comme on peut en trouver en Java) ou non, de polymorphisme dynamique (ou au runtime) grâce aux fonctions membres virtuelles ;
- programmation générique ou méta-programmation : il introduit les templates ou modèles générique de code qui permettent de créer automatiquement des fonctions ou des classes à partir d'un ou plusieurs paramètres.
- programmation 'lambda-closure' : le C++ 11 introduit la notion de fermeture.

Outre ces grands paradigmes, C++ implémente la notion de typage strict, de constance, de polymorphisme statique (ou à la compilation) grâce à la surcharge et aux fonctions génériques, de références (une alternative aux pointeurs bien plus robuste et bien moins dangereuse) et permet également la surcharge d'opérateurs et de simuler les mixins grâce au mécanisme de dérivation multiple. Il est largement compatible avec le langage C, ce qui est à la fois une richesse et un problème. En effet, le langage C est actuellement (en 2006) un langage largement utilisé pour écrire des systèmes d'exploitation (Windows, Linux, Mac OS). Le C++ peut donc très naturellement avoir un accès direct au système d'exploitation dans le cadre d'une programmation de bas niveau, tout en permettant une programmation de haut niveau en exploitant toute la richesse des concepts orientés objet. Le problème levé par ce lien filial étant que les développeurs ayant appris le C avant le C++ utilisent les techniques C là où le C++ possède des ajouts moins dangereux (par exemple, les références). Ce type de code est surnommé « *C with classes* ».

Les apports du C++ par rapport au C

Le C++ a apporté par rapport au langage C les notions suivantes :

- les concepts orientés objet (encapsulation, héritage) ;
- les références ;
- la vérification stricte des types ;
- les valeurs par défaut des paramètres de fonctions ;
- la surcharge de fonctions (plusieurs fonctions portant le même nom se distinguent par le nombre et/ou le type de leurs paramètres) ;
- la surcharge des opérateurs (pour utiliser les opérateurs avec les objets) ;
- les templates de classes et de fonctions ;
- les constantes typées ;
- la possibilité de déclaration de variables entre deux instructions d'un même bloc.

Compilation

Définition de la compilation

La compilation consiste en une série d'étapes de transformation du code source en du code machine exécutable sur un processeur cible.

Le langage C++ fait partie des langages compilés : le fichier exécutable est produit à partir de **fichiers sources** par un **compilateur**.

Contrairement aux langages interprétés où il faut un logiciel interprétant le source, le fichier exécutable est produit pour une machine donnée : il est directement exécuté par le processeur. L'exécution est donc plus rapide.

Phases de la compilation

La compilation passe par différentes phases, produisant ou non des fichiers intermédiaires :

- **préprocessing** : Le code source original est transformé en code source brut. Les commentaires sont enlevés et les directives de compilation commençant par `#` sont d'abord traités pour obtenir le code source brut ;
- **compilation en fichier objet** : les fichiers de code source brut sont transformés en un fichier dit objet, c'est-à-dire un fichier contenant du code machine ainsi que toutes les informations nécessaires pour l'étape suivante (édition des liens). Généralement, ces fichiers portent l'extension `.obj` ou `.o` ;
- **édition de liens** : dans cette phase, l'éditeur de liens (linker) s'occupe d'assembler les fichiers objet en une entité exécutable et doit pour ce faire résoudre toutes les adresses non encore résolues, tant des mémoires adressées que des appels de fonction. L'entité exécutable est généralement soit un exécutable, soit une bibliothèque dynamique (DLLs sous Windows et toutes les variantes, tels que objet COM, OCX, etc, et les `.so` sous Linux).

Les compilateurs sont capables de générer des bibliothèques statiques, qui sont en quelques sortes le rassemblement d'un ensemble de fichiers objet au sein d'un même fichier. Dans ce cas, la phase d'édition de liens n'a pas eu lieu.

Cette découpe en phases permet de compiler séparément les bibliothèques en fichiers objets, et l'application et évite donc de tout re-compiler, ce qui prendrait beaucoup de temps pour les applications ayant un code source important.

Les fichiers sources

Les fichiers sources d'un programme C++ portent souvent l'extension `.cpp`, `.cxx`, `.cc`, parfois `.C` et sont des fichiers textes lisibles par le programmeur.

Les fichiers d'en-tête

Les fichiers "entêtes" ("headers" en anglais), traditionnellement d'extension `.h` ou `.hpp` (mais la plupart des entêtes systèmes du C++ standard n'ont plus d'extension du tout), contiennent généralement les prototypes de différentes fonctions, structures et classes. Ces prototypes proviennent :

- des bibliothèques standards du C++ : tout compilateur C++ doit fournir ces fichiers ainsi que les fichiers objets contenant l'implémentation des bibliothèques standards (souvent liées par défaut) ;
- de bibliothèques non standards fournis par l'éditeur du compilateur ou de l'environnement de développement ;
- de bibliothèques non standards (gratuites ou payantes) que le programmeur s'est procuré : citons par exemple la bibliothèque permettant d'accéder à une base de données MySQL. Lorsqu'une bibliothèque non standard est utilisée et que celle-ci englobe de nombreuses fonctionnalités (interface graphiques, accès à une base de données, surcouche système, communications réseaux, etc.), on parle parfois de framework. Citons par exemple Qt et WxWidgets dans cette catégorie. Un framework apparaît parfois comme une véritable surcouche du système d'exploitation.

Les fichiers .cpp

Les fichiers `.cpp` (parfois `.c` ou `.cc`) contiennent la définition (l'implémentation) des différentes fonctions et méthodes définies dans les fichiers d'en-tête. La compilation des fichiers `.cpp` produit dans un premier temps des fichiers objets (extension `.obj` ou `.o` en général).

A noter que ces fichiers `.cpp` utilisent les fichiers d'en-tête. Ils les appellent en utilisant la syntaxe `#include "nomdefichier"` (nomdefichier comprenant l'extension, donc, souvent ".h"). Comme indiqué ci-dessus, ces fichiers d'en-tête seront donc inclus complètement par le préprocesseur à l'intérieur du code source brut.

L'édition de liens (linking)

L'édition de liens est la phase finale de la compilation qui va rassembler tous les fichiers objets afin de former un fichier exécutable. Les fichiers objets proviennent :

- de la compilation de fichiers .cpp ;
- de la bibliothèque standard (ceux-ci sont souvent liés automatiquement) ;
- de bibliothèques ou framework extérieurs. Il faut dans ce cas explicitement dire à l'éditeur de liens quels sont les fichiers qu'il doit lier.

Les projets

Les makefiles

Afin de rendre possible le développement d'applications, les fichiers sources sont organisés sous forme de **projet**. Le C++ ayant été conçu sur les bases du C, il s'est ensuivi une gestion assez équivalente des projets. Avant l'avènement des environnements intégrés tels qu'on les connaît aujourd'hui, bon nombre de projets étaient construits sous forme de "makefile"s pris en charge par un outil spécifique make. De nos jours, cette technique est encore fortement utilisée dans les environnements UNIX et Linux. Il existe des alternatives plus modernes à make tels que CMake, automake, SCons, Boost.Build (bjam)...

Les environnements de développement intégrés

Les environnements de développement intégrés (EDI en français et IDE en anglais) sont des outils complets permettant de développer dans un certain langage de programmation. Ils contiennent en général :

- un compilateur ;
- un éditeur avec mise en évidence de la syntaxe (syntax highlighting) ;
- des outils facilitant la gestion d'un projet :
 - outil pour ajouter/supprimer/déplacer des sources,
 - outil pour paramétrer la compilation et l'édition de liens,
 - outil pour créer des modes de compilation (typiquement debug/release) et tous les paramètres y afférent ;
- des bibliothèques non standard censées aider le programmeur. Parfois, il s'agit d'un véritable framework ;
- des outils pour permettre le lancement de l'application ;
- des outils pour déboguer l'application ;
- des outils pour créer des interfaces graphiques ;
- ...

Quelques exemples

Multi-plateformes

- Eclipse, gratuit et Open Source, avec le plugin CDT, permet le développement en C/C++
- Code::Blocks, gratuit et Open Source (EDI complet fourni avec gcc mais utilisable avec d'autres compilateurs).
- CodeLite, gratuit et Open Source de base, seule exception : les plugins développés en annexe pour cet EDI peuvent être de n'importe quelle licence, y compris propriétaire et fermée.
- Qt Creator, gratuit et Open Source, dédié au framework Qt (mais n'impose pas son usage).

Sous windows

- Visual C++ de Microsoft (basé sur Visual studio).
- Visual C++ Express (disponible gratuitement et complètement utilisable).
- Borland C++ Builder.
- Devcpp (disponible gratuitement sous Windows).
- CLion de Jetbrains

Sous Linux

- Anjuta
- Kdevelop
- `g++` (Commun à tout système Unix, ligne de commande)
- clang (Plus rapide que GCC mais toutefois compatible, ligne de commande)

Sous MacOS X

- Apple Xcode (livré avec Mac OS X, utilise le compilateur clang).
- Metrowerks CodeWarrior.
- g++ (commun à tout système Unix, ligne de commande)

Un premier programme

Après ces quelques introductions, nous allons désormais pouvoir commencer à apprendre le C++.

Le fichier source

Tapez le fichier suivant dans un éditeur de texte :

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

int main()
{
    cout << "BONJOUR" << endl;

    return 0;
}
```

En général, un fichier source C++ utilise l'extension `.cpp`. On trouve également l'extension `.cc`. Le principal est de choisir l'extension une fois pour toute, pour la cohérence des fichiers.

Explications

La directive `#include`

La directive de compilation `#include <iostream>` permet d'inclure les prototypes des différentes classes contenues dans la bibliothèque standard `iostream`. Cette bibliothèque contient la définition de `cout` qui permet entre autre d'afficher des messages à l'écran.

L'espace de nommage standard

Un espace de nommage peut être vu comme un ensemble d'identifiants C++ (types, classes, variables etc.). `cout` fait partie de l'espace de nommage `std`. Pour parler de l'objet prédéfini `cout` de l'espace de nommage `std`, on peut écrire `std::cout`. Cette notation est assez lourde car elle parseme le code de `std::`. Pour cela on a écrit `using namespace std` qui précise que, par défaut, la recherche s'effectuera aussi dans l'espace de nommage `std`. On pourra donc alors écrire tout simplement `cout` pour parler de `std::cout`. On dit que `std` est un espace de nom (*namespace* en anglais).

La fonction `main`

Tout programme en C++ commence par l'exécution de la fonction `main`. Il se termine lorsque la fonction `main` est terminée. La fonction `main` peut être vue comme le point d'entrée de tout programme en C++. Cette fonction renvoie un entier, très souvent 0, qui permet d'indiquer au système d'exploitation que l'application s'est terminée normalement.

L'objet `cout`

Il permet d'envoyer des caractères vers le flux de sortie standard du programme, c'est-à-dire l'écran pour ce programme (sa fenêtre console). Il permet donc d'afficher des messages à l'écran. En utilisant l'opérateur `<<`, on peut écrire une chaîne de caractères à l'écran. L'instruction `cout << "BONJOUR"` ; affiche donc le message `BONJOUR` à l'écran.

`return 0`

Cette instruction (facultative ici) indique que la fonction `main` est terminée et que tout s'est bien passé. Nous verrons plus loin ce que veut dire exactement l'instruction `return`, mais même si elle est facultative il est fortement recommandé de la mettre, que ce soit par simple souci de conformité ou du fait que votre programme est censé renvoyer une valeur à la fin de son exécution.

Exécution

Si on compile et exécute ce programme, le message `BONJOUR` s'affiche à l'écran.

Il est possible que vous ne voyez qu'une fenêtre noire "flasher" si vous êtes sous Windows . Pour résoudre ce problème, ajoutez avant le "return 0" l'instruction :

```
system("pause");
```

Les commentaires


Tout bon programme a des fichiers sources bien commentés pour expliquer comment cela fonctionne et pourquoi certains choix ont été faits. Ce qui évite une perte de temps lorsque le code source est repris et modifié, soit par un autre développeur, soit par l'auteur qui ne se souvient pas forcément de son projet s'il n'y a pas touché depuis longtemps.

Bloc de commentaire

Un bloc de commentaire est délimité par les signes slash-étoile `/*` et étoile-slash `*/` comme en Java et en C#. Exemple :

```
/*
  Un commentaire explicatif
  sur plusieurs lignes...
*/
```

Les blocs ne peuvent être imbriqués car dès que le compilateur trouve slash-étoile `/*`, il recherche la première occurrence d'étoile-slash `*/` terminant le commentaire.

 Ce code contient **une erreur volontaire** !

```
/* : début du commentaire
   /* : ignoré
   fin du commentaire : */
erreur ici car le commentaire est fini : */
```

Commentaire de fin de ligne

Un commentaire de fin de ligne débute par un double slash `//` et se termine au prochain retour à la ligne. Exemple :

```
x++; // augmenter x de 1
```

Astuce : La majorité des éditeurs évolués (Visual Studio, Borland C++, Eclipse ...) utilisent ce type de commentaire pour les commandes commenter/décommenter le groupe de lignes sélectionnées.

Le préprocesseur

Avant de compiler le programme, il est possible d'effectuer certaines modifications sur le code source. Le programme effectuant ces modifications s'appelle le préprocesseur. Les commandes destinées au préprocesseur commencent toutes par # en début de ligne.

Inclusion de fichiers

Pour inclure un fichier à un certain endroit dans le fichier source, on écrit :

```
#include "nom_du_fichier"
```

Le contenu du fichier `nom_du_fichier` est alors inséré dans le fichier source.

Le nom du fichier peut être écrit entre guillemets "`nom_du_fichier`" ou entre chevrons `<nom_du_fichier>`. Dans le premier cas, cela signifie que le fichier se trouve dans le même dossier que le fichier source, tandis que dans le deuxième cas, il s'agit d'un fichier se situant dans un endroit différent (ce fichier pouvant être fourni par le compilateur ou une librairie externe par exemple).

Exemple :

```
#include <iostream>
```

Le fichier C++ standard `iostream` est inclus à cet endroit-là dans le code. Il contient la définition de certains objets standards notamment `cin` et `cout`.

#define, #undef

La directive `#define` permet de remplacer toutes les occurrences d'un certain mot par un autre. Par exemple :

```
#define N 1143
```

Sur cet exemple toutes les occurrences de `N` seront remplacées par `1143`. Cela est parfois utilisé pour définir des constantes. On préférera toutefois utiliser le mot-clé `const`.

On peut très bien ne pas fixer de valeur et écrire :

```
#define PLATEFORME_INTEL
```

La variable de compilation `PLATEFORME_INTEL` est ici définie. Combiné à `#ifdef`, on pourra compiler ou non certaines parties du code à certains endroits du programme.

De la même façon que l'on peut définir une variable, on peut arrêter une définition en utilisant `#undef`. Son utilisation est rare, mais peut servir à ne plus définir une variable de compilation. Par exemple:

```
#undef PLATEFORME_INTEL
```

#ifdef, #ifndef, #if, #endif et #else

Présentation

Toutes ces directives permettent la compilation conditionnelle. C'est-à-dire que la partie du code comprise entre la directive conditionnelle (`#ifdef`, `#ifndef` ou `#if`) et la fin du bloc signalée par la directive `#endif` n'est compilée que si la condition est remplie.

- La directive `#ifdef` permet de compiler toute une série de lignes du programme si une variable de compilation a précédemment été définie (par la directive `#define`). La directive `#endif` indique la fin de la partie de code conditionnelle. La partie du programme compilée sera toute la partie comprise entre le `#ifdef` et le prochain `#endif`.
- La directive `#ifndef` permet de compiler un bout de programme si une variable de compilation n'est **pas**

définie. C'est donc l'inverse de `#ifndef`. La fin de la partie à inclure est déterminée également par `#endif`.

- La directive `#if` permet de tester qu'une expression est vraie. Cette expression ne peut utiliser que des constantes (éventuellement définies par une directive `#define`), et la fonction `defined` permettant de tester si une variable de compilation existe.

Il faut noter que les directives `#ifdef` et `#ifndef`, bien que très largement utilisées, sont considérées comme dépréciées ("deprecated"). On préférera donc la syntaxe : `#if defined(MA_VARIABLE)` à `#ifdef MA_VARIABLE` et `#if !defined(MA_VARIABLE)` à `#ifndef MA_VARIABLE`.

- Chacune de ces conditions peut être accompagné d'une directive `#else` qui permet d'inclure un bout de programme si la condition n'est pas vérifiée.

Il ne faut pas abuser de ces directives et elles sont surtout utilisées :

- pour gérer des problèmes de portabilité.
- au début des fichiers d'en-tête pour éviter une double compilation.
- dans les fichiers d'en-tête de DLL sous Windows.

Exemples

Exemple 1

```
#include <iostream>
using namespace std;
#define FRENCH

int main()
{
#ifdef FRENCH
    cout << "BONJOUR";
#else
    cout << "HELLO";
#endif
    return 0;
}
```

Dans ce programme, il suffit d'effacer `#define FRENCH` et de recompiler le programme pour passer d'une version française à une version anglaise. Ceci pourrait être utile si le programme comporte 10 000 lignes (ce qui est faible pour un programme réel). Bien évidemment, il existe bien d'autres façons de gérer le multilinguisme en C++.

Exemple 2

```
// Définir la taille d'un tableau contenant le prix :
// - de 5 variétés d'orange
#define N_ORANGES 5
// - de 3 variétés de pommes
#define N_POMMES 3

#define N_TOTAL_FRUITS N_ORANGES+N_POMMES

double prix_fruits[N_TOTAL_FRUITS];

#if N_TOTAL_FRUITS > 7
    // ... plus de 7 variétés de fruits
#else
    // ... moins de 7 ou égale à 7 variétés de fruits
#endif
```

Exemple 3

Fichier `toto.h`

```
#ifndef TOTO_H
```

```
#define TOTO_H
... écrire ici les prototypes ...
#endif
```

Le problème :

Imaginons qu'un fichier header **toto.h** contienne le prototype d'une certaine classe ou d'une fonction. Imaginons que le programme contiennent 3 autres fichiers headers nommés A.h, B.h et C.h qui ont tous les 3 besoin des prototypes inclus dans toto.h, ces 3 fichiers vont commencer par `#include "toto.h"`. Imaginons également que C.h a besoin des prototypes inclus dans A.h et B.h. C.h va donc commence par `#include "A.h"` et `#include "B.h"`. Le problème est que le fichier *toto.h* va être inclus plusieurs fois. Le compilateur va alors refuser de compiler le programme en indiquant que plusieurs prototypes d'une même fonction sont inclus.

Solution :

Pour résoudre l'inclusion multiple de fichier headers (inévitable), on va faire commencer le fichier header par `#ifndef TOTO_H` et il se termine par `#endif`. Si la variable de compilation *TOTO_H* n'est pas définie, alors le header sera inclus, sinon, il sera tout simplement vide. Juste après `#ifndef TOTO_H`, nous allons écrire `#define TOTO_H` définissant justement cette variable de compilation *TOTO_H*.

La première fois que le header sera inclus, *TOTO_H* n'est pas défini, le header normal sera donc inclus. `#define TOTO_H` définira alors la variable de compilation *TOTO_H*. La deuxième fois que ce même header sera inclus, et les fois suivantes, *TOTO_H* sera défini et par conséquent, le header sera vide. les prototypes n'auront donc été inclus qu'une seule fois. Le tour est joué. Il faut donc faire commencer systématiquement (c'est tout du moins conseillé) tous les fichiers header par les 2 lignes `#ifndef ...` et `#define ...` et les faire se terminer par `#endif`.

Autre solution

Toutefois, il existe une autre solution au problème précédent en utilisant la directive suivante en début de fichier .h :

```
#pragma once
```

Cette directive indique au compilateur d'ignorer le fichier s'il a déjà été "visité", et ne fonctionne qu'avec certains compilateurs :

- Visual C++
- CodeWarrior
- GCC for Darwin

Les macros

Présentation

Les macros sont des `#define` particulier parce qu'ils contiennent des paramètres. Ainsi si vous écrivez :

```
#define AFFICHE(x) cout << x << endl;
```

Alors vous pouvez écrire `AFFICHE("BONJOUR")` et le préprocesseur modifiera cette ligne et la transformera en `cout << "BONJOUR" << endl;`. Il y aura substitution de *x* par "BONJOUR". Il ne faut pas abuser des macros et très souvent l'utilisation de fonctions, notamment les fonctions inline, est préférable.

Exemple

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

#define AFFICHER(x) cout << x << endl;

int main()
{
    AFFICHER("BONJOUR");
    return 0;
}
```

Bonnes pratiques

Afin d'utiliser correctement les macros, il est préférable de les afficher clairement et de les rendre suffisamment flexible à différentes utilisations. Si la macro est constituée de plusieurs instructions séparés par des `;`, il est préférable d'écrire la macro sur plusieurs lignes afin d'accroître sa lisibilité. Pour indiquer à une macro que sa définition continue sur la ligne suivante, il suffit d'indiquer un antislash (`\`) en dernier caractère de la ligne.

```
#define AFFICHER(x) \  
    cout << x; \  
    cout << endl;
```

L'utilisation la plus courante des macros est de ne pas mettre de `;` à la fin de celle-ci, mais de le mettre dans le code, là où elle est utilisée. En effet, on peut prévoir qu'une macro soit utilisable en tant qu'instruction simple, ou en tant que condition ou paramètre de fonction où l'on ne doit pas mettre de `;`. Pour les macros qui ne retournent rien (comme la macro `AFFICHER` dans l'exemple précédent), le placement du `;` n'est pas un problème car elles ne retournent rien et ne seront jamais utilisées dans une condition ou un appel de fonction.

```
#include <iostream>  
using namespace std;  
  
#define DIVISER(x, y) ((x) / (y))  
  
int main()  
{  
    int valeur = DIVISER(5, 3);  
  
    if (DIVISER(8, 2) == 4)  
        cout << DIVISER(1.0, 5.0) << endl;  
  
    return 0;  
}
```

Il est fortement conseillé de toujours utiliser un paramètre de macro avec des parenthèses autour. Si l'on reprend l'exemple précédent sans parenthèses :

```
#include <iostream>  
using namespace std;  
  
#define DIVISER(x, y) x / y  
  
int main()  
{  
    int valeur = DIVISER(5, 3);  
  
    if (DIVISER(4 + 4, 2) == 4)  
        cout << DIVISER(1.0, 5.0) << endl;  
  
    return 0;  
}
```

Ici, le résultat obtenu n'est pas forcément celui désiré. `DIVISER(4 + 4, 2)` sera traduit après la précompilation par `4 + 4 / 2`. Ceci donne pour valeur `4 + 2`, soit 6. Ajouter un maximum de parenthèses permet de s'assurer de la validité de la macro sous plusieurs utilisations différentes. Ainsi, dans l'exemple précédent, une utilisation de parenthèses dans la macro (`#define DIVISER(x, y) ((x) / (y))`), aurait traduit `DIVISER(4 + 4, 2)` en `((4 + 4) / (2))`. Ceci aurait donné comme valeur `8 / 2 = 4`, la valeur attendue.

Les types de base et les déclarations

Déclarations, types et identificateurs

Les variables

Comme la plupart des langages de programmation, le C++ utilise la notion de variable. Une variable peut être vue comme une zone de la mémoire qui comprend une certaine valeur.

Les types en C++ et les systèmes de représentation

Le langage C++ impose un mécanisme de type pour indiquer la nature des données contenues dans une variable. Ainsi un `double` permettra de stocker un réel et un `int` permettra de stocker un entier. Par contre, il ne définit pas de système de représentation pour représenter ces variables. Ainsi, le standard ne spécifie pas comment on représente un double sous la forme d'une suite de bits. Le système de représentation utilisé peut donc varier entre deux ordinateurs ou en fonction du compilateur utilisé. Cette particularité peut parfois poser de graves problèmes de portabilité d'un programme.

Les déclarations

Toute variable en C++ doit être déclarée : la déclaration indique l'identificateur de la variable (son nom) et sa nature (son type).

Syntaxe :

```
type identificateur;
```

Exemple :

```
int a;
```

Cette déclaration définit une variable d'identificateur `a` qui contient un entier de type `int`.

Identificateurs valides

Un identificateur est une suite de caractères (pouvant être majuscules ou minuscules), de chiffres ou d'underscores (underscore ou "tiret bas" est le caractère `_`). Cette suite ne peut pas commencer par un chiffre. Un identificateur ne peut contenir ni espace, ni tiret - (utilisé pour l'opération de soustraction).

Les types entiers

Le langage C++ possède plusieurs types de base pour désigner un entier.

- **int** : contient un entier de taille normale, positif ou négatif.
- **short int** : contient un entier de petite taille, positif ou négatif.
- **long int** : contient un entier de grande taille (32 bits), positif ou négatif.
- **long long int** : contient un entier de plus grande taille (64 bits), positif ou négatif.
- **unsigned int** : contient un entier de taille normale, positif ou nul.
- **unsigned short int** : contient un entier de petite taille, positif ou nul.
- **unsigned long int** : contient un entier de grande taille (32 bits), positif ou nul.
- **unsigned long long int** : contient un entier de plus grande taille (64 bits), positif ou nul.

La longueur d'un `long int`, d'un `int` et d'un `short int` n'est pas spécifié par le langage. Plus un entier est représenté sur un grand nombre de bits, plus il pourra être grand. Ainsi, il est usuel de représenter un `int` sur 32 bits : il peut alors représenter n'importe quel entier entre -2^{31} et $2^{31}-1$. Le langage impose juste que la taille d'un `long int` doit être supérieure ou égale à celle d'un `int` et que la taille d'un `int` doit être supérieure ou égale à celle d'un `short int` !

Le système de représentation utilisé non plus. En général, la base 2 est utilisée pour les types unsigned int, unsigned long int et unsigned short int et le complément à 2 est utilisé pour les types int, long int et short int. Ce n'est toutefois nullement obligatoire.

Interprétation des constantes entières

- si une constante commence par 0x, elle sera interprétée comme une valeur en hexadécimal (base 16).
- si une constante commence par 0 suivi d'un chiffre, elle sera interprétée comme valeur en octal (base 8).
- dans le cas contraire, elle sera interprétée comme étant en base 10.

Exemples :

98 représente 98 en base 10.

0x62 représente 98 en hexadécimal ($6*16+2$).

0142 représente 98 en octal ($1*64+4*8+2$).

- si une constante entière se termine par un U, elle sera interprétée comme étant un unsigned.
- si une constante entière se termine par un L, elle sera interprétée comme un long.
- si une constante entière se termine par LL, elle sera interprétée comme un long long.

Exemples

- 78U représente le unsigned int valant 78.
- 78L représente le long int valant 78.
- 78LL représente le long long int valant 78.
- 78ULL représente le unsigned long long int valant 78.

Les types réels

Pour représenter un réel, il existe 3 types de base :

- float (simple précision)
- double (double précision)
- long double (précision étendue)

Le langage ne précise pas ni le système de représentation, ni la précision de ces différents formats. Le type long double est juste censé être plus précis que le double, lui-même plus précis que le float.

Il est toutefois usuel (mais non obligatoire) de représenter le float sur 32 bits dans le format IEEE 754 simple précision et le double sur 64 bits dans le format IEEE 754 double précision.

Interprétation des constantes réelles

- On peut écrire un réel sous la forme 1.87 ou .56 ou 8. ou $7.6e10$ ou $5.4e-3$ ou encore $10.3E+2$.
- Une constante qui se termine par un f ou un F sera interprétée comme un float.
- Une constante qui se termine par un l ou un L sera interprétée comme un long double.

Exemples :

- 3.65L représente le long double valant 3.65.
- 3.65F représente le float valant 3.65.

Les caractères

Le caractère est l'élément de base de tout texte et donc sans doute de toute pensée. Les changements de l'informatique ont conduit à différentes approches pour représenter un caractère sur cinq, sept, huit, seize, dix-sept ou trente-deux bits.

Aujourd'hui, les deux types incontournables sont le type char hérité du C, et ceux relatifs à Unicode.

Ce chapitre traite des types de bases utiles pour représenter des caractères. Toutefois les caractères sont rarement utilisés seuls, surtout dans les applications Unicode. A ce sujet, vous trouverez dans la suite de cet ouvrage les chapitres suivants :

- [Programmation C++/La_libririe_standard#Les chaînes de caractères](#)
- [Programmation C++/Les_tableaux#Les tableaux de caractères](#)

Le type char

Il s'agit du type historique pour représenter un caractère. Bien que le type char fasse penser à un caractère (*character* en anglais), il désigne souvent de facto un octet, qui peut être signé ou non signé suivant le compilateur. C'est l'un des concepts que le C++ a repris du langage C. Aucun système de représentation n'est imposé pour les caractères et on utilise en général des dérivés (8 bits) du code ASCII (qui est un code sept bits). ^[1] Usuellement, le type char est exactement 8 bits, ce qui fait que c'est le seul type utilisé pour représenter un octet, créant ainsi une confusion entre caractère et octet. Le chapitre suivant s'intéresse au huitième bit.

Exemple :

```
char c;
c = 'E';
```

Le caractère E est transféré dans la variable c de type char .

On peut transférer un char dans un int pour récupérer son codage.

Ainsi, on peut écrire :

```
int a;
char b;
b = 'W';
a = b;
```

On récupère alors dans a le codage du caractère 'W'. On récupérera donc en général le code ASCII de 'W' dans la variable a.

Aujourd'hui, il est désuet de considérer que l'on code tous caractères sur un seul char. Le type char reste cependant incontournable car il est souvent utilisé pour désigner un octet.

Transformation de majuscule en minuscule

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

int main()
{
    char a, b;
    cout<<"Tapez un caractere : "; cin>>a;
    if (a>='A' && a<='Z') {
        cout<<"Vous avez tapé une majuscule."<<endl;
        b = a + ('a'-'A');
        cout<<"La minuscule correspondante est "<< b <<endl;
    }
    else if (a>='a' && a<='z') {
        cout<<"Vous avez tapé une minuscule."<<endl;
        b = a + ('A'-'a');
        cout<<"La majuscule correspondante est "<< b <<endl;
    }
    else cout<<"Vous n'avez pas tapé une lettre."<<endl;
    return 0;
}
```

■ Explications

- On demande à l'utilisateur de taper un caractère dans une variable a.
- Si l'utilisateur a tapé une majuscule, on affiche la minuscule correspondante.
- Si l'utilisateur a tapé une minuscule, on affiche la majuscule correspondante.

■ Exécution 1

Tapez un caractère : **H**

Vous avez tapé une majuscule.

La minuscule correspondante est h.

■ Exécution 2

Tapez un caractère : **w**

Vous avez tapé une minuscule.
La majuscule correspondante est W.

▪ Exécution 3

Tapez un caractère : **9**
Vous n'avez pas tapé une lettre.

Les types signed char et unsigned char

Lorsqu'on transfère un char dans un int, peut-on récupérer une valeur négative ? La réponse est oui si on utilise le type **signed char** et non si on utilise le type **unsigned char**. Ces types peuvent être utile lorsqu'on manipule des caractères non ASCII.

Pour les données de type char, lorsque ni signed ni unsigned ne sont précisés, le choix entre les deux est fait par le compilateur. Dans tous les cas à l'époque où seuls les codages ASCII et autres codages ISO-646 étaient utilisés cela n'avait pas d'importance.

Aujourd'hui cependant, quasiment tous les codages de caractères utilisent a minima huit bits. C'est notamment le cas d'UTF-8. L'éventuel bit de signe doit donc être considéré pour permettre la portabilité du logiciel.

Les types char16_t and char32_t

À partir de C++11 (C++ norme de 2011) trois types de chaînes de caractères sont prise en charge: UTF-8, UTF-16, et UTF-32. Le type **char** conserve ses unités de codage de huit bits pour le codage des caractères Unicode via UTF-8, les nouveaux types **char16_t** et **char32_t** sont des unités de codage de seize ou trente-deux bits pour le codage des caractères Unicode via UTF-16 ou UTF-32.

Ces types sont standard à partir de C++2011 mais n'existent pas sur des compilateurs plus anciens, ni même sur les compilateurs C-2011.

Le type wchar_t

Ce type de caractère n'existe qu'avec les compilateurs supportant l'Unicode (jeu de caractère international standard couvrant les langues du monde entier). Ces caractères sont stockés sur 2 octets ou 4. Les valeurs constantes de caractère (entre simple quote) ou de chaîne de caractères (entre double quote) doivent alors être précédées du caractère L.

Exemple :

```
wchar_t a = L'é'; // caractère 'é' unicode(16 ou 32 bits);
wchar_t[] chaîne = L"Bonjour, monde !"; // chaîne de caractère unicode
```

Ce type présente le problème de ne pas être standard: certaines implémentations n'offrent que 16 bits soit une portion limitée des caractères Unicode.

API exceptionnellement Unicode

Certaines API C++ telle que Visual C++ sous Windows sont paramétrables par une option dite *unicode* ou *non*. Pour cela elles se basent sur le type de caractère TCHAR que le compilateur interprète (en fait remplace) par **char** ou **wchar_t** selon l'option Unicode. Dans ce cas, les valeurs constantes de chaînes et de caractères doivent être encadrées par la macro `_T`. Cette macro peut alors faire précéder les constantes d'un caractère L ou non.

Exemple :

```
TCHAR a = _T('é'); // caractère 'é' unicode ou huit bits;
TCHAR[] chaîne = _T("Bonjour, monde !"); // chaîne de caractère unicode ou ascii (huit bits)
```

Les booléens

Le C++ utilise le type **bool** pour représenter une variable booléenne. Un bool ne peut prendre que 2 valeurs : **true** ou **false**. On peut affecter à un bool le résultat d'une condition, ou l'une des deux constantes citées précédemment.

Exemple :

```
bool a;
```

```
int c;
c = 89;
a = (c > 87);
// a reçoit alors la valeur true.
```

L'opérateur const

L'opérateur `const` placé devant une déclaration de variable décrit celle-ci comme constante; elle ne peut être changée.

Exemple :

```
int x = 2;
const int y = 2;
x += 1; // x recevra la valeur 3
y += 1; // Cette opération est interdite par un compilateur C++ conforme
```

Dans cet exemple, nous déclarons deux variables `x` et `y`, chacune avec la même valeur. La première variable (`x`) peut être modifiée, ce qui n'est pas le cas de la seconde (`y`) qui est déclarée `const`. Le compilateur refusera toute opération qui tenterait d'en modifier son contenu (opérations d'assignation et d'incrémentations `=`, `+=`, `-=`, `*=`, `/=`, `>>=`, `<<=`, `++`).

Il en va de même pour les objets. Un objet déclaré `const` ne pourra pas être modifié, c'est-à-dire que le compilateur refusera l'invocation d'une méthode non `const` sur cet objet.

Exemple :

```
class X {
public:
    X() : valeur_(0) {}
    explicit X(int valeur) : valeur_(valeur) {}
    void annule() { this->valeur_ = 0; }
    void init(int valeur) { this->valeur_ = valeur; }
    int valeur() const { return this->valeur_; }
private:
    int valeur_;
};

X x(2)
const X y(5);
x.annule(); // Ok
y.annule(); // Erreur de compilation.
```

Dans cet exemple, deux objets `x` et `y` ont été déclarés. Il est possible d'invoquer n'importe quelle méthode (`const` ou non) sur l'objet `x`, par contre seules les méthodes `const` peuvent être invoquées sur l'objet `y` puisqu'il est déclaré `const`.

L'opérateur sizeof

L'opérateur `sizeof` permet de savoir le nombre d'octets qu'occupe en RAM une certaine variable. On peut écrire `sizeof(a)` pour savoir le nombre d'octets occupé par la variable `a`. On peut aussi écrire, `sizeof(int)` pour connaître le nombre d'octets occupés par une variable de type `int`. Cet opérateur est très utile lorsqu'on veut résoudre des problèmes de portabilité d'un programme.

Plus précisément `sizeof(char)` vaut toujours 1, par définition. Sur la plupart des architectures un `char` est codé par huit bits, soit un octet. [Programmation C/Types de base#Caractères](#)

Définir un alias de type

L'instruction `typedef` permet de définir un alias pour un type de données. Ceci permet dans certains cas de raccourcir le code, et dans tous les cas c'est l'occasion de donner un nom plus explicite à un type. Ceci favorise une meilleure lecture du code.

La syntaxe de `typedef` est exactement la même que celle de la déclaration d'une variable, excepté que l'instruction commence par `typedef` et qu'aucune variable n'est réservée en mémoire, mais un alias du type est créé.

Exemple:

```
typedef unsigned long data_size;  
data_size readData(char* buffer, data_size buffer_size);
```

On s'aperçoit plus facilement que la fonction retourne le nombre d'octets lus que dans la déclaration sans `typedef` :

```
unsigned long readData(char* buffer, unsigned long buffer_size);
```

Voir aussi

- [Exercices](#)

Les opérations de base

Les C++ possède un grand nombre d'opérateurs de base effectuant entre autre des opérations arithmétiques et capables de travailler sur les entiers, les réels, etc... Nous allons présenter ici ces principaux opérateurs.

L'affectation

Syntaxe

```
identificateur=expression
```

Sémantique

On commence par évaluer l'expression et on met le résultat dans la variable *identificateur*.

Exemple

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

int main()
{
    int a, b, s;
    cout << "Tapez la valeur de a : "; cin >> a;
    cout << "Tapez la valeur de b : "; cin >> b;

    s = a + b; // affecter le résultat de l'addition à la variable s

    cout << "La somme a+b vaut : " << s << endl;
    return 0;
}
```

Exécution

```
Tapez la valeur de a : 45
Tapez la valeur de b : 67
La somme a+b vaut 112
```

Explications

Dans ce programme, on déclare 3 variables a, b et s. On demande à l'utilisateur du programme de taper la valeur de a puis la valeur de b. **cout** sert à l'affichage à l'écran et **cin** à la saisie au clavier. Le programme calcule ensuite dans la variable s la somme a+b. On affiche finalement la valeur de s.

Affectation à la déclaration

L'affectation à la déclaration d'une variable est appelée "déclaration avec initialisation", car est un cas particulier de l'affectation (en particulier avec les objets).

Exemple :

```
int a = 57;
```

Affectation en série

Le résultat d'une même expression peut être assigné à plusieurs variables sans la réévaluer, en une seule instruction :

```
identificateur2=identificateur1=expression
```

Par exemple :

```
a = b = 38 + t;
```

En fait, l'affectation retourne une valeur : celle affectée à la variable. L'exemple précédent est donc équivalent à :

```
a = (b = 38 + t);
```

L'opération se déroule de la manière suivante :

1. Le résultat de $38+t$ est calculé ;
2. Il est affecté à la variable b ;
3. Il est retourné par l'opérateur d'affectation ;
4. Cette valeur retournée est affectée à la variable a .

Remarque importante

Pour effectuer un test de comparaison, par exemple comparer a à 53 , il ne faut pas écrire `if(a=53)` mais `if(a==53)` en utilisant 2 fois le symbole `=`. Une erreur classique !

Opérations arithmétiques

sur les entiers

On peut effectuer les opérations arithmétiques usuelles sur les entiers en utilisant les opérateurs `+`, `-`, `*` et `/`. Il faut juste avoir en tête que la division sur les entiers effectue une troncature (la partie décimale est perdue). Le modulo s'obtient en utilisant l'opérateur `%`. Ainsi `a%b` désigne le reste de la division de a par b . On peut utiliser les parenthèses pour fixer l'ordre d'évaluation des expressions. On peut aussi utiliser l'opérateur `++` pour incrémenter une variable de 1. L'opérateur `--`, quant à lui, décrémente une variable de 1.

sur les réels

Sur les réels, on utilise les opérateurs `+`, `-`, `*` et `/` pour effectuer les 4 opérations de base. Il faut avoir en tête que toute opération sur les réels est entachée d'une minuscule erreur de calcul : il s'agit d'un arrondi sur le dernier bit. Si on effectue de nombreux calculs, cette erreur peut s'amplifier et devenir extrêmement grande.

Les fonctions mathématiques habituelles sont présentes dans la bibliothèque standard `<cmath>`.

Exemple

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

int main()
{
    int a, b, M;

    cout << "Tapez la valeur de a : "; cin >> a;
    cout << "Tapez la valeur de b : "; cin >> b;

    M = (a+3) * (6+a) + (b-5) * 2;

    cout << "M vaut " << M << endl;

    return 0;
}
```

```
}
-----
```

Exécution :

```
Tapez la valeur de a : 1
Tapez la valeur de b : 2
M vaut 22
-----
```

Opérations binaires

Les opérations présentées dans cette section opèrent au niveau des bits.

Les décalages

Les décalages permettent de décaler vers la droite ou vers la gauche toute la représentation en binaire d'une valeur d'un certain nombre de bits.

Syntaxe	Sémantique
<i>valeur</i> << <i>decalage</i>	Décale la valeur de <i>decalage</i> bits vers la gauche.
<i>valeur</i> >> <i>decalage</i>	Décale la valeur de <i>decalage</i> bits vers la droite.

Exemple

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

int main()
{
    int a = 5;
    int b;

    b = a << 3;
    cout << "b vaut : " << b << endl;

    b = (a+1) << 3;
    cout << "b vaut : " << b << endl;

    return 0;
}
-----
```

Exécution

```
b vaut 40
b vaut 48
-----
```

Le ET binaire

Syntaxe

```
exp1 & exp2
-----
```

Sémantique

Le résultat est obtenu en faisant un ET logique sur chaque bit de la représentation de exp1 avec le bit correspondant de la représentation de exp2. Le ET logique a pour résultat 1 si les deux bits correspondants en entrée sont à 1 (pour 1-1), et sinon 0 (pour 0-1, 1-0 ou 0-0).

Exemple

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

int main()
{
    int a = 53;
    int b = 167;
    int c = a & b;
    cout << "c vaut " << c << endl;

    return 0;
}
```

Exécution

```
c vaut 37
```

Explications

```
53 s'écrit en binaire 0000 0000 0000 0000 0000 0000 0011 0101
167 s'écrit en binaire 0000 0000 0000 0000 0000 0000 1010 0111
ET binaire           0000 0000 0000 0000 0000 0000 0010 0101
==> le résultat vaut 37 en décimal
```

Le OU binaire

Syntaxe

```
exp1 | exp2
```

Sémantique

Le résultat est obtenu en faisant un OU logique sur chaque bit de la représentation de exp1 avec le bit correspondant de la représentation de exp2. Le OU logique a pour résultat 1 si l'un des deux bits correspondant en entrée est à 1 (pour 1-1, 1-0 ou 0-1), et sinon 0 (pour 0-0).

Exemple

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

int main()
{
    int a = 53;
    int b = 167;
    int c = a | b;
    cout << "c vaut " << c << endl;

    return 0;
}
```

Exécution

```
c vaut 183
```

Explications

```

53 s'écrit en binaire 0000 0000 0000 00000 0000 0000 0011 0101
167 s'écrit en binaire 0000 0000 0000 00000 0000 0000 1010 0111
OU binaire           0000 0000 0000 00000 0000 0000 1011 0111
==> le résultat vaut 183 en décimal
    
```

Le OU exclusif binaire

Syntaxe

```
exp1 ^ exp2
```

Sémantique

Le résultat est obtenu est faisant un OU exclusif sur chaque bit de la représentation de exp1 avec le bit correspondant de la représentation de exp2. Le OU exclusif logique a pour résultat 1 si les deux bits correspondant en entrée sont différents (pour 1-0 ou 0-1), et sinon 0 (pour 1-1 ou 0-0).

Exemple

```

#include <iostream>
using namespace std;

int main()
{
    int a = 53;
    int b = 167;
    int c = a ^ b;
    cout << "c vaut " << c << endl;

    return 0;
}
    
```

Exécution

```
c vaut 146
```

Explications

```

53 s'écrit en binaire 0000 0000 0000 00000 0000 0000 0011 0101
167 s'écrit en binaire 0000 0000 0000 00000 0000 0000 1010 0111
OU exclusif binaire   0000 0000 0000 00000 0000 0000 1001 0010
==> le résultat vaut 146 en décimal
    
```

Le NON binaire

Syntaxe

```
~exp
```

Sémantique

Le résultat est obtenu en inversant tous les bits de la représentation de exp. Le 1 devient 0 et *vice versa*.

Exemple

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

int main()
{
    int a = 53;
    int b = ~a;
    cout << "b vaut " << b << endl;
    return 0;
}
```

Exécution

```
b vaut -54
```

Explications

```
53 s'écrit en binaire  0000 0000 0000 0000 0000 0000 0011 0101
NON binaire           1111 1111 1111 1111 1111 1111 1100 1010
==> le résultat vaut -54 en décimal (signed int)
      ou 4294967242 (unsigned int)
```

Opérations booléennes

Comparaisons usuelles

Les comparaisons usuelles s'effectuent selon la syntaxe *a symbole b*, *symbole* désignant le test effectué :

- Le symbole > désigne le test *strictement supérieur à*.
- Le symbole >= désigne le test *supérieur ou égal à*.
- Le symbole < désigne le test *strictement inférieur à*.
- Le symbole <= désigne le test *inférieur ou égal à*.
- Le symbole == désigne le test d'égalité.
- Le symbole != désigne le test différent.

Le ET logique

Syntaxe

```
condition1 && condition2
```

Sémantique

Le ET logique est vrai si à la fois les condition1 et condition2 sont vraies. Il est faux dans le cas contraire.

Le OU logique

Syntaxe

```
condition1 || condition2
```

Sémantique

Le OU logique est vrai si au moins une des 2 conditions *condition1* ou *condition2* est vraie. Il est faux dans le cas contraire.

Le NON logique

Syntaxe

```
!(condition)
```

Sémantique

Le NON logique inverse la valeur de *condition*: si *condition* vaut `true` le résultat vaut `false`. Si *condition* vaut `false` le résultat vaut `true`.

Exemples d'utilisation des opérateurs booléens

```
bool b;
int u = 18;
b = !( (u>20 || (u<0)));
```

b vaut `true`.

```
bool b = ! false ; // -> b vaut true
b = ! true ; // -> b vaut false
```

Affectation avec opérateur

Il existe toute une gamme d'opérateurs permettant d'effectuer une opération (une addition par exemple) avec le contenu d'une variable et de mettre le résultat dans cette même variable. Ainsi il sera plus pratique d'écrire `b += a ;` que d'écrire `b = b + a ;`.

Une opération du type

```
a opérateur= expression ;
```

équivalent à :

```
a = a opérateur (expression) ;
```

L'opérateur +=

Syntaxe

```
identificateur+=expression
```

Sémantique

Cet opérateur ajoute à la variable *identificateur* la valeur de *expression* et stocke le résultat dans la variable *identificateur*.

Exemple

```
#include <iostream>
using namespace std;
int main()
{
```

```
int a = 80;
int b = 70;
b += a;
cout << "La valeur de b est : " << b << endl;
return 0;
}
```

Exécution

```
La valeur de b est 150
```

Autres opérateurs

Sur le même modèle, on peut utiliser les opérateurs suivants : -=, *=, /=, %=, >>=, <<=, &=, |= et ^=.

Priorité des opérateurs

Comme en mathématique, tous les opérateurs n'ont pas la même priorité.

Par exemple, l'expression $1+2*3$ retournera la valeur 7 car l'opérateur `*` a une plus grande priorité et est évalué avant l'opérateur `+`. L'expression est donc équivalent à $1+(2*3)$.

Les parenthèses permettent de modifier les priorités en encadrant ce qu'il faut évaluer avant. Ainsi l'expression $(1+2)*3$ retournera la valeur 9.

La liste complète des opérateurs du C++

La liste ci-dessous présente les différents opérateurs du C++ avec leur associativité dans l'ordre de leur priorité (du premier évalué au dernier). Les opérateurs situés dans le même bloc ont la même priorité. Les opérateurs en **rouge** ne peuvent être surchargés.

Opérateurs	Description	Associativité	
::	Sélection d'un membre d'un espace de nom, ou d'un membre statique d'une classe	de gauche à droite	
()	Parenthèses pour évaluer en priorité		
[]	Tableau		
. ->	Sélection d'un membre par un identificateur (structures et objets) Sélection d'un membre par un pointeur (structures et objets)		
++ -- + - ! ~ (type) * & sizeof new delete delete[]	Incrémentation post ou pré-fixée Opérateur moins unaire Non logique et NON logique bit à bit cast Déréférencement Référencement (adresse d'une variable) Taille d'une variable / d'un type Allocation mémoire Libération mémoire Libération mémoire d'un tableau	de droite à gauche	
.* ->*	Déréférencement d'un pointeur de membre d'un objet Déréférencement d'un pointeur de membre d'un objet pointé	de gauche à droite	
* / %	Multiplication, division, et modulo (reste d'une division)		
+ -	Addition et soustraction		
<< >>	Décalage de bits vers la droite ou vers la gauche		
< <= > >=	Comparaison " inférieur strictement " et " inférieur ou égal " Comparaison " supérieur strictement " et " supérieur ou égal "		
== !=	Condition " égal " et " différent "		
&	ET logique bit à bit		
^	OU exclusif bit à bit		
	OU logique bit à bit		
&&	ET logique booléen		
	OU logique booléen		
c?t:f	Opérateur ternaire de condition		
= += -= *= /= %= <<= >>= &= ^= =	Affectation Affectation avec somme ou soustraction Affectation avec multiplication, division ou modulo Affectation avec décalage de bits Affectation avec ET logique, OU logique ou OU exclusif bit à bit		de droite à gauche
,	Séquence d'expressions		de gauche à droite

Les entrées-sorties

Classes de gestion des flux

Les entrées et sorties sont gérées par deux classes définies dans le fichier d'en-tête `<iostream>` :

- `ostream` (Output stream) permet d'écrire des données vers la console, un fichier, ... Cette classe surdéfinit l'opérateur `<<`.
- `istream` (Input stream) permet de lire des données à partir de la console, d'un fichier, ... Cette classe surdéfinit l'opérateur `>>`.

Flux standards

Trois instances de ces classes représentent les flux standards :

- `cout` écrit vers la sortie standard,
- `cerr` écrit vers la sortie d'erreur,
- `clog` écrit vers la sortie technique,
- `cin` lit à partir de l'entrée standard (jusqu'au premier espace exclu, éventuellement). Demander un nombre et y entrer des lettres provoque une erreur.
- `getline` lit à partir de l'entrée standard (tout).

Ces objets sont définis dans l'espace de nom `std`.

Exemple

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "Entrez un nombre positif : ";
    cin >> n;
    if (n<0) cerr << "Erreur: Le nombre " << n
                << " n'est pas positif " << endl;
    else cout << "Vous avez entré " << n << endl;
    return 0;
}
```

Autres types de flux

Les instances des classes dérivées des classes `istream` et `ostream` sont également manipulés avec les opérateurs `<<` et `>>`. Cependant, il ne faut pas oublier de les fermer en appelant la méthode `close()`.

Note: Les noms de fichiers sont codés sur 8 bits sous Linux/Unix et sur 16 bits sur Windows, ce qui peut induire des problèmes de portabilité, le cas échéant.

Flux de fichier

La classe `ifstream` permet de lire à partir d'un fichier. Le constructeur a la syntaxe suivante :

```
ifstream(const char* filename, openmode mode=in)
```

Le paramètre `mode` peut être une combinaison des valeurs suivantes :

app

(**append**) Placer le curseur à la fin du fichier avant écriture.

- ate** (**at end**) Placer le curseur à la fin du fichier.
- binary** Ouvrir en mode binaire plutôt que texte.
- in** Autoriser la lecture.
- out** Autoriser l'écriture.
- trunc** (**truncate**) Tronquer le fichier à une taille nulle.

Exemple 1 : lire un entier depuis un fichier

```
ifstream fichier("test.txt");
int a;
fichier >> a; // lire un entier
cout << "A = " << a;
fichier.close();
```

Exemple 2 : afficher tous les caractères d'un fichier

```
ifstream fichier("test.txt");
while (fichier.good())
    cout << (char) fichier.get();
fichier.close();
```

La classe `ofstream` permet d'écrire vers un fichier. Son constructeur a une syntaxe similaire :

```
ofstream(const char* filename, openmode mode=out|trunc)
```

Exemple :

```
ofstream fichier("test.txt");
fichier << setw(10) << a << endl;
fichier.close();
```

La classe `fstream` dérive de la classe `iostream` permettant à la fois la lecture et l'écriture. Cette dernière (`iostream`) dérive donc à la fois de la classe `ostream` et de la classe `istream`. Son constructeur a la syntaxe suivante :

```
fstream(const char* filename, openmode mode=in|out)
```

Exemple :

```
fstream fichier("test.txt");
fichier << setw(10) << a << endl;
fichier.seekg(0, ios_base::beg);
fichier >> b;
fichier.close();
```

Flux de chaîne de caractères

Ces flux permettent d'écrire pour produire une chaîne de caractères, ou de lire à partir d'une chaîne de caractères.

La classe `istringstream` dérivée de `istream` permet de lire à partir d'une chaîne de caractères, et possède deux constructeurs :

```
istringstream ( openmode mode = in );
istringstream ( const string & str, openmode mode = in );
```


Exemple :

```
int n, val;
string stringvalues;
stringvalues = "125 320 512 750 333";
istringstream iss (stringvalues, istringstream::in);

for (n = 0; n < 5; n++)
{
    iss >> val;
    cout << val << endl;
}
```

La classe `ostringstream` dérivée de `ostream` permet d'écrire pour créer une chaîne de caractères, et possède également deux constructeurs :

```
ostringstream ( openmode mode = out );
ostringstream ( const string & str, openmode mode = out );
```

Le second permet de spécifier le début de la chaîne de caractères produite.

La méthode `str()` retourne la chaîne de caractères produite.

Exemple :

```
ostringstream oss (ostringstream::out);
int a = 100;
oss << "Test d'écriture a=" << a << "\n";
cout << oss.str();
```

La classe `stringstream` dérivée de `iostream` permet d'écrire et lire, et possède deux constructeurs :

```
stringstream ( openmode mode = in | out );
stringstream ( const string & str, openmode mode = in | out );
```

Exemple :

```
int n, val;
stringstream ss (stringstream::in | stringstream::out);

// écriture
ss << "120 42 377 6 5 2000";

// lecture
for (int n = 0; n < 6; n++)
{
    ss >> val;
    cout << val << endl;
}
```

Manipulateurs

Le fichier d'en-tête `<iomanip>` définit des manipulateurs de flux tels que `endl`, `hex`. Ces manipulateurs modifient la façon d'écrire ou lire les données qui suivent celui-ci.

Manipulateur `endl`

Ce manipulateur écrit un retour à la ligne dans le flux, quel qu'il soit (`\r\n` pour Windows, `\n` pour Unix/Linux, `\r` pour Mac, ...). Il est donc conseillé de l'utiliser au lieu du/des caractère(s) correspondant(s), si la portabilité de votre application joue un rôle important.

Exemple:

```
cout << "Une première ligne" << endl << "Une deuxième ligne" << endl;
```

N.B.: Certains compilateurs C++ (notamment Visual C++) ne supporte pas que le manipulateur `endl` soit suivi d'autres données à écrire. Dans ce cas, il faut écrire les données suivantes dans une nouvelle instruction :

```
cout << "Une première ligne" << endl;
cout << "Une deuxième ligne" << endl;
```

Manipulateur hex

Ce manipulateur indique que les prochains entiers sont à lire ou écrire en base hexadécimale.

Manipulateur dec

Ce manipulateur indique que les prochains entiers sont à lire ou écrire en base décimale.

Manipulateur setbase(*base*)

Les 2 manipulateurs précédents sont des alias de celui-ci, qui permet de spécifier la base des prochains entiers à lire ou écrire.

Exemple :

```
int a = 200; // 200 en décimal
cout << "Valeur de a en base 16 = " << setbase(16) << a << endl;
// affiche: Valeur de a en base 16 = C8

cout << "Valeur de a en base 10 = " << setbase(10) << a << endl;
// affiche: Valeur de a en base 10 = 200

cout << "Valeur de a en base 8 = " << setbase(8) << a << endl;
// affiche: Valeur de a en base 8 = 310
```

Manipulateur setw(*width*)

Ce manipulateur indique que les prochaines données doivent être écrites sur le nombre de caractères indiqué, en ajoutant des caractères espaces avant.

Exemple :

```
int a = 11;
cout << "Valeur de a = " << setw(5) << a << endl;
```

Ce code affiche :

```
Valeur de a =    11
```

Manipulateur setfill(*char*)

Ce manipulateur modifie le caractère utilisé pour compléter les données utilisant le manipulateur `setw`.

Exemple :

```
int a = 11;
cout << "Valeur de a = " << setfill('x') << setw(5) << a << endl;
```

Ce code affiche :

```
Valeur de a = xxx11
```

Manipulateur `setprecision(digits)`

Ce manipulateur spécifie que les prochains nombres à virgule flottante doivent être écrits avec la précision donnée. La précision donne le nombre maximum de chiffres à écrire (avant et après la virgule).

Exemple :

```
double f = 3.14159;
cout << setprecision (5) << f << endl;
cout << setprecision (9) << f << endl;
```

Ce code affiche :

```
3.1416
3.14159
```

Manipulateurs `setiosflags` et `resetiosflags`

Le manipulateur `setiosflags` (resp. `resetiosflags`) active (resp. désactive) des options de format des données.

Ces deux manipulateurs possèdent un argument dont le type est défini par l'énumération `ios_base::fmtflags`. Cet argument peut être :

`ios_base::boolalpha`

Ecrire/lire les données de type `bool` sous forme textuelle, càd `true` ou `false`.

`ios_base::oct`

Ecrire/lire les entiers en base octale (base 8).

`ios_base::dec`

Ecrire/lire les entiers en base décimale (base 10).

`ios_base::hex`

Ecrire/lire les entiers en base hexadécimale (base 16).

`ios_base::showbase`

Faire précéder les entiers par leur base.

`ios_base::showpos`

Faire précéder les nombres positifs du signe plus (+).

`ios_base::showpoint`

Toujours écrire la virgule des nombres réels.

`ios_base::fixed`

Ecrire les nombres réels avec une virgule fixe.

`ios_base::scientific`

Ecrire les nombres réels sous forme scientifique.

`ios_base::left`

Aligner les donnés à gauche (setw).

`ios_base::right`

Aligner les donnés à droite (setw).

`ios_base::internal`

Aligner les donnés en remplissant à une position interne (setw).

`ios_base::skipws`

Ignorer les caractères blancs avant de lire les données.

`ios_base::unitbuf`

Vider le buffer de sortie à chaque écriture.

`ios_base::uppercase`

Ecrire les données en majuscules.



Cette section est vide, pas assez détaillée ou incomplète.

Les pointeurs

Introduction

Une variable correspond à un emplacement en mémoire (adresse) où se trouve une valeur. Toute variable *a* permet d'accéder :

- à sa valeur en lecture et en écriture :

```
int a;
a = 10; // écriture de la valeur de a
cout << "A vaut " << a ; // lecture de la valeur de a
```

- à son adresse en lecture seulement car l'adresse (l'emplacement mémoire) est choisie par le système :

```
cout << "L'adresse de A est " << &a ; // lecture de l'adresse de a
```

Un pointeur désigne un type particulier de variable dont la valeur est une adresse. Un pointeur permet donc de contourner la restriction sur le choix de l'adresse d'une variable, et permet essentiellement d'utiliser la mémoire allouée dynamiquement.

Il est utilisé lorsque l'on veut manipuler les données stockées à cette adresse. C'est donc un moyen indirect de construire et de manipuler des données souvent très complexes.

Déclaration

```
type* identificateur;
```

La variable *identificateur* est un pointeur vers une valeur de type *type*.

L'opérateur &

C'est l'opérateur d'*indirection*. Il permet d'obtenir l'adresse d'une variable, c'est-à-dire un pointeur vers cette variable.

```
&identificateur // permet d'obtenir l'adresse mémoire de la variable identificateur
```

Il renvoie en réalité une adresse mémoire, l'adresse où est stockée physiquement la variable *identificateur*.

L'opérateur *

```
*variable
```

C'est l'opérateur de *déréférencement*. Il permet d'obtenir et donc de manipuler les données pointées par la variable *variable*. Ainsi **pointeur* permet d'accéder à la valeur pointée par *pointeur* en lecture et en écriture.

Comparaison avec une variable classique

```
int a;
int* pA;
pA = &a; // l'adresse de a est stockée dans pA
```

écriture de la valeur de a `a = 10;`

`*pA = 10;`

lecture de la valeur de a `cout << "A vaut " << a ;` `cout << "A vaut " << *pA ;`

lecture de l'adresse de a `cout << "L'adresse de A est " << &a ;` `cout << "L'adresse de A est " << pA ;`

Le pointeur pA peut par la suite pointer l'adresse d'une autre variable, où bien pointer l'adresse d'un bloc de mémoire alloué dynamiquement.

Exemple de programme

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

int main()
{
    int a, b, c;
    int *x, *y;

    a = 98;
    x = &a;
    c = *x + 5;
    y = &b;
    *y = a + 10;

    cout << "La variable b vaut : " << b << endl;
    cout << "La variable c vaut : " << c << endl;

    return 0;
}
```

Exécution

```
La variable b vaut 108
La variable c vaut 103
```

Explications

- Dans ce programme, on déclare 3 variables a, b et c. On déclare ensuite 2 pointeurs vers des entiers x et y.
- a est initialisé à 98.
- x=&a; permet de mettre dans x l'adresse de a. x est désormais un pointeur vers a.
- *x est la variable pointée par x, c'est-à-dire a, qui vaut donc 98 après évaluation.
 - c=*x+5; permet donc de transférer 98+5 donc 103 dans la variable c.
- y=&b; permet de mettre dans la variable y l'adresse de la variable b. y est désormais un pointeur vers b.
 - a+10 vaut 98+10 donc 108.
- *y=a+10; permet de transférer dans la variable pointée par y la valeur de a+10, c'est-à-dire 108. On stocke donc 108 dans b, de manière indirecte via le pointeur y.
- on affiche ensuite les valeurs de b et c c'est-à-dire respectivement 108 et 103.

Opérations arithmétiques sur les pointeurs

hormis l'opérateur de déréférencement, les pointeurs peuvent être utilisés avec l'opérateur d'addition (+). L'addition d'un pointeur avec une valeur entière permet d'avancer ou reculer le pointeur du nombre d'éléments indiqué.

Exemple 1

```

char* ptr="Pointeur"; // ptr pointe le premier caractère de la chaîne de caractères
cout << ptr << endl; // affiche "Pointeur"

ptr = ptr+3;
cout << ptr << endl; // affiche "nteur"

cout << ++ptr << endl; // affiche "teur"
cout << --ptr << endl; // affiche "nteur"

```

Comme l'exemple précédent le montre, il est également possible d'utiliser les opérateurs d'incrémement et de décrémement.

Exemple 2

```

int premiers[] = { 2, 3, 5, 7, 11, 13, 17 };
int* ptr = premiers; // pointe le premier élément du tableau

cout << hex << ptr << endl; // affiche l'adresse pointée (par exemple 01C23004)
cout << *ptr << endl; // affiche "2"
cout << *(ptr+5) << endl; // affiche "13"

ptr=&premiers[3]; // pointe le 4e élément (index 3)
cout << hex << ptr << endl; // affiche l'adresse pointée (par exemple 01C23018)
cout << *ptr << endl; // affiche "7"
cout << *(ptr-1) << endl; // affiche "5"

```

Dans l'exemple 2, la différence d'adresse est de 20 octets ($5 * \text{sizeof}(\text{int})$). Cela montre que l'adresse contenue dans le pointeur est toujours incrémentée ou décrémentée d'un multiple de la taille d'un élément ($\text{sizeof} * \text{ptr}$).

Opération utilisant deux pointeurs

La seule opération valable (ayant un sens) utilisant deux pointeurs de même type est la soustraction (-) donnant le nombre d'éléments entre les deux adresses. Elle n'a de sens que si les deux pointeurs pointent dans le même tableau d'éléments.

Exemple :

```

char str[]="Message où rechercher des caractères.";
char *p1 = strchr(str,'a'); // recherche le caractère 'a' dans str
char *p2 = strchr(str,'è'); // recherche le caractère 'è' dans str

cout << "Nombre de caractères de 'a' à 'è' = " << (p2-p1) << endl;
// affiche : Nombre de caractères de 'a' à 'è' = 28

```

Pointeur constant et pointeur vers valeur constante

Il ne faut pas confondre un pointeur constant (qui ne peut pointer ailleurs) avec un pointeur vers une valeur constante (l'adresse contenue dans le pointeur peut être modifiée mais pas la valeur pointée).

Dans les 2 cas le mot clé `const` est utilisé, mais à 2 endroits différents dans le type de la variable.

Pointeur vers une valeur constante

Le mot-clé `const` placé avant le type du pointeur permet d'empêcher la modification de la valeur pointée.

Exemple:

```

const char* msg = "essai constant";

*msg = 'E'; // <- Interdit, la valeur pointée ne peut être modifiée

```

```
msg = "Test"; // OK -> le pointeur contient l'adresse de "Test"
```

Pointeur constant

Le mot-clé `const` placé entre le type du pointeur et la variable permet d'empêcher la modification du pointeur lui-même (l'adresse).

Exemple:

```
char* const msg = "essai constant";
msg = "Test"; // <- Interdit (erreur de compilation),
              // ne peut pointer la chaîne "Test"
*msg = 'E'; // OK -> "Essai constant"
```

Pointeur constant vers une valeur constante

Le mot-clé `const` apparaissant aux 2 endroits empêche à la fois la modification du pointeur lui-même et celle de la valeur pointée.

Exemple:

```
const char* const msg = "essai constant";
msg = "Test"; // <- Interdit (erreur de compilation),
              // ne peut pointer la chaîne "Test"
*msg = 'E'; // <- Interdit, la valeur pointée ne peut être modifiée
```

Position du mot clé const

Une méthode simple a été proposée par Dan Sacks pour déterminer si c'est la valeur pointée ou le pointeur lui-même qui est constant. Elle se résume en une seule phrase mais n'a jamais été intégrée dans les compilateurs :

```
"const s'applique toujours à ce qui le précède".
```

Par conséquent, une déclaration ne commencera jamais par `const` qui ne serait précédé de rien et qui ne pourrait donc s'appliquer à rien. En effet, les deux déclarations suivantes sont strictement équivalentes en C++ :

```
const char * msg; // déclaration habituelle
char const * msg; // déclaration Sacks
```

Cette méthode peut s'avérer précieuse dans le cas de déclarations plus complexes :

```
int const ** const *ptr; // ptr est un pointeur vers un pointeur constant de pointeur d'entier constant :
                        // *ptr et ***ptr ne peuvent être modifiés
```

Voir aussi

- [Exercices](#)

Les références

Présentation des références

Une référence peut être vue comme un alias d'une variable. C'est-à-dire qu'utiliser la variable, ou une référence à cette variable est équivalent. Ce qui signifie que l'on peut modifier le contenu de la variable en utilisant une référence.

Une référence ne peut être initialisée qu'une seule fois : à la déclaration. Toute autre affectation modifie en fait la variable référencée. Une référence ne peut donc référencer qu'une seule variable tout au long de sa durée de vie.

Déclaration

```
type& identificateur=variable;  
ou  
type& identificateur(variable);
```

Sémantique

La variable *identificateur* est une référence vers la variable *variable*. La variable *variable* doit être de type *type*.

Exemple de programme

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int a = 98,  
        b = 78,  
        c;  
  
    int &x = a;  
    c = x + 5; // équivaut à : c = a + 5;  
  
    int &y = b;  
    y = a + 10; // équivaut à : b = a + 10;  
  
    cout << "La variable b vaut : " << b << endl;  
    cout << "La variable c vaut : " << c << endl;  
  
    return 0;  
}
```

Exécution

```
La variable b vaut : 108  
La variable c vaut : 103
```

Explications

- Dans ce programme, on définit 3 variables entières a, b et c et on initialise a à 98 et b à 78.
- `int &x=a;` permet de déclarer une référence x vers la variable a. `x+5` vaut donc la même chose que `a+5` donc 103.
- `c=x+5;` permet donc de transférer 103 dans la variable c.
- `int &y=b;` permet de déclarer une référence y vers la variable b. `a+10` vaut 98+10 donc 108.
- `y=a+10;` permet de transférer 108 dans la variable b.
- on affiche ensuite b et c c'est-à-dire respectivement 108 et 103.

Pourquoi utiliser une référence ?

C'est la question qui peut se poser en regardant l'exemple précédent, où il serait plus clair d'utiliser directement les variables.

Les références sont principalement utilisées pour passer des paramètres aux fonctions. Voir le [chapitre sur les fonctions](#), section « [passage de paramètres par référence](#) ».

Les références constantes sont également utilisées pour référencer des résultats de retour de fonctions afin d'éviter les copies. C'est particulièrement indiqué dans le cas d'objets retournés par des fonctions. Dans ce cas, la valeur ou objet temporaire retourné a une durée de vie aussi longue que la référence.

Exemple :

```

class Retour
{
public:
    void g() const {}
};

Retour f() { return Retour(); }

int main(int argc, char *argv[])
{
    const Retour &retour = f();
    retour.g();
    return 0;
}
    
```

Les références et leur lien avec les pointeurs

Une référence est un pointeur que l'on ne peut pas réaffecter (car le compilateur l'interdit), qui se déréférence automatiquement (à l'inverse d'un pointeur pour lequel on doit utiliser l'opérateur d'indirection), et dont à l'inverse d'un pointeur on ne peut connaître l'adresse car le compilateur ne le permet pas. En effet, si *v* est une référence alors *&v* donnera l'adresse de l'objet référencé par *v*, et non l'adresse de la case mémoire où est stockée la référence.

Exercices

Exercice 1

Faites une fonction dont la déclaration sera `void échanger(int & a, int & b)` qui devra échanger les deux valeurs.

Solution

```

void échanger(int & a, int & b)
{
    int c = a;
    a = b;
    b = c;
}
    
```

Exercice 2

Faites une fonction pour calculer la factorielle d'un nombre. Sa déclaration sera `int fact(int & n)`. La fonction sera récursive et la valeur de retour sera *n*. En cas de problèmes, consulter l'Aide 1.

Aide 1

La factorielle (notée "!") est une fonction mathématique.

Voici quelques exemple :

4! = 4 x 3 x 2 x 1 = 24

3! = 3 x 2 x 1 = 6

$2! = 2 \times 1 = 2$
 $1! = 1$
 Notez que :
 $4! = 4 \times 3!$
 $3! = 3 \times 2!$
 $2! = 2 \times 1! = 2 \times 1$
 D'où :
 $n! = n \times (n-1)!$ si $n > 1$

Solution

Voici un exemple de fonction récursive qui ne répond pas à la consigne d'avoir une déclaration `int factorielle (int & n)` et qui par conséquent ne peut être qualifiée de solution à l'exercice 2 :

```

#include <iostream>
using namespace std;
int factorielle(int n)
{
    if(n == 1) return 1;
    return n * factorielle(n-1);
}
int main(void)
{
    int y = factorielle(2);
    cout << "resultat : " << y << endl;
}
    
```

La vraie solution est :

```

#include <iostream>
using namespace std;
int factorielle(int& n)
{
    if(n == 1) return 1;
    n--;
    return (n+1) * factorielle(n);
}
int main(void)
{
    int n = 2;
    int y = factorielle(n);
    cout << "resultat : " << y << endl;
}
    
```

Tests

Test 1

Indiquez si la syntaxe est correcte ou non.

Cas 1

```

int b = n;
int & ref = b;
    
```

Cas 2

```

int x=5;
int & var = x;
    
```

Cas 3

```
int n = 2;
int & ref = n;
if(*(ref) == 2) ref++; //ceci provoque une erreur car ref n'est pas un pointeur
```

Cas 4

```
#include <iostream>
using namespace std;
void afficher_par_reference(int & a)
{
cout << a << endl;
}
```

Cas 5

```
int b = 2;
int ref& = b;
```

Solution

Solutions

1. vrai
2. vrai
3. faux
4. vrai
5. faux

Test 2

Dans ces exemples, trouvez ce que le programme va afficher.

Cas 1

```
#include <iostream>
using namespace std;
int main()
{
int b = 2;
int a = 4;
int & ref1 = b;
int & ref2 = a;
ref2 += ref1;
ref1 -= ref2;
cout << ref2 << " " << ref1 << endl;
}
```

Solution

6 -4

Les tableaux

Les tableaux à une dimension

Les tableaux sont des structures de données constituées d'un certain nombre d'éléments de même type. On peut accéder directement à un élément du tableau en indiquant son indice entre crochets (indice de 0 à *nombre_d_éléments*-1).

Les tableaux statiques

Syntaxe

```
type identificateur[taille];
```

Sémantique

identificateur est un tableau de *taille* éléments de type *type*. *taille* est obligatoirement une valeur constante. La taille du tableau est donc figée une fois pour toute et ne peut pas être modifiée en cours d'exécution.

Pour désigner la *i*-ième case de tableau, on écrit

```
identificateur[i]
```

où *i* est un entier quelconque. Les cases sont numérotées à partir de 0 : les valeurs possibles de *i* vont donc de 0 à *taille*-1.

Pointeurs et tableaux

Il existe un lien entre les pointeurs et les tableaux : *identificateur* est en fait un pointeur constant vers un élément de type *type*, pointant le premier élément du tableau, en d'autres termes le nom d'un tableau est un pointeur constant sur le premier élément du tableau.

Exemple

```
#include <iostream>

using namespace std;

int main()
{
    int i;

    int const tailleTableau(10); //taille du tableau
    int tableau[tailleTableau]; //déclaration du tableau

    for (int i(0); i<tailleTableau; i++ )
    {
        tableau[i]=i*i;
        cout<<"Le tableau ["<<i<<" contient la valeur "<<tableau[i]<<endl;
    }

    return 0;
}
```

Exécution

Le tableau [0] contient la valeur 0

Le tableau [1] contient la valeur 1

Le tableau [2] contient la valeur 4

Le tableau [3] contient la valeur 9

Le tableau [4] contient la valeur 16

Le tableau [5] contient la valeur 25

Le tableau [6] contient la valeur 36

Le tableau [7] contient la valeur 49

Le tableau [8] contient la valeur 64

Le tableau [9] contient la valeur 81

Les tableaux dynamiques

Un tableau dynamique est un tableau dont le nombre de cases peut varier au cours de l'exécution du programme. Il permet d'ajuster la taille du tableau au besoin du programmeur.

L'opérateur new

Syntaxe :

```
pointeur=new type[taille];
```

L'opérateur `new` permet d'allouer une zone mémoire pouvant stocker *taille* éléments de type *type*, et retourne l'adresse de cette zone. Le paramètre *taille* est un entier qui peut être quelconque (variable, constante, expression). `new` renverra un pointeur vers un *type*. La variable *pointeur* est donc du type *type* *. Les cases du tableaux seront numérotées de 0 à *taille*-1 et on y accédera comme un tableau statique. S'il n'y a pas assez de mémoire disponible, `new` renvoie le pointeur `NULL`.

L'opérateur delete[]

Syntaxe :

```
delete[] pointeur;
```

Cette utilisation de `delete[]` permet de détruire un tableau précédemment alloué grâce à `new`. N'oubliez surtout pas les crochets juste après `delete`, sinon le tableau ne sera pas correctement libéré. Le programmeur en C++ doit gérer la destruction effective des tableaux qu'il a créé dynamiquement.

Exemple

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

int main()
{
    int i, taille;

    cout << "Tapez la valeur de taille : ";
    cin >> taille;
    int *t;
    t = new int[taille];

    for (i = 0; i < taille; i++)
        t[i] = i * i;

    for (i = 0; i < taille; i++)
        cout << t[i] << endl;
    delete[] t;

    return 0;
}
```

Exécution 1

```

Tapez la valeur de taille : 4
;0
;1
;4
;9
    
```

Exécution 2

```

Tapez la valeur de taille : 6
;0
;1
;4
;9
;16
;25
    
```

Tableaux multidimensionnels

Un tableau peut avoir plus d'une dimension (matrice 2D, matrice 3D, ...). Par contre, on ne peut pas utiliser l'expression suivante pour l'allocation dynamique de tableaux à plusieurs dimensions:

```

pointeur=new type[taille1][taille2]...;
    
```

L'allocation dynamique de tableaux à deux dimensions est expliquée ci-dessous.

Les vectors

Les *vectors* (vecteur en français) sont un type de tableaux dynamiques très puissants qui suivent la syntaxe suivante :

```

vector<type> nom(taille);
    
```

Par exemple, un tableau dynamique constitué de 5 entiers et nommé « tableau_nombre_entiers » sera défini de la sorte :

```

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

Mais on peut bien sûr définir un tableau qui ne comporte aucune taille, ce qui est bien utile pour un tableau... dynamique !

```

vector<int> tableau_nombres;
    
```

Pour accéder aux valeurs d'un *vector*, on procède la même manière que pour tous les tableaux, par exemple :

```

tableau_nombres_entiers[0] = 0;
tableau_nombres_entiers[1] = 2;
...
    
```

Mais on peut également affecter à tout le *vector* une même valeur en l'indiquant juste après la taille avec une virgule. Exemple :

```

#include <iostream>
#include <string> //ATTENTION : PENSER À INCLURE "STRING"
using namespace std;
    
```

```
vector<int> tableau_nombre_entiers(5, 10); //Ce tableau est composé de 5 nombres entiers qui sont tous égaux à 10
vector<string> tableau_lettres(4, "bonjour !"); //Ce tableau est composé de 4 chaînes de caractères qui sont égales à « bonjour ! »
```

Cela signifie que `tableau_nombre_entiers[0]=tableau_nombre_entiers[1]=tableau_nombre_entiers[2]=tableau_nombre_entiers[3]=tableau_nombre_entiers[4]=10` ; et de même `tableau_lettres[0]=tableau_lettres[1]=tableau_lettres[2]=tableau_lettres[3]="bonjour !"`.

Comme on le sait, les *vectors* sont des tableaux dynamiques, ainsi peuvent-ils s'agrandir ou se rétrécir. Pour ajouter une case supplémentaire au tableau, il suffit de faire appel à la fonction `push_back()` :

```
vector<int> tableau_entiers;
tableau_entiers.push_back(7); //On ajoute une première case au tableau qui comporte le nombre 7
tableau_entiers.push_back(18); //On ajoute une deuxième case au tableau qui comporte le nombre 18
...
```

À l'inverse, pour supprimer une case, on utilise la fonction `pop_back()` :

```
vector<int> tableau_entiers(10); //Un tableau dynamique d'entiers à 10 cases
tableau_entiers.pop_back(); //Plus que 9 cases
tableau_entiers.pop_back(); //Plus que 8 cases
...
```

Enfin, pour récupérer la taille du *vector*, on utilise la fonction `size()` :

```
vector<int> tableau_entiers(5); //Un tableau de 5 entiers
int taille = tableau_entiers.size(); // 'taille' a pour valeur 5
```

Parcours d'un tableau

Le parcours des éléments d'un tableau se fait généralement avec une boucle `for` (on peut aussi utiliser `while` ou `do...while`). Exemple :

```
int nombres[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
for (int i = 0; i < 10; i++) // i va de 0 à 9 inclus
    cout << "nombres [ " << i << " ] = " << nombres[i] << endl;
```

Les éléments d'un tableau sont stockés dans des zones contiguës. Il est donc possible d'utiliser un pointeur pour parcourir un tableau :

```
int nombres[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
int *p = nombres;
for (int i = 0; i < 10; i++, p++)
    cout << "Nombre : " << *p << endl;
```

Initialement le pointeur `p` pointe le premier élément du tableau. L'instruction `p++` dans la boucle `for` incrémente le pointeur, c'est-à-dire qu'il passe à l'élément suivant. L'adresse contenue dans le pointeur n'augmente pas de 1, mais de la taille de l'élément pointé (ici `int`, soit 4 octets en général).

Les tableaux à deux dimensions

Statiques

■ Syntaxe

`type identificateur[taille_i][taille_j];`

■ Sémantique :

`identificateur` est un tableau de `taille_i` sur `taille_j` cases. Pour chaque case `i` choisie, il y a `taille_j` cases disponibles. Les `taille_i` et `taille_j` ne peuvent changer au cours du programme. Pour le faire, il faut utiliser les tableaux dynamiques.

Exemple

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

int main() {
    int t[10][5];

    for (int i = 0; i < 10; i++)
        for (int j = 0; j < 5; j++)
            t[i][j] = i * j;

    for (int i = 0; i < 10; ++i) {
        for (int j = 0; j < 5; ++j){
            cout << t[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Exécution

```
0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12
0 4 8 12 16
0 5 10 15 20
0 6 12 18 24
0 7 14 21 28
0 8 16 24 32
0 9 18 27 36
```

Dynamiques

Exemple

```
#include <iostream>
using std::cout;
using std::cin;

int **t;
int nColonnes;
int nLignes;

void Free_Tab(); // Libérer l'espace mémoire;

int main(){

    cout << "Nombre de colonnes : "; cin >> nColonnes;
    cout << "Nombre de lignes : "; cin >> nLignes;

    /* Allocation dynamique */
    t = new int* [ nLignes ];
    for (int i=0; i < nLignes; i++)
        t[i] = new int[ nColonnes ];

    /* Initialisation */
    for (int i=0; i < nLignes; i++)
        for (int j=0; j < nColonnes; j++)
            t[i][j] = i*j;

    /* Affichage */
    for (int i=0; i < nLignes; i++) {
```

```

    for (int j=0; j < nColonnes; j++)
        cout << t[i][j] << " ";
    cout << endl;
}

Free_Tab();
system("pause>nul");
return 0;
}

void Free_Tab(){
    for (int i=0; i < nLignes; i++)
        delete[] t[i];
    delete[] t;
}

```

Exécution 1

```

Nombre de colonnes : 6
Nombre de lignes : 3
0 0 0 0 0
0 1 2 3 4 5
0 2 4 6 8 10

```

Exécution 2

```

Tapez la valeur de taille_i : 8
Tapez la valeur de taille_j : 2
0
0 1
0 2
0 3
0 4
0 5
0 6
0 7

```

Les tableaux de caractères

Un tableau de caractères constitue une chaîne de caractères. Exemple avec des caractères de huit bits:

```
char chaine[20] = "BONJOUR";
```

Les tableaux de char

Un tableau de `char` constitue une chaîne de caractères. Exemple :

```
char chaine[20] = "BONJOUR";
```

La variable `chaine` peut stocker jusqu'à 20 caractères, et contient les caractères suivants :

```
'B', 'O', 'N', 'J', 'O', 'U', 'R', '\0'
```

Le dernier caractère est nul pour indiquer la fin de la chaîne de caractères. Il est donc important de prévoir son stockage dans le tableau. Dans cet exemple, 8 caractères sont donc nécessaires.

L'exemple précédent équivaut à :

```
char chaîne[20] = {'B','O','N','J','O','U','R','\0'};
```

Si la chaîne est initialisée à la déclaration, on peut également laisser le compilateur détecter le nombre de caractères nécessaires, en omettant la dimension :

```
char chaîne[] = "BONJOUR"; // tableau de 8 caractères
```

Dans ce cas la variable ne pourra stocker que des chaînes de taille inférieure ou égale à celle d'initialisation.

Les fonctions de manipulation de chaîne de caractères sont les mêmes que dans le langage C. Elles portent un nom commençant par `str` (**String**) :

int strlen(const char* source)

Retourne la longueur de la chaîne `source` (sans le caractère nul final).

void strcpy(char* dest, const char* source)

Copie la chaîne `source` dans le tableau pointé par `dest`.

Les tableaux de `char16_t` and `char32_t`

À partir de C++11 (C++ norme de 2011) trois types de chaînes de caractères sont prise en charge: [UTF-8](#), [UTF-16](#), et [UTF-32](#). Le type `char` conserve ses unités de codage de huit bits pour le codage des caractères Unicode via UTF-8, les nouveaux types `char16_t` et `char32_t` sont des unités de codage de seize ou trente-deux bits pour le codage des caractères Unicode via [UTF-16](#) ou [UTF-32](#).

Ces types sont standard à partir de C++2011 mais n'existent pas sur des compilateurs plus anciens, ni même sur les compilateur C-2011.

Il est également possible de les initialiser avec des littéraux, selon le cas:

```
u8"I'm a UTF-8 string."
u"This is a UTF-16 string."
U"This is a UTF-32 string."
```

Les tableaux de `wchar_t`

Les compilateurs moderne permettent de faire des chaînes de `wchar_t` (16 bits ou 32 bits) au lieu de `char` (8 bits). Ceci a été fait pour permettre de représenter une plus grande partie des caractères Unicode, même si on peut représenter l'ensemble de ces caractères en [UTF-8](#).

Dans ce cas, il faut précéder les caractères par `L` :

```
wchar_t chaîne[20] = L"BONJOUR";
```

Équivaut à :

```
wchar_t chaîne[20] = {L'B',L'O',L'N',L'J',L'O',L'U',L'R',L'\0'};
```

Les fonctions permettant leur manipulation portent un nom similaire, excepté que `str` doit être remplacé par `wcs` (**Wide Character String**) :

- `int wcslen(const wchar_t* source)`
- `wcsncpy(wchar_t* dest, const wchar_t* source)`

Portabilité

Certains environnements de développement (Visual C++ en particulier) considèrent Unicode comme une option de compilation. Cette option de compilation définit alors le type `TCHAR` comme `char` (option legacy) ou `wchar_t` (option unicode), selon l'option Unicode. Les constantes sont alors à encadrer par la macro `_T(x)` qui ajoutera `L` si nécessaire :

```
_TCHAR chaîne1[20] = _T("BONJOUR");
```

```
TCHAR chaine2[20] =  
{_T('B'),_T('O'),_T('N'),_T('J'),_T('O'),_T('U'),_T('R'),_T('\0')};
```

Le nom des fonctions de manipulation commencent par `_tcs` :

- `int _tcslen(const TCHAR* source)`
- `_tcscopy(TCHAR* dest, const TCHAR* source)`

Voir aussi

- [Exercices](#)
- [Exercices sur les tableaux statiques](#)

Les structures de contrôles

Une série d'instructions dans une fonction s'exécute séquentiellement par défaut. Cependant, il est nécessaire que certaines parties du code ne s'exécutent que sous certaines conditions, ou s'exécutent plusieurs fois dans une boucle pour par exemple traiter tous les éléments d'un tableau.

Le if

Cette structure de contrôle permet d'exécuter une instruction ou une suite d'instructions seulement si une condition est vraie.

Syntaxe :

```
if (condition) instruction
```

condition

Expression booléenne de la condition d'exécution.

instruction

Une instruction ou un bloc d'instructions entre accolades exécuté si la condition est vraie.

Lorsqu'il n'y a qu'une ligne d'instructions conditionnelles, la paire d'accolades est conditionnelle, mais les omettre est dangereux.

Sémantique :

On évalue la condition :

- si elle est vraie on exécute l'instruction et on passe à l'instruction suivante.
- si elle est fausse on passe directement à l'instruction suivante.

L'instruction peut être remplacée par une suite d'instructions entre accolades.

Exemple :

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

int main()
{
    int a;
    cout << "Tapez la valeur de a : "; cin >> a;

    if (a > 10)
    {
        cout << "a est plus grand que 10" << endl;
    }
    cout << "Le programme est fini." << endl;
    return 0;
}
```

Ce programme demande à l'utilisateur de taper un entier a. Si a est strictement plus grand que 10, il affiche "a est plus grand que 10.". Dans le cas contraire, il n'affiche rien.

Exécution 1

```
Tapez la valeur de a : 12
a est plus grand que 10
```

Exécution 2

```
Tapez la valeur de a : 8
```

Condition multiple contenant des opérateurs logiques

Si la condition à évaluer est complexe et contient des opérateurs logiques, chaque condition doit être écrite entre parenthèse.

Par exemple, si la condition `condition1` et la condition `condition2` doivent être vérifiées en même temps, on écrira

```
if ( (condition1) && (condition2) ) instruction
```

où l'opérateur logique `&&` peut être remplacé par tout autre opérateur logique.

Ajouter des opérateurs logiques se fait selon le même principe, en prenant en compte les règles de préséance de ces opérateurs pour adapter la place des parenthèses.

Le if...else

Cette structure de contrôle permet d'exécuter une instruction si une condition est vraie, ou une autre instruction si elle est fausse.

Syntaxe

```
if (condition) instruction1
else instruction2
```

condition

Expression booléenne de la condition d'exécution.

instruction1

Une instruction ou un bloc d'instructions exécuté si la condition est vraie.

instruction2

Une instruction ou un bloc d'instructions exécuté si la condition est fausse.

Sémantique

On évalue la condition :

- si elle est vraie, on exécute l'*instruction1* et on passe à l'instruction suivante
- si elle est fausse, on exécute l'*instruction2* et on passe à l'instruction suivante

L'*instruction1* ou l'*instruction2* peuvent être remplacées par une suite d'instructions entre accolades. Lorsqu'il n'y a qu'une ligne d'instructions conditionnelles, la paire d'accolades est conditionnelle, mais les omettre est dangereux.

Exemple :

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

int main()
{
    int a;
    cout << "Tapez la valeur de a : "; cin >> a;

    if (a > 10)
    {
        cout << "a est plus grand que 10" << endl;
    }
    else
    {
        cout << "a est inférieur ou égal à 10" << endl;
    }
}
```

```

return 0;
}

```

Ce programme demande à l'utilisateur de taper un entier *a*. Si *a* est strictement plus grand que 10, il affiche "a est plus grand que 10". Dans le cas contraire, il affiche "a est inférieur ou égal à 10".

Exécution 1

```

Tapez la valeur de a : 12
a est plus grand que 10

```

Exécution 2

```

Tapez la valeur de a : 8
a est inférieur ou égal à 10

```

Plusieurs instructions par condition

Si plusieurs actions doivent s'enchaîner lorsqu'une condition est vraie, on met obligatoirement ces actions entre accolades. L'accolade fermante ne doit pas être suivie d'un point virgule dans ce cas. Ainsi, on écrira

```

if (condition1)
{
instruction1.1 ;
instruction1.2 ;
...
} // pas de point virgule ici

else if (condition2)
{
instruction2.1 ;
...
}
...

```

qui exécute les instructions *instruction1.1* et *instruction1.2*, ... si la condition *condition1* est vraie, et les instructions *instruction2.1*, ... si la condition *condition2* est vraie. Lorsqu'il n'y a qu'une ligne d'instructions conditionnelles, la paire d'accolades est conditionnelle, mais les omettre est dangereux.

Le switch

L'instruction `switch` permet de tester plusieurs valeurs pour une expression.

Syntaxe

```

switch(expression)
{
case constante1:
instruction1_1
instruction1_2...
case constante2:
instruction2_1
instruction2_2...
...
default:
instruction_1
instruction_2...
}

```

expression

Expression de type scalaire (entier, caractère, énumération, booléen).

case constante1: instruction1_...

Une série d'instructions ou de blocs d'instructions exécutés si *expression* vaut *constante1*.

case constante2: instruction2_...

Une série d'instructions ou de blocs d'instructions exécutés si *expression* vaut *constante2*.

default: instruction_...

Une série d'instructions ou de blocs d'instructions exécutés quand aucun des cas précédent ne correspond.

Sémantique

On teste la valeur de l'expression spécifiée. On la compare successivement aux constantes `constante1`, `constante2`, etc.

Si la valeur de l'expression correspond à l'une des constantes, l'exécution débute à l'instruction correspondante. Si aucune constante ne correspond, l'exécution débute à l'instruction correspondant à `default:`. L'exécution se termine à l'accolade fermante du `switch` ou avant si l'instruction `break;` est utilisée.

En général, l'instruction `break` sépare les différents cas. Il n'est pas utilisé quand plusieurs cas sont traités par les mêmes instructions.

En C++, l'expression n'est jamais une chaîne de caractères, ni un tableau, ni un objet, ni une structure !

Exemple 1

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

int main()
{
    int a;
    cout << "Tapez la valeur de a : "; cin >> a;

    switch(a)
    {
        case 1 :
            cout << "a vaut 1" << endl;
            break;

        case 2 :
            cout << "a vaut 2" << endl;
            break;

        case 3 :
            cout << "a vaut 3" << endl;
            break;

        default :
            cout << "a ne vaut ni 1, ni 2, ni 3" << endl;
            break;
    }
    return 0;
}
```

Ce programme demande à l'utilisateur de taper une valeur entière et la stocke dans la variable `a`. On teste ensuite la valeur de `a` : en fonction de cette valeur on affiche respectivement les messages "a vaut 1", "a vaut 2", "a vaut 3", ou "a ne vaut ni 1, ni 2, ni 3".

Exécution 1 :

```
Tapez la valeur de a : 1
a vaut 1
```

Exécution 2 :


```
Tapez la valeur de a : 2
a vaut 2
```

Exécution 3 :

```
Tapez la valeur de a : 3
a vaut 3
```

Exécution 4 :

```
Tapez la valeur de a : 11
a vaut ne vaut ni 1, ni 2, ni 3
```

Exemple 2

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

int main()
{
    int a;
    cout << "Tapez la valeur de a : "; cin >> a;

    switch(a)
    {
        case 1 :
            cout << "a vaut 1" << endl;
            break;

        case 2 :
        case 4 :
            cout << "a vaut 2 ou 4" << endl;
            break;

        case 3 :
        case 7 :
        case 8 :
            cout << "a vaut 3, 7 ou 8" << endl;
            break;

        default :
            cout << "valeur autre" << endl;
            break;
    }
    return 0;
}
```

Ce programme demande à l'utilisateur de taper une valeur entière et la stocke dans la variable a. On teste ensuite la valeur de a : en fonction de cette valeur on affiche respectivement les messages "a vaut 1", "a vaut 2 ou 4", "a vaut 3, 7 ou 8", ou "valeur autre".

Exécution 1 :

```
Tapez la valeur de a : 1
a vaut 1
```

Exécution 2 :

```
Tapez la valeur de a : 4
a vaut 2 ou 4
```

Exécution 3 :

```
Tapez la valeur de a : 7
a vaut 3, 7 ou 8
```

Exécution 4 :

```
Tapez la valeur de a : 11
valeur autre
```

L'instruction `switch` de cet exemple regroupe le traitement des valeurs 2 et 4, et des valeurs 3, 7 et 8.

Le for "classique"

Le `for` est une structure de contrôle qui permet de répéter un certain nombre de fois une partie d'un programme.

Syntaxe

```
for(instruction_init ; condition ; instruction_suivant)
    instruction_répétée
```

instruction_init

Une instruction (ou une série d'instruction séparées par une virgule) d'initialisation de la boucle (initialiser un compteur à 0, pointer le premier élément d'une liste, ...).

condition

Expression booléenne de la condition de répétition de la boucle.

instruction_suivant

Une instruction (ou une série d'instruction séparées par une virgule) pour passer à l'itération suivante (incrémenter un index, passer à l'élément suivant, ...)

instruction_répétée

Une instruction ou un bloc d'instruction répété à chaque itération de la boucle.

Sémantique

1. on exécute l'*instruction_init*.
2. On teste la *condition* :
 - tant qu'elle est vraie, on exécute l'*instruction_repetée*, puis l'*instruction_suivant* puis on revient au 2.
 - si elle est fausse, la boucle est terminée et on passe à l'instruction suivante.
3. L'*instruction_repetée* peut être une suite d'instructions entre accolades.

Exemples

Premier cas simple :

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

int main()
{
    int i;
    for(i=0 ; i<10 ; i++)
        cout << "BONJOUR" << endl;
    return 0;
}
```

Second cas où deux index dans un tableau sont utilisés pour le parcourir à partir des deux bouts : le début et la fin.

```
int main()
{
    // 10 entiers indexés de 0 à 9 inclus :
    int elements[10] = { 3, 14, 15, 9, 26, 5, 35, 8 };

    int i, j;
    {
        for (i=0, j=9; i <= j; i++, j--)
            cout << "Elements"
                 << " [" << i << "]= " << elements[i]
                 << " [" << j << "]= " << elements[j]
                 << endl;
    }

    return 0;
}
```

Instructions multiples

Il existe également une pratique répandue qui consiste à initialiser plusieurs variables, évaluer plusieurs conditions ou, plus fréquemment, modifier plusieurs variables. Exemple :

```
float c=10;
for(int a=2, b=a; a<2, b*=a; a++, c/=b)
{
}
```

Ceci compilera parfaitement, mais est également parfaitement illisible. Dans la plupart des cas, il vaut mieux n'utiliser la boucle for elle-même que pour traiter le ou les compteurs, afin de conserver un code compréhensible. Cela compile parce qu'en C++, la virgule ',' est un opérateur comme les autres, qui évalue l'expression qui la précède et celle qui la suit, et renvoie le résultat de la dernière. Pour donner un exemple plus clair :

```
int b = 2,10; // b=10. Vous pouvez compiler pour vérifier.
```

Boucle infinie

Quand aucune condition n'est spécifiée, celle-ci vaut `true` par défaut, ce qui crée donc une boucle infinie. Une boucle infinie est parfois nécessaire quand on ne connaît pas l'instant où la boucle doit être arrêtée, et que la condition d'arrêt est évaluée au cœur de la boucle (gestion de messages, écoute de connexions réseau, ...).

```
for(;;)
{
    // ...
}
```

le for "moderne"

Le dernier standard C++ (surnommé C++11) a ajouté une nouvelle syntaxe pour la boucle for. En effet, on s'est aperçu que l'on utilise le plus souvent la boucle for pour parcourir une collection (une collection est un tableau classique ou un conteneur comme ceux de la stl.) en entier, et que donc le même code revenait très souvent (à la langue des commentaires près :). Voici des exemples de code fréquents :

```
typedef std::list<int> MaListe;
MaListe maListe;
for(MaListe::iterator it=maListe.begin(); it!=maListe.end(); ++it)
{
}

int tableau[]={0,1,2,3,4};
const int tableauMax=5;//il y a 5 cases dans tableau
for(int i=0; i<tableauMax; ++i)
{
}
```

Très répétitif, non (surtout le 1er exemple d'ailleurs) ? Donc, une nouvelle syntaxe a été créée qui permet de s'affranchir d'une grande partie de ces frappes et donc de prolonger la durée de vie de nos claviers ;)

Syntaxe

`for(variable : collection) instruction`

Sémantique

variable indique la variable qui recevra les valeurs issues de *collection*, en commençant par la première jusqu'à la dernière. Si *variable* n'existe pas, il est possible de la déclarer dans la même ligne, de la même façon que pour l'écriture classique.

Un bémol de cette syntaxe est que le compteur est perdu : si vous avez besoin du compteur dans le bloc d'instruction (ou plus tard) cette syntaxe est inutilisable.

Exemple

Appliqué au dernier exemple, voici le résultat :

```
typedef std::list<int> MaListe;
MaListe maListe;
for(int i : maListe)
{
}

int tableau[]={0,1,2,3,4};
for(int i: tableau)
{
}
```

Beaucoup plus clair, non ?

Le while

Syntaxe

```
while (condition) instruction;
```

Sémantique

On teste la condition :

- si elle est vraie, on exécute l'instruction et on recommence.
- si elle est fausse, la boucle est terminée, on passe à l'instruction suivante.

L'instruction peut être une suite d'instructions entre accolades.

Exemple de programme

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

int main()
{
    int i = 0;

    while (i < 10)
```

```
{
    cout << "La valeur de i est : " << i << endl;
    i++;
}

cout << "La valeur finale de i est : " << i << endl;
return 0;
}
```

Exécution

```
La valeur de i est : 0
La valeur de i est : 1
La valeur de i est : 2
La valeur de i est : 3
La valeur de i est : 4
La valeur de i est : 5
La valeur de i est : 6
La valeur de i est : 7
La valeur de i est : 8
La valeur de i est : 9
La valeur finale de i est : 10
```

Explications

La variable `i` est initialisée à 0.

À chaque étape, à la fin du corps du `while`, on incrémente `i` de 1.

On exécute donc le corps du `while` la première fois avec `i` valant 0, la dernière fois avec `i` valant 9.

Lorsqu'on sort du `while`, `i` vaut 10.

Le `do ... while`

Syntaxe

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

Sémantique

1. on exécute les instructions ;
2. on évalue la condition ;
3. si elle est vraie, on recommence au 1 ;
4. si elle est fausse, la boucle est terminée, on passe à l'instruction suivante.

Exemple

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

int main()
{
    int i = 0;

    do
    {
        cout << "La valeur de i vaut : " << i << endl;
        i++;
    }
    while ( i < 10);
}
```

```

    cout << "La valeur finale de i est " << i << endl;
    return 0;
}

```

Exécution :

```

;La valeur de i vaut : 0
;La valeur de i vaut : 1
;La valeur de i vaut : 2
;La valeur de i vaut : 3
;La valeur de i vaut : 4
;La valeur de i vaut : 5
;La valeur de i vaut : 6
;La valeur de i vaut : 7
;La valeur de i vaut : 8
;La valeur de i vaut : 9
;La valeur finale de i est : 10

```

Le goto

Le `goto` est une structure de contrôle obsolète et ne doit plus être utilisé.

Le break

L'instruction `break` sert à "casser" ou interrompre une boucle (`for`, `while` et `do`), ou un `switch`.

Syntaxe

```

break;

```

Sémantique

- Sort de la boucle ou de l'instruction `switch` la plus imbriquée.

Exemple

```

#include <iostream>
using namespace std;

int main()
{
    for (int i = 0; i < 10; i++)
    {
        cout << "La variable i vaut : " << i << endl;
        if (i == 5)
            break;
    }

    return 0;
}

```

Ce programme interrompt sa boucle lors de la sixième itération.

Exécution

```

;La variable i vaut : 0
;La variable i vaut : 1
;La variable i vaut : 2
;La variable i vaut : 3

```

```
La variable i vaut : 4
La variable i vaut : 5
```

Note importante

Le `break` rend le code difficile à lire, à l'exception de son utilisation dans un `switch`, il ne doit pas être utilisé (au même titre que le `goto`).

Le continue

L'instruction continue sert à "continuer" une boucle (`for`, `while` et `do`) avec la prochaine itération.

Syntaxe

```
continue;
```

Sémantique

Passe à l'itération suivante de la boucle la plus imbriquée.

Exemple

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

int main()
{
    for (int i = 0; i < 10; i++)
    {
        if (i == 5) continue;
        cout << "La variable i vaut : " << i << endl;
    }

    return 0;
}
```

Ce programme saute l'affichage de la sixième itération.

Exécution

```
La variable i vaut : 0
La variable i vaut : 1
La variable i vaut : 2
La variable i vaut : 3
La variable i vaut : 4
La variable i vaut : 6
La variable i vaut : 7
La variable i vaut : 8
La variable i vaut : 9
```

Voir aussi

- [Exercices](#)
- [Exercices if...else et switch](#)

Les fonctions

Utilisation des fonctions

Le C++ est un langage procédural (entre autres paradigmes), on peut définir des fonctions qui vont effectuer une certaine tâche. On peut paramétrer des fonctions qui vont permettre de paramétrer cette tâche et rendre ainsi les fonctions réutilisables dans d'autres contextes.

Exemple général trivial :

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

int doubleur(int a_locale_doubleur) // Mise en évidence facultative mais très pédagogique de la portée locale
de la variable.
{
    return 2*a_locale_doubleur;
}

int main()
{
    int nombre = 21;
    cout << doubleur(nombre) << endl; // 42.
    return 0;
}
```

Compléments indispensables pour les débutants

- On indique devant le nom de la fonction la nature du return : int, string, char, double... (void en l'absence de return).
- Les variables définies localement restent locales. On peut ajouter au nom de la variable un affixe comme au début de la page, avec par exemple `_locale_` et le nom de la fonction, pour plus de facilité de relecture pour les grands débutants, mais cela n'est pas habituel.
- Une fonction pourra appeler d'autres fonctions et ainsi de suite.
- Une fonction peut même s'appeler elle-même : on parle alors de fonctions récursives.
- Limitations ("Ce ne sont pas des bugs, ce sont des fonctionnalités !") actuelles du C++ :
 - Le return n'est **jamais** un tableau.
 - Le return renvoie **une seule** valeur.

Prototype d'une fonction

Le prototype va préciser le nom de la fonction, donner le type de la valeur de retour de la fonction (**void** quand il n'y a pas de retour), et donner les types des paramètres éventuels. Le prototype d'une fonction est facultatif. Si on ne met pas de prototype, on devra rédiger la fonction **avant** le main, au lieu de la rédiger **après** ou **dans un fichier séparé** (extension `.h`).

syntaxe

```
type identificateur(paramètres);
```

Exemple

```
// prototype de la fonction f :
double f(double x, double y);
```

Rôle

Le rôle d'un prototype n'est pas de définir les instructions de la fonction, mais donner sa signature. Il est utilisé pour spécifier que la fonction existe, et est implémentée ailleurs (dans un autre fichier, une librairie, ou à la fin du fichier source).

On les trouve principalement dans les fichiers d'en-tête (extension .h).

Définition d'une fonction

La définition va reprendre le prototype mais va préciser cette fois le contenu de la fonction (le corps).

Syntaxe

```
type identificateur(paramètres)
{
    ... Liste d'instructions ...
}
```

Exemple

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

// définition de la fonction f :
double f(double x, double y)
{
    double a;
    a = x*x + y*y;
    return a;
}

int main()
{
    double u, v, w;
    cout << "Tapez la valeur de u : "; cin >> u;
    cout << "Tapez la valeur de v : "; cin >> v;

    w = f (u, v); //appel de notre fonction

    cout << "Le résultat est " << w << endl;
    return 0;
}
```

Exemple avec prototype

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

// prototype de la fonction f :
double f(double x, double y);

int main()
{
    double u, v, w;
    cout << "Tapez la valeur de u : "; cin >> u;
    cout << "Tapez la valeur de v : "; cin >> v;

    w = f (u, v); //appel de notre fonction

    cout << "Le résultat est " << w << endl;
    return 0;
}

// définition de la fonction f :
double f(double x, double y)
{
    double a;
    a = x*x + y*y;
    return a;
}
```

Dans cet exemple, le prototype est nécessaire, car la fonction est définie **après** la fonction `main` qui l'utilise. Si le prototype est omis, le compilateur signale une erreur.

Portée des variables

Présentation

Une fonction peut accéder :

- à ses différents paramètres,
- à ses variables définies localement,
- aux variables globales (on évitera au maximum d'utiliser de telles variables).

On appelle environnement d'une fonction l'ensemble des variables auxquelles elle peut accéder. Les différents environnements sont donc largement séparés et indépendants les uns des autres. Cette séparation permet de mieux structurer les programmes.

Exemple

```

#include <iostream>
using namespace std;

int b; // variable globale

double f(double x, double y)
{
    double a; // variable locale à la fonction f
    a = x*x + y*y;
    return a;
}

double g(double x)
{
    int r, s, t; // variables locales à la fonction g
    /* ... */
}

int main()
{
    double u, v, w; // variables locales à la fonction main
    /* ... */
    return 0;
}
    
```

- `b` est une variable globale (à éviter : une structure la remplace avantageusement !).
- `f` peut accéder
 - à ses paramètres `x` et `y`.
 - à sa variable locale `a`.
 - à la variable globale `b`.
- `g` peut accéder
 - à son paramètre `x`.
 - à ses variables locales `r,s` et `t`.
 - à la variable globale `b`.
- la fonction `main` peut accéder
 - à ses variables locales `u,v` et `w`.
 - à la variable globale `b`.

Passage de paramètres par pointeur

Passer un paramètre par pointeur permet de modifier la valeur pointée en utilisant l'opérateur de déréférencement `*`.

Pour passer un pointeur comme argument de fonction, il faut en spécifier le type dans la définition de la fonction et (si pertinent), dans le prototype

de la fonction, comme ceci :

```
void passagePointeur(int *); // Prototype d'une fonction renvoyant void
// et prenant comme argument un pointeur vers un int

void passagePointeur(int * ptr) // Définition d'une fonction renvoyant void
// et prenant comme argument un pointeur vers un int appelé ptr
{
    /* ... */
}
```

De même, il faut, lors de l'appel de fonction, non pas spécifier le nom de la variable comme lors d'un passage par valeur, mais son adresse. Pour ceci, il suffit de placer le signe & devant la variable.

```
int a = 5; // Initialisation d'une variable a, de valeur 5
passageValeur( a ); // Appel d'une fonction par valeur
passagePointeur( &a ); // Appel d'une fonction par pointeur
```

Notez donc le & devant la variable, ceci a pour effet de passer l'adresse mémoire de la variable.

Examinons le programme simple suivant :

```
#include <iostream>
using std::cout;

void passagePointeur(int *);
void passageValeur(int);

int main()
{
    int a = 5;
    int b = 7;

    cout << "a : " << a << endl;
    cout << "b : " << b; // Affiche les deux variables

    passageValeur( a); // Appel de la fonction en passant la variable a par valeur
    // Une copie de la valeur est transmise à la fonction

    passagePointeur( &b); // Appel de la fonction en passant l'adresse de la variable b par pointeur
    // Une copie de l'adresse est transmise à la fonction

    cout << endl;
    cout << "a : " << a << endl;
    cout << "b : " << b; // Réaffiche les deux variables

    system( "PAUSE" );
    return 0;
}

void passagePointeur(int * ptr)
{
    int num = 100;
    cout << endl;
    cout << "*ptr : " << *ptr; // Affiche la valeur pointée

    * ptr = 9; // Change la valeur pointée;

    ptr = &num; // <-- modification de l'adresse ignorée par la fonction appelante
}

void passageValeur(int val)
{
    cout << endl;
    cout << "val : " << val;

    val = 12; // <-- modification de la valeur ignorée par la fonction appelante
}
```

On affiche les deux variables puis on appelle les deux fonctions, une par valeur et une par pointeur. Puis on affiche dans ces deux fonctions la valeur et on modifie la valeur. De retour dans main, on réaffiche les deux variables a et b. a, qui a été passée par valeur, n'a pas été modifiée et a toujours sa valeur initiale (5), spécifiée à l'initialisation. Or, b n'a plus la même valeur que lors de son initialisation, 7, mais la valeur de 9. En effet, lors d'un appel par valeur, une copie de cette valeur est créée, donc lorsqu'un appel de fonction par valeur est effectué, on ne modifie pas la valeur d'origine mais une copie, ce qui fait que lorsqu'on retourne dans la fonction d'origine, les modifications effectuées sur la variable dans la fonction appelée ne sont pas prises en comptes. En revanche, lors d'un appel par pointeur, il s'agit de la variable elle-même qui est modifiée (puisque l'on passe son adresse). Donc si elle est modifiée dans la fonction appelée, elle le sera également dans la fonction appelante.

Les avantages sont que cet appel nécessite moins de charge de travail pour la machine. En effet, par valeur, il faut faire une copie de l'objet, alors que par pointeur, seule l'adresse de l'objet est copiée. L'inconvénient, c'est qu'on peut accidentellement modifier dans la fonction appelée la valeur de la variable, ce qui se répercutera également dans la fonction appelante. La solution est simple : il suffit d'ajouter `const` dans le prototype de fonction de cette manière :

```
void passagePointeur(const int *);
// En d'autres termes, un pointeur vers un int constant
```

ainsi que la définition de fonction comme ceci :

```
void passagePointeur(const int * ptr)
{
    // * ptr = 9; // <- maintenant interdit
}
```

En recompilant le programme mis plus haut avec ces deux choses, on constate un message d'erreur indiquant que l'on n'a pas le droit de modifier une valeur constante.

Passage de paramètres par référence

Passer un paramètre par référence a les mêmes avantages que le passage d'un paramètre par pointeur. Celui-ci est également modifiable. La différence est que l'opérateur de déréférencement n'est pas utilisé, car il s'agit déjà d'une référence.

Le passage de paramètre par référence utilise une syntaxe similaire au passage par pointeur dans la déclaration de la fonction, en utilisant `&` au lieu de `*`.

Par exemple :

```
void incrementer(int& value)
// value : référence initialisée quand la fonction est appelée
{
    value++;
    // la référence est un alias de la variable passée en paramètre
}

void test()
{
    int a = 5;
    cout << "a = " << a << endl; // a = 5

    incrementer(a);

    cout << "a = " << a << endl; // a = 6
}
```

Le paramètre ainsi passé ne peut être qu'une variable. Sa valeur peut être modifiée dans la fonction appelée, à moins d'utiliser le mot `const` :

```
void incrementer(const int& value)
{
    value++; // <- erreur générée par le compilateur
}

void test()
{
```

```

int a = 5;
cout << "a = " << a << endl;
incrementer(a);
cout << "a = " << a << endl;
}

```

La question que l'on peut se poser est *puisque le passage par référence permet de modifier le paramètre, pourquoi l'en empêcher, et ne pas utiliser un simple passage par valeur* ? La réponse se trouve lorsqu'on utilise des objets ou des structures. Le passage par valeur d'un objet ou d'une structure demande une recopie de la valeur de ses membres (utilisant un constructeur de recopie pour les objets). L'avantage de passer une référence est d'éviter la recopie. Le mot `CONST` permet de garantir qu'aucun membre de l'objet ou de la structure n'est modifié par la fonction.

Pointeur de fonction

Un pointeur de fonction stocke l'adresse d'une fonction, qui peut être appelée en utilisant ce pointeur.

La syntaxe de la déclaration d'un tel pointeur peut paraître compliquée, mais il suffit de savoir que cette déclaration est identique à celle de la fonction pointée, excepté que le nom de la fonction est remplacé par (`* pointeur`).

Obtenir l'adresse d'une fonction ne nécessite pas l'opérateur `&`. Il suffit de donner le nom de la fonction seul.

Exemple :

```

#include <iostream>
#include <iomanip>

using namespace std;

int (* pf_comparateur)(int a, int b);
// peut pointer les fonctions prenant 2 entiers en arguments et retournant un entier

int compare_prix(int premier, int second)
{
    return premier - second;
}

int main()
{
    pf_comparateur = &compare_prix;
    cout << "compare 1 et 2 : " << (*pf_comparateur)(1, 2) << endl;
    return 0;
}

```

La syntaxe d'appel à une fonction par un pointeur est identique à l'appel d'une fonction classique.

Passage de fonctions en paramètre

Il s'agit en fait de passer un pointeur de fonction.

Exemple :

```

void sort_array( int[] data, int count, int (* comparateur_tri)(int a, int b) );

```

Pour clarifier la lecture du code, il est préférable d'utiliser l'instruction `typedef` :

```

typedef int (* pf_comparateur)(int a, int b);

void sort_array( int[] data, int count, pf_comparateur comparateur_tri );

```

Exercices

Exercice 1

Créez une fonction `factorielle` pour calculer la factorielle d'un entier n. Elle retourne un entier long. Créez une fonction non récursive.

Aide

La factorielle (noté ! en mathématiques) se définit ainsi:

$$!1 = 1 = 1$$

$$!2 = 2 \times 1 = 2$$

$$!3 = 3 \times 2 \times 1 = 6$$

$$!4 = 4 \times 3 \times 2 \times 1 = 24$$

Solution

Voici le code source:

```
long factorielle(int n)
{
    long r = 1;
    while(n-- > 0)
        r *= n;
    return r;
}
```

Exercice 2

Créez une autre fonction `factorielle` mais récursive cette fois.

Aide

Une fonction récursive est une fonction qui s'appelle elle-même.

$$!n = n \times !(n - 1)$$

Solution

Voici le code source:

```
long factorielle(long n)
{
    if(n == 1) return 1;
    return n*factorielle(n - 1);
}
```

Exercice 3

Donner la déclaration d'un pointeur `fct` sur la dernière fonction `factorielle`.

Solution

Voici le code:

```
long (*fct)(long);
```

Exercice 4

En C++, les systèmes `callback` sont des systèmes qui enregistrent des fonctions, afin de les exécuter en cas d'évènement (comme par exemple si on touche le clavier).

Albert n'est pas fort en maths. Il veut un système pour faire des racines carrées, des carré, des cosinus et des factorielles. Il utilise l'environnement `callback` situé dans l'annexe 1. Cet environnement crée une structure `f_maths`, qui enregistre une fonction et son symbole mathématique. Pour utiliser sa calculatrice, il doit enregistrer correctement les fonctions dans le `gestionnaire`. Ensuite, le programme interroge l'utilisateur le symbole de son opération. Puis, l'utilisateur entre sa valeur et le programme effectue l'opération. Compléter le programme en annexe avec la fonction `main()`

uniquement.

Annexe 1

Système:

```

#include <cmath>
#include <stdio.h>

typedef double (*fct)(double);
int nb_fct;

struct f_maths { fct fonction; char symb; };

f_maths gestionnaire[10];

void initialiser(){ nb_fct = 0; }

double carré(double a) { return a * a; }

double factorielle(double n)
{
    if(n == 1) return 1;
    return n*factorielle(n - 1);
}

void enregistrer(fct f, char sy)
{
    struct s;
    s.fonction = f;
    s.symb = sy;
    gestionnaire[nb_fct++] = s;
}

void executer()
{
    char sy;
    double val;
    printf("Entrer le symbole:\n");
    scanf("%c", &sy);
    printf("Entrer la valeur:\n");
    scanf("%g", &val);
    for(int i = 0; i < nb_fct; i++)
        if(gestionnaire[i].symb == sy)
        {
            printf("Le résultat est %g.\n", (*gestionnaire[i].fonction)(val));
            return;
        }
    printf("Aucune fonction trouvée.\n");
}

```

Aide

Pour faire des cosinus et des racines carrées, cmath contient les fonctions `cos()` et `sqrt()`.

Voir aussi

- [Exercices](#)

Les structures

Présentation

Les structures permettent de regrouper plusieurs variables dans une même entité. Ainsi il est possible de construire de nouveaux types plus complexes.

Syntaxe

```
struct identificateur
{
    // liste des différents champs constituant notre structure
    // utilise la même syntaxe que la déclaration de variables
} [variables];
```

identificateur identifie le nouveau type de données. Une variable de ce type est donc précédé de `struct identificateur` ou *identificateur* car le mot `struct` est optionnel.

[*variables*] est optionnel et permet de déclarer une ou plusieurs variables de ce nouveau type.

Exemple

L'exemple ci-dessous illustre l'utilisation d'une structure nommée `Position` comportant deux champs (appelées également variables membres) `x` et `y` de type `double`.

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

struct Position
{
    double x;
    double y;
};

int main()
{
    Position A, B;
    double dx, dy, distance;

    cout << "Tapez l'abscisse de A : "; cin >> A.x;
    cout << "Tapez l'ordonnée de A : "; cin >> A.y;
    cout << "Tapez l'abscisse de B : "; cin >> B.x;
    cout << "Tapez l'ordonnée de B : "; cin >> B.y;

    dx = A.x - B.x;
    dy = A.y - B.y;
    distance = sqrt( dx*dx + dy*dy );

    cout << "La distance AB vaut : " << distance << endl;

    return 0;
}
```

Les variables `A` et `B` sont deux structures de type `Position`. L'accès à un champ d'une structure se fait par l'opérateur point (`.`) séparant la variable structure du nom du membre accédé.

Pointeur vers une structure

On peut écrire naturellement

```
Point* pA;
```



```
cout << "Tapez l'abscisse de A : "; cin >> (*pA).x;
cout << "Tapez l'ordonnée de A : "; cin >> (*pA).y;
```

Pour faciliter la lecture et éviter les erreurs, on utilise l'opérateur `->`. On peut écrire `pA->x` à la place de `(*pA).x`. Attention, `*pA.x` donne une erreur de compilation, à cause de la priorité des opérateurs.

```
Point* pA;
cout << "Tapez l'abscisse de A : "; cin >> pA->x;
cout << "Tapez l'ordonnée de A : "; cin >> pA->y;
```

Copier une structure

Copier une structure se fait très simplement

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

struct Point
{
    double x;
    double y;
};

int main()
{
    Point A, B;

    A.x = 2.0;
    A.y = 3.0;

    B = A;

    cout << "A a été recopié dans B : " << B.x << ", " << B.y << endl;

    return 0;
}
```

Voir aussi

- [Exercices](#)

Les classes

La notion de classe en programmation orientée objet

Une classe permet de regrouper dans une même entité des données et des fonctions membres (appelées aussi méthodes) permettant de manipuler ces données. La classe est la notion de base de la programmation orientée objet. Il s'agit en fait d'une évolution de la notion de structure, qui apporte de nouvelles notions orientées objet absolument fondamentales. Un **objet** est un élément d'une certaine classe : on parle de **l'instance d'une classe**.

L'encapsulation en C++

L'encapsulation est un mécanisme qui interdit d'accéder à certaines données depuis l'extérieur de la classe. Ainsi, un utilisateur de la classe ne pourra pas accéder à tous les éléments de celles-ci. Il sera obligé d'utiliser certaines fonctions membres de la classe (celles qui sont publiques). L'avantage de cette restriction est qu'il empêche par exemple un utilisateur de la classe de mettre les données dans un état incohérent. Vue de l'extérieur, la classe apparaît comme une boîte noire, qui a un certain comportement à laquelle on ne peut accéder que par les méthodes publiques. Cette notion est extrêmement puissante et permet d'éviter de nombreux effets de bord.

L'encapsulation permet de distinguer très nettement ce que fait la classe et sa sémantique précises de la manière dont on l'implémente. Cette réflexion permet de répondre dans un premier temps à la question *Comment utilise-t-on la classe ?* Ce n'est que dans un second temps que l'aspect technique entre en jeu et que le programmeur doit répondre à la question *Comment vais-je programmer les fonctionnalités de la classe qui ont été spécifiées ?* L'encapsulation permet de faire abstraction du fonctionnement interne (c'est-à-dire, l'implémentation) d'une classe et ainsi de ne se préoccuper que des services rendus par celle-ci.

Le C++ implémente l'encapsulation en permettant de déclarer les membres d'une classe avec l'un des mots réservés `public`, `private` et `protected`. Ainsi, lorsqu'un membre est déclaré:

- *public*, il sera accessible depuis n'importe quelle fonction.
- *private*, il sera uniquement accessible d'une part, depuis les fonctions qui sont membres de la classe et, d'autre part, depuis les fonctions et classes autorisées explicitement par la classe (par l'intermédiaire du mot réservé `friend`). Ces dernières fonctions/classes sont appelées **fonctions/classes amies** de la classe.
- *protected*, il aura les mêmes restrictions que s'il était déclaré *private*, mais il sera en revanche accessible par les classes filles.

Le C++ n'impose pas l'encapsulation des membres dans leurs classes. On pourrait donc déclarer tous les membres publiques, mais en perdant une partie des bénéfices apportés par la programmation orientée objet. Il est de bon usage de déclarer toutes les données privées, ou au moins protégées, et de rendre publiques les méthodes agissant sur ces données. Ceci permet de cacher les détails de l'implémentation de la classe.

Les fonctions membres

Parmi les fonctions membres, on distingue :

- les **Accesseurs**: ce sont des fonctions membres qui ne modifient pas l'état de l'objet. Le C++ permet de déclarer une fonction membre *const*, indiquant au compilateur qu'elle ne modifie pas l'état de l'objet. Si l'implémentation de la fonction *const* tente de modifier une variable membre de la classe, le compilateur signalera une erreur. Seules les fonctions membres *const* peuvent être invoquées depuis un objet déclaré *const*. Une tentative d'appel à une fonction membre non *const* depuis un objet *const* échouerait à la compilation.
- les **Modificateurs** : ce sont des fonctions membres qui peuvent modifier l'état de l'objet. Dans ce cas, on omet le *const* dans la déclaration. Si une fonction membre déclarée non *const* ne modifie pas l'état de l'objet, il y a lieu de se demander s'il ne faudrait pas la déclarer *const*.

Déclaration

La déclaration d'une fonction membre se fait de la même manière qu'une fonction régulière mais au sein de la portée de la classe. En outre, on pourra la suffixer du mot clé *const* pour signifier qu'il s'agit d'un accesseur.

- **Syntaxe**

```
class A
{
public:
```

```

    int getValue() const;
    void setValue(int value);
private:
    int value;
};

```

Définition

Dans le cas d'une définition au sein de la classe même, elle est faite comme une fonction régulière. Dans ce cas, la fonction membre est automatiquement considérée comme *inline*.

Dans le cas d'une définition en dehors de la classe, il faudra préfixer le nom de la fonction membre par le nom de la classe suivi de l'opérateur de portée (::). Dans ce cas, elle est automatiquement considérée comme non *inline*.

■ Syntaxe

```

class A
{
public:
    int getValue() const { return this->value; } // définition au sein de la classe
    void setValue(int value);
    void print() const;
private:
    int value;
};

inline void A::setValue(int value) // définition en dehors de la classe (inline)
{
    this->value = value;
}

void A::print() const // définition en dehors de la classe (non inline)
{
    std::cout << "Value=" << this->value << std::endl;
}

```

En règle générale, les fonctions membres non *inline* devront être déclarées dans le fichier source .cpp de la classe, pour éviter d'avoir plusieurs fois la même définition au moment de l'édition des liens, et les fonctions membres *inline* devront être déclarées dans le fichier d'entête .hpp de la classe, pour permettre au compilateur de générer le code à chaque appel de la fonction.

Constructeurs et destructeurs

Les constructeurs et destructeur d'une classe sont des fonctions membres particulières de cette classe.

Constructeurs

- Lorsqu'on crée une nouvelle instance d'une classe, les données membres de cette instance ne sont pas initialisées par défaut. Un constructeur est une fonction membre qui sera appelée au moment de la création d'une nouvelle instance d'une classe. Il peut y avoir plusieurs constructeurs d'une classe avec différents paramètres qui serviront à initialiser notre classe. Le constructeur d'une classe A est appelé automatiquement lorsqu'on crée une instance en écrivant : `A toto`; ou encore `A toto(6,9)`; Ils sont également appelés lorsqu'une instance est créée grâce à l'opérateur `new`.
- Parmi les constructeurs, on retrouve une série de constructeurs particuliers:
 - Le **constructeur par défaut** : c'est le constructeur sans argument. Si aucun constructeur n'est présent explicitement dans la classe, il est généré par le compilateur.
 - Le **constructeur de copie** : c'est un constructeur à un argument dont le type est celui de la classe. En général, l'argument est passé par référence constante. Plus rarement, il peut l'être par valeur ou référence non constante. Si il n'est pas explicitement présent dans la classe, il est généré par le compilateur. Notons encore que:
 - Si un constructeur de copie est fourni dans une classe, il sera très probable de devoir fournir l'opérateur d'assignation avec le même type d'argument.
 - Il est possible de rendre une classe non copiable en privatisant la déclaration du constructeur de copie et de l'opérateur d'assignation. Dans ce cas, il n'est pas nécessaire (recommandé) de fournir leur

définition.

- Les **constructeurs de conversion implicite** : il s'agit de constructeurs à un seul argument dont le type est différent de celui de la classe et qui ne sont pas déclarés *explicit*. Ces constructeurs rendent possible la conversion implicite du type d'argument vers le type de la classe.
- **Syntaxe** : (*Pour une classe A*)
 - `A();` // constructeur par défaut
 - `A(const A &);` // constructeur de copie
 - `A(const T &);` // constructeur de conversion implicite
 - `explicit A(const T &);` // constructeur à un argument

Destructeurs

Les destructeurs sont appelés lorsqu'une instance d'une classe doit être détruite. Cela arrive à la fin de l'exécution d'une fonction/méthode: si vous avez déclaré, localement à une fonction/méthode, un élément d'une classe *A* par *A toto*; alors à la fin de l'exécution de la fonction/méthode, le destructeur de la classe est appelé. Il est également appelé lorsqu'on détruit une instance grâce à `delete`. Il ne peut y avoir qu'un seul destructeur pour une même classe. S'il n'est pas explicitement présent dans la classe, il sera généré par le compilateur. La syntaxe est identique à celle du constructeur, mais le destructeur est introduit par un tilde:

```
~A();
```

Le destructeur ne reçoit aucun argument. Il est utile lorsque par exemple, une classe Véhicule a un pointeur sur une classe Moteur. Si la classe est détruite sans appeler un destructeur, la classe moteur ne sera pas supprimée et il y aura une fuite de mémoire, la classe Moteur restant en mémoire mais étant inaccessible.

Exemples de classes

Dans cet exemple de classe, les fichiers **Point.h**, **Point.cpp** et **main.cpp** vont vous être exposés. Tout d'abord, **Point.h**:

```
#ifndef POINT_H
#define POINT_H

#include <iostream>
using namespace std;

class Point
{
public:
    // Constructeurs
    Point();
    Point(double x, double y);

    //Accesseurs et mutateurs
    void setX(double x);
    void setY(double y);
    double getX() const;
    double getY() const;

    // Autres méthodes
    double distance(const Point &P) const;
    Point milieu(const Point &P) const;

    void saisir();
    void afficher() const;

private:
    double x,y;
};

#endif
```

Voici le fichier **Point.cpp**:

```
#include "Point.h"
```

```

#include <cmath>
#include <iostream>
using namespace std;

Point::Point() : x(0), y(0)
{}

Point::Point(double x, double y) : x(x), y(y)
{}

void Point::setX(double x)
{
    this->x = x;
}

void Point::setY(double y)
{
    this->y = y;
}

double Point::getX() const
{
    return this->x;
}

double Point::getY() const
{
    return this->y;
}

double Point::distance(const Point &P) const
{
    double dx = this->x - P.x;
    double dy = this->y - P.y;
    return sqrt(dx*dx + dy*dy);
}

Point Point::milieu(const Point &P) const
{
    Point result;
    result.x = (P.x + this->x) / 2;
    result.y = (P.y + this->y) / 2;
    return result;
}

void Point::saisir()
{
    cout << "Tapez l'abscisse : "; cin >> this->x;
    cout << "Tapez l'ordonnée : "; cin >> this->y;
}

void Point::afficher() const
{
    cout << "L'abscisse vaut " << this->x << endl;
    cout << "L'ordonnée vaut " << this->y << endl;
}

```

Et le fichier principal **main.cpp**:

```

#include <iostream>
using namespace std;

#include "Point.h"

int main()
{
    Point A, B, C;
    double d;

    cout << "SAISIE DU POINT A" << endl;
    A.saisir();
}

```

```

cout << endl;

cout << "SAISIE DU POINT B" << endl;
B.saisir();
cout << endl;

C = A.milieu(B);
d = A.distance(B);

cout << "MILIEU DE AB" << endl;
C.afficher();
cout << endl;

cout << "La distance AB vaut : " << d << endl;
return 0;
}
    
```

Les opérateurs new et delete

Il est parfois intéressant de créer dynamiquement de nouvelles instances d'une classe. Cela s'avère indispensable lorsque vous manipulez certaines structures de données complexes. L'opérateur `new` permet de créer une nouvelle instance d'une classe `A` en écrivant :

```

A* pointeur=new A; // Ou bien:
A* pointeur=new A();
    
```

Si le constructeur de la classe `A` prend des arguments, la syntaxe devient:

```

A* pointeur=new A(arguments);
    
```

La variable `pointeur` doit alors être du type `A*`.

La libération se fait en utilisant l'opérateur `delete` et le destructeur de la classe est appelé:

```

delete pointeur;
    
```

Surcharge d'opérateurs

Présentation

Pour certaines classes, la notion d'addition ou de multiplication est totalement naturelle. La surcharge d'opérateurs permet d'écrire directement $U+V$ lorsqu'on veut additionner deux instances `U` et `V` d'une même classe `A`.

Opérateurs surchargeables

- Les opérateurs unaires : ce sont des opérateurs qui s'appliquent sans argument supplémentaire

opérateur	définition
+	opérateur signe +
-	opérateur négation
*	opérateur déréférencement
&	opérateur adresse
->	opérateur indirection
~	opérateur négation binaire
++	opérateur post/pré-incrémentation
--	opérateur post/pré-décrémentation

- Les opérateurs binaires: ce sont des opérateurs qui s'appliquent moyennant un argument supplémentaire

opérateur	définition
=	opérateur assignation
+	opérateur addition
+=	opérateur addition/assignation
-	opérateur soustraction
-=	opérateur soustraction/assignation
*	opérateur multiplication
*=	opérateur multiplication/assignation
/	opérateur division
/=	opérateur division/assignation
%	opérateur modulo
%=	opérateur modulo/assignation
^	opérateur ou exclusif
^=	opérateur ou exclusif/assignation
&	opérateur et binaire
&=	opérateur et binaire/assignation
	opérateur ou binaire
=	opérateur ou binaire/assignation
<	opérateur plus petit que
<=	opérateur plus petit ou égal
>	opérateur plus grand que
>=	opérateur plus grand ou égal
<<	opérateur de décalage à gauche
<<=	opérateur de décalage à gauche/assignation
>>	opérateur de décalage à droite
>>=	opérateur de décalage à droite/assignation
==	opérateur d'égalité
!=	opérateur d'inégalité
&&	opérateur et logique
	opérateur ou logique
[]	opérateur indexation
,	opérateur enchaînement
()	opérateur fonction

Syntaxe

Les opérateurs unaires doivent être surchargés en tant que méthode de la classe.

La signature générique au sein de la classe est

- `type_retour operator @();` // pour les méthodes non const
- `type_retour operator @() const;` // pour les méthodes const

où @ est l'opérateur à surcharger: + - * / ...

Excepté pour certains d'entre eux, les opérateurs binaires peuvent être surchargés en tant que méthode de la classe ou comme une fonction à 2 arguments.

La signature générique est la suivante

- `type_retour operator @(Arg);` // pour les méthodes non const
- `type_retour operator @(Arg) const;` // pour les méthodes const
- `type_retour operator @(Arg1, Arg2) const;` // pour les fonctions

où @ est l'opérateur à surcharger: + - * / ...

Pour surcharger l'opérateur+ de la classe A, il suffit de créer une méthode:

- `A operator+(const A& a) const;`

ou encore une fonction externe à la classe:

- `A operator+(const A&, const A &)`

Exemple

▪ Fichier Fraction.h

```
#define Fraction_h
#include <iostream>
using namespace std;

class Fraction
{
    friend ostream & operator<<(ostream & out, const Fraction &f);
    friend istream & operator>>(istream &in, Fraction &f);

public:
    Fraction();
    Fraction(int i);
    Fraction(int num, int den);

    Fraction operator+(const Fraction & f) const;
    Fraction operator-(const Fraction & f) const;
    Fraction operator*(const Fraction & f) const;
    Fraction operator/(const Fraction & f) const;

private:
    int num,den;
    static int pgcd(int x, int y);
    void normalise();
};

#endif
```

▪ Fichier Fraction.cpp

```
#include "Fraction.h"
#include <sstream>

Fraction::Fraction() : num(0), den(1)
{
}

Fraction::Fraction(int i) : num(i), den(1)
{
}

Fraction::Fraction(int num, int den) : num(num), den(den)
{
    normalise();
}

ostream & operator<<(ostream &out, const Fraction &f)
{
    if (f.den != 1)
        out << f.num << "/" << f.den;
    else
        out << f.num;

    return out;
}
```



```

istream & operator>>(istream &in, Fraction &f)
{
    string s;
    bool ok = true;

    do
    {
        cout << "Tapez le numerateur : "; getline(in, s);
        istringstream is1(s);
        ok = (is1 >> f.num) && is1.eof();
    } while(!ok);

    do
    {
        cout << "Tapez le denominateur : "; getline(in, s);
        istringstream is2(s);
        ok = (is2 >> f.den) && is2.eof();
    } while(!ok);

    f.normalise();
    return in;
}

int Fraction::pgcd(int x, int y)
{
    int r;

    if (x <= 0 || y <= 0)
        r = -1;
    else
    {
        while (x != 0 && y != 0 && x != y)
        {
            if (y > x)
                y = y % x;
            else
                x = x % y;
        }

        if (x == 0)
            r = y;
        else
            r = x;
    }

    return r;
}

void Fraction::normalise()
{
    int s, n, d, p;
    if (den < 0)
    {
        s = -1;
        d = -den;
    }
    else
    {
        s = 1;
        d = den;
    }

    if(num < 0)
    {
        s = -s;
        n = -num;
    }
    else
        n = num;

    if(n != 0)
    {

```

```

        if(d != 0)
        {
            p = pgcd(n, d);
            n = n/p;
            d = d/p;
            num = n*s;
            den = d;
        }
    }
    else
    {
        num = 0;
        den = 1;
    }
}

Fraction Fraction::operator+(const Fraction & f) const
{
    Fraction r;
    r.num = f.den*num + den*f.num;
    r.den = f.den*den;
    r.normalise();
    return r;
}

Fraction Fraction::operator-(const Fraction & f) const
{
    Fraction r;
    r.num = f.den*num - den*f.num;
    r.den = f.den*den;
    r.normalise();
    return r;
}

Fraction Fraction::operator*(const Fraction & f) const
{
    Fraction r;
    r.num = f.num*num;
    r.den = f.den*den;
    r.normalise();
    return r;
}

Fraction Fraction::operator/(const Fraction & f) const
{
    Fraction r;
    r.num = f.den*num;
    r.den = f.num*den;
    r.normalise();
    return r;
}

```

■ Fichier main.cpp

```

#include <iostream>
using namespace std;

#include "Fraction.h"

int main()
{
    Fraction f1, f2, f3, f4, E;
    cout << "SAISIE de f1 " << endl;  cin >> f1;
    cout << "SAISIE de f2 " << endl;  cin >> f2;

    f3 = Fraction(3, 4);
    f4 = Fraction(5, 8);

    E = (f1 + f3 - f2) / (f1*f2 - f4) + 4;

    cout << "E = " << E << endl;
}

```

```

}
return 0;
}

```

Héritage

Présentation

Dans la conception orientée objet, la généralisation consiste à modéliser des concepts communs à un ensemble d'autres concepts. Les autres concepts deviennent dès lors des spécialisations de la généralisation. Cette manière de modéliser s'appelle l'héritage, car les spécialisations héritent de la généralisation. En C++, les classes peuvent hériter d'autres classes et la relation d'héritage est exprimée à l'aide de l'opérateur de dérivation ":".

À partir d'une classe A, on peut créer une classe B qui possède toutes les caractéristiques de la classe A, à laquelle on ajoute un certain nombre de méthodes qui sont spécifiques à B. Cette notion orientée objet fondamentale s'appelle l'héritage.

On dit que :

- la classe B hérite de la classe A ;
- la classe B est une sous-classe de la classe A ;
- la classe B spécialise la classe A ;
- la classe B étend la classe A ;
- la classe B dérive de la classe A (cette notation n'est toutefois pas appréciée de tous, mais a l'avantage d'éviter les ambiguïtés à propos de super classe et sous classe) ;
- la classe A est une super-classe de la classe B ;
- la classe A généralise la classe B.

On dit aussi que:

- la classe A est une classe de base
- la classe B est une classe dérivée

Il existe en fait trois types d'héritage *public*, *private* ou *protected* qui permet de spécifier si oui ou non une méthode de la classe B peut modifier une donnée membre de la classe A, selon qu'elle soit *public*, *private* ou *protected*.

Syntaxe

```

class B : 'type_héritage' A
{
    .....
};

```

type_héritage est l'un des mots clés d'accès *public*, *protected* ou *private*. Les membres de A sont alors à la fois accessibles à la classe dérivée B et en dehors de ces classes selon la valeur utilisée pour *type_héritage* :

- Quand *type_héritage* est *public*, les membres de la classe A conservent leur accès (privé, protégé et public).
- Quand *type_héritage* est *protected*, les membres publics de la classe A deviennent protégés.
- Quand *type_héritage* est *private*, les membres publics et protégés de la classe A deviennent privés.

Pour résumer, *type_héritage* ne peut que restreindre l'accès des membres de la classe A, ou le conserver.

Exemple

- Fichier **VehiculeRoulant.h**

```

class VehiculeRoulant
{
public:
    VehiculeRoulant();
    void avancer();
    void accelerer();
};

```

```
protected:
    int position, vitesse;
};
```

■ Fichier **VehiculeRoulant.cpp**

```
// Initialement à l'arrêt
VehiculeRoulant::VehiculeRoulant()
    : position(0)
    , vitesse(0)
{}

void VehiculeRoulant::avancer()
{
    position += vitesse;
}

void VehiculeRoulant::accelerer()
{
    vitesse++;
}
```

■ Fichier **Automobile.h**

```
class Automobile : public VehiculeRoulant
{
public:
    Automobile(int places);
protected:
    int m_places;
};
```

■ Fichier **Automobile.cpp**

```
Automobile::Automobile(int places)
    : VehiculeRoulant() // <--- pour initialiser les variables héritées
    , m_places(places) // position et vitesse // initialiser le nombre de places
{}

```

L'appel explicite au constructeur de la classe de base dans cet exemple n'est pas nécessaire car par défaut, le constructeur (sans paramètres) est appelé. Dans le cas où il est nécessaire d'appeler un autre constructeur, il faut le faire dans la liste d'initialisation du constructeur de la classe dérivée.

Méthodes virtuelles

Dans l'exemple précédent, si l'on ajoute une autre méthode à la classe `VehiculeRoulant` :

■ Fichier **VehiculeRoulant.h**

```
class VehiculeRoulant
{
public:
    ...
    void acclerelerEtAvancer();
    ...
};
```

■ Fichier **VehiculeRoulant.cpp**

```
void VehiculeRoulant::accelererEtAvancer()
{
    acclereler();
    avancer();
}
```

et que l'on modifie la façon d'avancer d'une automobile :

■ Fichier **Automobile.h**

```
class Automobile : public VehiculeRoulant
{
public:
    ...
    void avancer();
    ...
};
```

■ Fichier **Automobile.cpp**

```
void Automobile::avancer()
{
    position += vitesse - frottementRoues;
}
```

Alors si on utilise la méthode `accelererEtAvancer()` sur un objet de classe `Automobile`, le résultat sera incorrect, car la méthode `avancer()` appelée sera celle définie par la classe `VehiculeRoulant`.

Pour que ce soit celle de la classe `Automobile` qui soit appelée, il faut que la méthode `avancer()` soit définie comme virtuelle dans la classe de base `VehiculeRoulant` :

■ Fichier **VehiculeRoulant.h**

```
class VehiculeRoulant
{
public:
    ...
    virtual void avancer();
    ...
};
```

■ Fichier **VehiculeRoulant.cpp**

```
...
virtual void VehiculeRoulant::avancer()
{
    position += vitesse;
}
...
```

Le mot clé `virtual` indique une méthode virtuelle, c'est-à-dire que son adresse n'est pas fixe, mais dépend de la classe de l'objet (le compilateur construit une table interne des méthodes virtuelles pour chaque classe). Il est possible d'accéder à une méthode précise de la table des méthodes virtuelles, en indiquant la classe de la méthode à appeler. Par exemple, pour appeler la méthode `avancer()` de `VehiculeRoulant` à partir d'une méthode de `automobile`:

```
void Automobile::avancer()
{
    VehiculeRoulant::avancer();

    position -= frottementRoues;
}
```

Destructeur virtuel

Il est important d'utiliser un destructeur virtuel pour toute classe qui sera dérivée. Dans le cas contraire seul celui de la classe de base est appelé, sans libérer les membres des sous-classes.

Ne pas déclarer un destructeur comme virtuel interdit donc de dériver cette classe.

Héritage multiple

L'héritage multiple permet à une classe d'hériter de plusieurs super-classes. Ceci permet d'intégrer dans la sous-classe plusieurs concept d'abstractions qui caractérisent cette sous-classe. Pour cela, il suffit de déclarer les super-classes les unes après les autres. Si une classe C hérite de A et de B, la syntaxe sera la suivante:

```
class C : 'type_héritage' A, 'type_héritage' B '...'
{
    .....
};
```

Par exemple, si l'on possède une classe décrivant un véhicule roulant, et une classe décrivant un navire, on pourrait créer une classe décrivant un véhicule amphibie qui hériterait des propriétés et méthodes de véhicule roulant et de navire.

L'héritage multiple peut poser des problèmes de définitions multiples de méthodes virtuelles. Supposons que l'on ait une classe **Base** ayant les sous-classes **H1** et **H2** qui en dérivent. Si l'on crée une classe **Finale** héritant à la fois de **H1** et de **H2**, alors, les instanciations des objets de classe provoqueront des appels successifs des constructeurs. D'abord **Finale::Finale()**, puis pour ses parents **H1::H1()** et **H2::H2()**, puis, le parent de **H1** **Base::Base()**, et enfin **Base::Base()** pour le parent de **H2**. On remarque alors que le constructeur de la classe **Base** est appelé à toute instanciation d'un objet de classe **Finale**. En C++, il existe un moyen d'éviter cela et de ne faire appel au constructeur de **Base** qu'une seule fois. Pour cela, il suffit d'indiquer lors des héritages le mot-clé **virtual** pour indiquer au compilateur que les constructeurs des classes multi-héritées ne doivent être appelés qu'une seule fois.

```
class Base
{
    Base::Base()
    {
        cout << "+++ Construction de Base" << endl;
    }

    virtual Base::~Base()
    {
        cout << "--- Destruction de Base" << endl;
    }
};

class H1 : virtual public Base
{
    H1::H1() : Base()
    {
        cout << "+++ Construction de H1" << endl;
    }

    virtual H1::~H1()
    {
        cout << "--- Destruction de H1" << endl;
    }
};

class H2 : virtual public Base
{
    H2::H2() : Base()
    {
        cout << "+++ Construction de H2" << endl;
    }

    virtual H2::~H2()
    {
        cout << "--- Destruction de H2" << endl;
    }
};

class Finale : virtual public H1, virtual public H2
{
    Finale::Finale() : H1(), H2()
```

```

{
    cout << "+++ Construction de Finale" << endl;
}

Finale::~Finale()
{
    cout << "--- Destruction de Finale" << endl;
}
};
    
```

Voir aussi

- [Exercices](#)

Classes abstraites

Présentation

Une classe abstraite est une classe pour laquelle on a défini une méthode mais on a explicitement indiqué qu'on ne fournira aucune implémentation de cette méthode. Il est interdit de créer une instance d'une classe abstraite. Ce mécanisme est extrêmement puissant pour manipuler des concepts abstraits. On peut même avoir une classe pour laquelle toutes les méthodes sont abstraites, on parle alors de classe abstraite pure.

Syntaxe

```
virtual type nom_méthode(paramètres)=0;
```

Une telle méthode définie uniquement dans une sous-classe est nécessairement virtuelle. Le mot-clé `virtual` est donc optionnel.

```
type nom_méthode(paramètres)=0; // Méthode d'une sous classe encore abstraite
type nom_méthode(paramètres); // Méthode d'une sous classe plus abstraite
```

Exemple

```
class Animal
{
public:
    // méthode à implémenter par les sous-classes
    virtual cstring cri() = 0;
}
```

Pointeur de membre

Un pointeur de membre pointe un membre d'un objet (variable ou méthode).

Ce genre de pointeur, rarement utilisé, possède une syntaxe spéciale. L'étoile `*` est remplacée par `class::*` signifiant un pointeur sur un membre de cette classe, ou d'une classe dérivée. Ce genre de pointeur occupe plus de place qu'un pointeur classique, et occupe un nombre d'octets variable selon la classe.

Exemple

```
int CBox::* pBoxInt;
// pBoxInt pointe un entier dans les objets de la classe CBox
```

L'utilisation d'un tel pointeur nécessite un objet de la classe concernée (ou classe dérivée). L'opérateur de déréférencement `*` est alors remplacé par

*objet.** ou bien par *pointeur_objet->**.

Exemple

```
pBoxInt = &CBox::length; // désigne l'adresse du membre length de la classe
CBox mybox(10, 12, 15);
cout << "length = " << mybox.*pBoxInt;

CBox* pbox = new CBox(20, 24, 30);
cout << "length = " << pbox->*pBoxInt;
```

Voir aussi

- [Exercices de développement](#)

Les espaces de noms

Introduction

Les espaces de nom ont été introduits pour permettre de faire des regroupements logiques et résoudre les collisions de noms. Un espace de nom permet de regrouper plusieurs déclarations de variables, fonctions et classes dans un groupe nommé. Une pratique courante du langage C consistant à concaténer des noms entre eux afin de minimiser la probabilité de collisions avec des noms d'autres bibliothèques n'a plus raison d'être en C++. En effet, un même nom pourra être déclaré dans des espaces de noms différents évitant ainsi la collision lorsqu'ils sont inclus en même temps dans une unité.

La déclaration d'un espace de noms se fait à l'aide du mot clé *namespace*.

Tout comme l'accès à un membre d'une classe, l'opérateur de portée `::` permet l'accès à l'ensemble des noms importés d'un espace de noms.

Avant l'introduction des espaces de noms, une pratique du C++ consistait à utiliser des classes pour faire des espaces de noms. Le problème est qu'une classe est instanciable et que sa portée est fermée. A contrario, la portée d'un espace de noms est ouvert, c'est-à-dire qu'il est possible de répartir un espace de noms sur plusieurs unités. Un espace de noms est donc extensible.

Enfin, il est possible d'imbriquer un espace de noms dans un autre espace de noms.

Un exemple d'espace de nom est `std`, défini par la bibliothèque standard, regroupant en outre les flux standards `cin`, `cout` et `cerr`. Ainsi lorsque `<iostream>` est inclus dans une source, il sera possible de les accéder en les dénommant par leurs noms complets `std::cin`, `std::cout` et `std::cerr`.

Pour simplifier le code, et les désigner sans spécifier l'espace de noms, il faut utiliser l'instruction suivante :

```
using namespace std;
```

Utiliser différents espaces de noms permet, lors d'un projet en équipe ou de l'utilisation de bibliothèque externes, de créer plusieurs entités portant le même nom.

Créer un espace de noms

Pour déclarer vos fonctions, variables et classes dans votre espace de noms, placez-les dans un bloc de ce type :

```
namespace identifiant
{
    // Déclarations ici ...
}
```

Exemple :

```
namespace exemple
{
    int suivant(int n) {
        return n+1;
    }
}
```

Utiliser un espace de noms

Pour utiliser les entités d'un espace de noms en dehors de celui-ci (voire dans un autre espace de noms), vous pouvez utiliser le nom précédé de l'espace de nom :

```
int a = exemple::suivant(5);
```

ou utiliser la déclaration `using` :

```
using exemple::suivant;
...
int a = suivant(5);
```

ou encore utiliser la directive `using`

```
using namespace exemple;
...
int a = suivant(5);
```

Recommandation: Il est préférable de ne pas utiliser les directives `using` dans un fichier entête, excepté dans les fonctions *inline*. En effet, la directive ayant pour effet d'ouvrir l'espace de noms à l'espace global, toute unité qui inclurait ce fichier entête se verrait automatiquement ouvrir l'espace en question pouvant provoquer rapidement des collisions si cette même unité incluait d'autres fichiers entêtes utilisant aussi la directive `using` sur d'autres espaces de noms.

Alias d'espace de noms

Une autre manière très pratique d'utiliser un espace de noms surtout lorsque celui-ci est imbriqué dans d'autres espaces de noms sur plusieurs niveaux est l'emploi d'alias. Un alias permet de ramener une portée à un seul nom.

Exemple:

```
namespace global {
namespace exemple {
    int suivant(int n);
} // namespace exemple
} // namespace global
...
namespace ge = global::exemple; // alias
int a = ge::suivant(5);
```

En plusieurs fois

Vous pouvez ajouter des déclarations à un espace de noms existant, par exemple lorsque vous utilisez un fichier d'en-tête et un fichier d'implémentation :

■ Fichier **exemple.h**

```
namespace exemple
{
    int suivant(int n);
}
```

■ Fichier **exemple.cpp**

```
#include "exemple.h"

namespace exemple
{
    int suivant(int n) {
        return n+1;
    }
}
```

Les exceptions

Exceptions en C++

Une exception est l'interruption de l'exécution du programme à la suite d'un événement particulier. Le but des exceptions est de réaliser des traitements spécifiques aux événements qui en sont la cause. Ces traitements peuvent rétablir le programme dans son mode de fonctionnement normal, auquel cas son exécution reprend. Il se peut aussi que le programme se termine, si aucun traitement n'est approprié.

Le C++ supporte les exceptions logicielles, dont le but est de gérer les erreurs qui surviennent lors de l'exécution des programmes. Lorsqu'une telle erreur survient, le programme doit lancer une exception. L'exécution normale du programme s'arrête dès que l'exception est lancée, et le contrôle est passé à un gestionnaire d'exception. Lorsqu'un gestionnaire d'exception s'exécute, on dit qu'il a attrapé l'exception.

Les exceptions permettent une gestion simplifiée des erreurs, parce qu'elles en reportent le traitement. Le code peut alors être écrit sans se soucier des cas particuliers, ce qui le simplifie grandement. Les cas particuliers sont traités dans les gestionnaires d'exception.

En général, une fonction qui détecte une erreur d'exécution ne peut pas se terminer normalement. Comme son traitement n'a pas pu se dérouler normalement, il est probable que la fonction qui l'a appelée considère elle aussi qu'une erreur a eu lieu et termine son exécution. L'erreur remonte ainsi la liste des appelants de la fonction qui a généré l'erreur. Ce processus continue, de fonction en fonction, jusqu'à ce que l'erreur soit complètement gérée ou jusqu'à ce que le programme se termine (ce cas survient lorsque la fonction principale ne peut pas gérer l'erreur)^[2].

Lancer une exception

Lancer une exception consiste à retourner une erreur sous la forme d'une valeur (message, code, objet exception) dont le type peut être quelconque (`int`, `char*`, `MyExceptionClass`, ...).

Le lancement se fait par l'instruction `throw` :

```
throw 0;
```

Attraper une exception

Pour attraper une exception, il faut qu'un bloc encadre l'instruction directement, ou indirectement, dans la fonction même ou dans la fonction appelante, ou à un niveau supérieur. Dans le cas contraire, le système récupère l'exception et met fin au programme.

Les instructions `try` et `catch` sont utilisées pour attraper les exceptions.

```
try {
    ... // code lançant une exception (appel de fonction, ...)
}
catch (int code)
{
    cerr << "Exception " << code << endl;
}
```

Exemple de fonction lançant une exception :

```
int division(int a, int b)
{
    if (b == 0)
    {
        throw 0; // division par zéro;
    }
    else return a / b;
}

void main()
{
    try {
        cout << "1/0 = " << division(1, 0) << endl;
    }
}
```

```

catch (int code)
{
    cerr << "Exception " << code << endl;
}
}

```

Attraper toutes les exceptions

Spécifier les points de suspension dans la clause `catch` permet d'attraper tous les autres types d'exception :

```

void main()
{
    try {
        cout << "1/0 = " << division(1, 0) << endl;
    }
    catch (int code)
    {
        cerr << "Exception " << code << endl;
    }
    catch (...)
    {
        cerr << "Exception inconnue !!!" << endl;
    }
}

```

Déclaration des exceptions lancées

La déclaration d'une fonction lançant un nombre limité de type d'exception, telle que la fonction `division` de l'exemple précédent, peut être suivie d'une liste de ces types d'exceptions dans une clause `throw` :

```

int division(int a, int b) throw(int)
{
    if (b == 0) throw 0; // division par zéro;
    else return a / b;
}

```

Par défaut, la fonction peut lancer n'importe quel type d'exception. La déclaration de la fonction `division` sans clause `throw` est donc équivalent à la déclaration suivante :

```

int division(int a, int b) throw(...) // n'importe quel type d'exception
{
    if (b == 0)
        throw 0; // division par zéro;
    else
        return a / b;
}

```

Si une fonction ne lance aucune exception, on peut la déclarer avec une clause `throw` vide :

```

int addition(int a, int b) throw() // aucune exception
{
    return a+b;
}

```

Cette clause peut être présente dans les cas suivants :

- prototype de fonction
- implémentation de fonction
- pointeur de fonction

En revanche, il n'est pas possible de l'utiliser avec `typedef`.

Une déclaration avec clause `throw` limite donc les types d'exception que la fonction peut lancer. Toute tentative de lancer un autre type d'exception

est détectée à l'exécution et provoque l'appel de la fonction `std::terminate()`, puis l'arrêt immédiat du programme. Ce comportement est spécifique au C++, et amène souvent les développeurs à remplacer cette clause par une simple documentation.

De plus, l'utilisation de la clause `throw` reste rare car rendant les pointeurs de fonctions incompatibles s'ils ne possèdent pas une clause couvrant les types lancés par la fonction pointée.

Il est possible de fournir sa propre fonction `std::terminate()` appelée en cas d'exception non attrapée, grâce à `std::set_terminate()`.

```
#include <exception>
#include <iostream>

void maTerminate()
{
    std::cout << "Dans ce cas, c'est grave !" << std::endl;
}

int main()
{
    std::set_terminate(maTerminate); // On fournit sa propre fonction
    std::terminate();               // Et on termine le programme.
    return 0;
}
```

Les templates

Définitions des templates

Un template est un patron définissant un modèle de fonction ou de classe dont certaines parties sont des paramètres (type traité, taille maximale). Le patron de fonction sert à généraliser un même algorithme, une même méthode de traitement, à différents cas ou types de données.

Le mot clé `template` est suivi de la liste des paramètres du patron entre les signes `<` et `>`, suivi de la définition de la fonction ou de la classe.

L'instanciation d'un patron permet la création effective d'une fonction ou d'une classe.

Patron de fonction

Un patron de fonction est un modèle de fonction précédé du mot clé `template` et de la liste des paramètres du patron entre les signes `<` et `>`.

Exemple avec un patron de fonction calculant la valeur maximale d'un tableau de données (le type étant le paramètre du patron de fonction) :

```
template<class T>
T max(T array[], int length)
{
    T vmax = array[0];
    for (int i = 1; i < length; i++)
        if (array[i] > vmax)
            vmax = array[i];
    return vmax;
}
```

Le type `class` utilisé pour le paramètre `T` est en fait un type de données, pas forcément une classe. La norme a introduit plus tard le mot clé `typename` qui peut venir se substituer à `class` dans ce cas-là.

Exemple d'utilisation :

```
int values[]={ 100, 50, 35, 47, 92, 107, 84, 11 };
cout << max(values, 8);
```

Le compilateur crée une fonction `max` à partir du type des arguments, en remplaçant le paramètre `T` par le type `int` dans le cas précédent. Après remplacement le compilateur génère la fonction suivante :

```
int max(int array[], int length)
{
    int vmax = array[0];
    for (int i = 1; i < length; i++)
        if (array[i] > vmax)
            vmax = array[i];
    return vmax;
}
```

Si on utilise la fonction `max` avec différents types (`int`, `double`, ...), le compilateur générera autant de fonctions. Les patrons permettent donc d'éviter d'écrire plusieurs fonctions pour chaque type de donnée traité. Quand on sait que la duplication inutile de code est source d'erreur, on comprend l'intérêt de mettre en facteur plusieurs fonctions potentielles dans un même patron de fonction.

Patron de classe

La déclaration d'un patron de classe utilise une syntaxe similaire.

Exemple d'une classe gérant un tableau de données :

```
template<class T,int maxsize>
class CdataArray
```

```
{
    public:
        CDataArray();

        int getSize();
        T get(int index);
        bool add(T element); // true si l'élément a pu être ajouté

    private:
        T array[maxsize];
        int length;
};
```

L'implémentation du constructeur et des méthodes de la classe exige une syntaxe un peu plus complexe. Par exemple, pour le constructeur :

```
template<class T,int maxsize>
CDataArray<T,maxsize>::CDataArray()
: length(0) // Liste d'initialisation des données membres
{}
```

La première ligne rappelle qu'il s'agit d'un patron de classe. Le nom du constructeur est précédé du nom de la classe suivi des paramètres du patron de classe, afin que le compilateur détermine de quelle classe il s'agit. Car en pratique, on peut très bien combiner patron de classe et patron de méthode, par exemple.

Les méthodes sont implémentées de manière identique :

```
template<class T,int maxsize>
int CDataArray<T,maxsize>::getSize()
{
    return length;
}

template<class T,int maxsize>
T CDataArray<T,maxsize>::get(int index)
{
    return array[index];
}

template<class T,int maxsize>
bool CDataArray<T,maxsize>::add(T element)
{
    if (length>=maxsize) return false;
    array[length++]=element;
    return true;
}
```

Contrairement aux patrons de fonctions, l'instanciation d'un patron de classe exige la présence de la valeur des paramètres à la suite du nom de la classe.

Exemple :

```
CDataArray<int,100> listeNumeros;

listeNumeros.add(10);
listeNumeros.add(15);

cout << listeNumeros.getSize() << endl;
```

Le compilateur génère une classe pour chaque ensemble de valeurs de paramètres d'instanciation différent.

Voir aussi

- [Exercices](#)

Conventions d'appel

Convention d'appel

Une convention d'appel à une fonction définit la manière d'appeler une fonction. La manière varie selon le langage de programmation, le compilateur, et parfois l'architecture du processeur cible.

En C++ la convention d'appel définit les points suivants :

- l'ordre d'empilage des paramètres (à partir du premier, ou du dernier),
- passage de certains paramètres dans les registres,
- appel avec une adresse longue ou courte (selon la localisation en mémoire de la fonction),
- désempilage des paramètres effectué par la fonction appelée ou le code appelant,
- méthode de récupération de la valeur retournée.

Pour l'avant dernier point, le désempilage des paramètres est toujours effectué par le code appelant lorsque la fonction a un nombre variable d'arguments, car seul le code appelant connaît assurément le nombre de paramètres qu'il a empilé.

Conventions d'appels sous Windows

Attention : cette section sort du cadre du C++ standard et est spécifique à des développements sous Windows. Ces conventions d'appels n'existent pas sous UNIX et Linux.

Spécifier la convention d'appel

La convention d'appel est spécifiée juste avant le nom de la fonction. Parmi les conventions d'appel possibles :

- `__cdecl` : Convention d'appel par défaut du C et du C++ quand la fonction est définie en dehors de toute classe, ou comporte un nombre variable d'arguments,
- `__stdcall` : Convention d'appel utilisé par l'API Windows,
- `__fastcall` : Passage de certains paramètres dans les registres,
- `thiscall` : Convention d'appel utilisée implicitement par défaut par les fonctions définies dans une classe car il ajoute un paramètre supplémentaire : la référence à l'objet `this`. Ce mot clé ne peut être utilisé.
- ...

Différences entre les conventions d'appel

Convention	Paramètres dans les registres	Empilage des paramètres	Dépilage des paramètres
<code>__cdecl</code>	aucun	de droite à gauche	par le code appelant
<code>__stdcall</code>	aucun	de droite à gauche	par la fonction appelée
<code>__fastcall</code>	les 2 premiers arguments de taille inférieure à 32 bits	de droite à gauche	par la fonction appelée
<code>thiscall</code>	aucun	de droite à gauche	par la fonction appelée

Si la convention d'appel utilisée par le code appelant ne correspond pas à celui de la fonction appelée, la fonction ne comprend pas l'appel effectué et peut causer des dégâts.

Nommage des fonctions

Pour éviter des problèmes lors de l'édition de liens avec les bibliothèques externes (extension `.dll` sous Windows, `.so` sous Unix/Linux), le compilateur nomme les fonctions selon les critères suivants :

- Nom de la fonction,
- Type des arguments,
- Convention d'appel utilisée,

- Type des exceptions lancées,
- Nom de la classe où se situe la fonction.

Chaque compilateur utilise un nommage différent. Il est donc difficile d'utiliser une librairie qui n'a pas été compilée avec le même compilateur que l'application l'utilisant.

Liens

- anglais [Calling Convention \(http://en.wikipedia.org/wiki/Calling_convention\)](http://en.wikipedia.org/wiki/Calling_convention)
- anglais [Calling Conventions in Microsoft Visual C++ \(http://www.hackcraft.net/cpp/MSCallingConventions/\)](http://www.hackcraft.net/cpp/MSCallingConventions/)

Les fichiers

Les fichiers

Généralité sur les fichiers

- La règle générale pour créer un fichier est la suivante :
 - il faut l'ouvrir en écriture.
 - on écrit des données dans le fichier.
 - on ferme le fichier.
- Pour lire des données écrites dans un fichier :
 - on l'ouvre en lecture.
 - on lit les données en provenance du fichier.
 - on ferme le fichier.

Fichiers textes ou binaires

Il existe deux types de fichiers :

- les fichiers textes qui sont des fichiers lisibles par un simple éditeur de texte.
- les fichiers binaires dont les données correspondent en général à une copie bit à bit du contenu de la RAM. Ils ne sont pas lisibles avec un éditeur de texte.

cstdio ou fstream

Il existe principalement 2 bibliothèques standard pour écrire des fichiers :

- `cstdio` qui provient en fait du langage C.
- `fstream` qui est typiquement C++.

Utilisation de `cstdio`

La fonction `FILE * fopen(const char * filepath, char * mode)`

Cette fonction permet d'ouvrir un fichier en lecture ou en écriture. Le paramètre *filepath* est un tableau de char contenant le chemin du fichier sur lequel on souhaite travailler. Le paramètre *mode* indique le mode d'ouverture de *filepath* : lecture ou écriture, texte ou binaire.

Le mode peut avoir l'une des valeurs suivantes :

- `"r"` (*read*) : lecture,
- `"w"` (*write*) : écriture, fichier créé ou écrasé s'il existait déjà,
- `"a"` (*append*) : écriture en fin de fichier existant (**a**jjout de données).

Sur certaines plateformes (Windows par exemple), on peut y ajouter le type d'écriture (texte ou binaire), sur les autres (Linux par exemple) le type est ignoré :

- `"b"` : mode **b**inaire,
- `"t"` : mode **t**exte.

Enfin, on peut ajouter le signe `"+"` afin d'ouvrir le fichier en lecture et écriture à la fois.

Exemples :

- `"r+"` : ouverture en lecture et écriture,
- `"wb"` : ouverture en écriture binaire.

La fonction `fopen` retourne le pointeur `NULL` si l'ouverture du fichier a échoué. Dans le cas contraire, elle retourne un pointeur vers une structure `FILE`. Ce pointeur servira à écrire ou lire dans le fichier, ainsi qu'à le fermer.

La fonction fclose(FILE *)

Cette fonction permet de fermer un fichier, qu'il soit ouvert en lecture ou en écriture. On passe en paramètre à cette fonction le pointeur FILE * fourni par la fonction fopen(...).

Les fichiers binaires

■ La fonction fwrite(const void * buffer, int size,int nb, FILE * f) :

Cette fonction écrit nb éléments de size octets (soit nb*size octets) à partir du pointeur buffer (dans la RAM) vers le fichier f qui doit être ouvert en écriture. Il s'agit donc d'un transferts d'octets de la RAM dans un fichier.

■ La fonction fread(const void * buffer, int size,int nb, FILE * f) :

Cette fonction lit nb éléments de size octets (soit nb*size octets) à partir du fichier f (qui doit être ouvert en lecture) vers le pointeur buffer (dans la RAM). Il s'agit donc d'un transferts d'octets d'un fichier vers la RAM.

■ Exemple : écriture du fichier

```

1 #include <iostream>
2 #include<cstdio>
3 using namespace std;
4
5 int main (void)
6 {
7     FILE * f;
8     int a = 78, i, t1 [6];
9     double b = 9.87;
10    char c = 'W', t2 [10];
11
12    for(i = 0; i < 6; i++)
13        t1 [i] = 10000 + i;
14
15    cout << t2 << endl;
16    f = fopen ("toto.xyz", "wb");
17    if (f == NULL)
18        cout << "Impossible d'ouvrir le fichier en écriture !" << endl;
19    else
20        {
21        fwrite (&a,sizeof(int),1,f);
22        fwrite (&b,sizeof(double),1,f);
23        fwrite (&c,sizeof(char),1,f);
24        fwrite (t1,sizeof(int),6,f);
25        fwrite (t2,sizeof(char),10,f);
26        fclose (f);
27        }
28
29    return 0;
30 }
```

Dans ce programme, on ouvre le fichier binaire nommé toto.xyz en écriture. Si on a réussi à ouvrir le fichier, on y écrit un entier, un double, un char, puis un tableau de 6 entiers et finalement un tableau de 10 char.

On remarquera que pour écrire un entier il faut écrire &a pour obtenir un pointeur vers cet entier. Pour copier le tableau t1 on écrit juste t1 car t1 est déjà un pointeur vers le premier élément du tableau.

■ Exemple : lecture du fichier

```

1 #include <iostream>
2 #include<cstdio>
3 using namespace std;
4
5 int main (void)
6 {
7     FILE * f;
8     int a, t1 [6], i;
9     double b;
```

```

10 char c, t2[10];
11
12 f = fopen("toto.xyz", "rb");
13 if (f == NULL)
14     cout << "Impossible d'ouvrir le fichier en lecture !" << endl;
15 else
16     {
17         fread (&a,sizeof(int),1,f);
18         fread (&b,sizeof(double),1,f);
19         fread (&c,sizeof(char),1,f);
20         fread (t1,sizeof(int),6,f);
21         fread (t2,sizeof(char),10,f);
22         fclose (f);
23     }
24
25 cout << "a=" << a << endl
26     << "b=" << b << endl
27     << "c=" << c << endl;
28 for (i = 0; i < 6; i++)
29     cout << t1 [i] << endl;
30 cout << t2 << endl;
31
32 return 0;
33 }

```

Dans ce programme, on ouvre le fichier binaire nommé toto.xyz en lecture seule. Si on a réussi à ouvrir le fichier, on lit un entier, un double un char, puis un tableau de 6 entiers et finalement un tableau de 10 char.

Les fichiers textes

- la fonction `fprintf(FILE *f, const char * format,...)`

La fonction `fprintf` permet d'écrire en mode texte dans un fichier différentes données. On n'oubliera pas de laisser un espace entre les données pour pouvoir les relire (ou un passage à la ligne).

Le paramètre format permet de spécifier la nature des données et des caractéristiques sur leur écriture dans le fichier (le nombre de caractères pare exemple).

Exemples de formats :

"%d" ==> indique un entier

"%lf" ==> indique un double

"%3.7lf" ==> indique un double avec 3 chiffres avant la virgule et 7 après.

"%s" ==> indique une chaîne de caractères sans espace.

- la fonction `fscanf(FILE * f, const char * format,...)`

La fonction `fscanf` permet de lire les données à partir d'un fichier texte en utilisant le format de données indiqué (qui est identique à `printf`).

- **Exemple : écriture du fichier**

```

1 #include <iostream>
2 #include<cstdio>
3 using namespace std;
4
5 int main (void)
6 {
7     FILE * f;
8     int a = 78, t1 [6], i;
9     double b = 9.87;
10    char c = 'W', t2 [10];
11
12    for (i = 0; i < 6; i++)
13        t1 [i] = 10000+i;
14    strcpy (t2, "AZERTYUIO");
15    f = fopen ("toto.txt", "wt");

```

```

;16  if (f == NULL)
;17      cout << "Impossible d'ouvrir le fichier en écriture !" << endl;
;18  else
;19      {
;20      fprintf (f, "%d %lf %c ", a, b, c);
;21      for (i=0;i<6;i++)
;22          fprintf (f, "%d ", t1[i]);
;23      fprintf (f, "%s ", t2);
;24      fclose (f);
;25      }
;26
;27  return 0;
;28 }

```

Dans ce programme, on ouvre le fichier texte nommé toto.txt en écriture. Si on a réussi à ouvrir le fichier, on y écrit un entier, un double, un char, puis un tableau de 6 entiers et finalement une chaîne de caractères sans espace contenu dans un tableau de char.

■ Exemple : lecture du fichier

```

1  #include <cstdlib>
2  #include <iostream>
3  #include<cstdio>
4  using namespace std;
5
6  int main (void)
7  {
8      FILE * f;
9      double b;
;10     int a, t1 [6], i;
;11     char c, t2 [10];
;12
;13     f = fopen ("toto.txt", "rt");
;14     if (f == NULL)
;15         cout << "Impossible d'ouvrir le fichier en lecture !" << endl;
;16     else
;17         {
;18         fscanf (f, "%d %lf %c ", &a, &b, &c);
;19         for (i = 0; i < 6; i++)
;20             fscanf (f, "%d ", &t1 [i]);
;21         fscanf (f, "%s ", t2);
;22         fclose (f);
;23
;24         cout << "a=" << a << endl
;25             << "b=" << b << endl
;26             << "c=" << c << endl;
;27         for (i = 0; i < 6; i++)
;28             cout << t1 [i] << endl;
;29         cout << t2 << endl;
;30         }
;31     return 0;
;32 }

```

Dans ce programme, on ouvre le fichier binaire nommé toto.txt en lecture. Si on a réussi à ouvrir le fichier, on lit un entier, un double, un char, puis un tableau de 6 entiers et finalement une chaîne de caractères.

Encodage

L'encodage des fichiers peut être précisé en paramètre^[3] :

```

;FILE *fp = fopen("newfile.txt", "rt+", ccs=UNICODE");
;FILE *fp = fopen("newfile.txt", "rt+", ccs=UTF-8");
;FILE *fp = fopen("newfile.txt", "rt+", ccs=UTF-16LE");

```

Utilisation de fstream

Les fichiers textes

■ La classe ofstream :

Il s'agit d'un fichier ouvert en écriture : pour créer un tel fichier il suffit d'appeler le constructeur qui a en paramètre le nom du fichier : par exemple `ofstream f("toto.txt");`.

Pour savoir si le fichier a bien été ouvert en écriture la méthode `is_open()` renvoie `true` si le fichier est effectivement ouvert.

Pour écrire dans le fichier on utilise l'opérateur `<<` sans oublier d'écrire des séparateurs dans le fichier texte.

■ La classe ifstream :

Il s'agit d'un fichier ouvert en lecture : pour créer un tel fichier il suffit d'appeler le constructeur qui a en paramètre le nom du fichier : par exemple `ifstream f("toto.txt");`.

Pour savoir si le fichier a bien été ouvert en lecture la méthode `is_open()` renvoie `true` si le fichier est effectivement ouvert.

Pour lire dans le fichier on utilise l'opérateur `>>`.

■ Exemple : écriture d'un fichier texte

```

1 #include <iostream>
2 #include <fstream>
3 #include <string>
4
5 using namespace std;
6
7 int main(void)
8 {
9     int a = 78, t1 [6], i;
10    double b = 9.87;
11    char c = 'W';
12    string s;
13
14    for (i = 0; i < 6; i++)
15        t1 [i] = 10000+i;
16    s = "AZERTYUIO";
17    ofstream f ("toto.txt");
18
19    if (!f.is_open())
20        cout << "Impossible d'ouvrir le fichier en écriture !" << endl;
21    else
22        {
23            f << a << " " << b << " " << c << endl;
24            for (i = 0; i < 6; i++)
25                f << t1 [i] << " ";
26            f << s;
27        }
28    f.close();
29    return 0;
30 }
```

■ Exemple : lecture d'un fichier texte

```

1 #include <iostream>
2 #include <fstream>
3 #include <string>
4
5 using namespace std;
6
7 int main (void)
8 {
9     int a, t1 [6], i;
10    double b;
11    char c;
12    string s;
13    ifstream f ("toto.txt");
14
```

```

;15  if (!f.is_open())
;16      cout << "Impossible d'ouvrir le fichier en lecture !" << endl;
;17  else
;18      {
;19      f >> a >> b >> c;
;20      for (i = 0; i < 6; i++)
;21          f >> t1 [i];
;22      f >> s;
;23      }
;24  f.close();
;25  cout << "a=" << a << endl
;26      << "b=" << b <<endl
;27      << "c=" << c <<endl;
;28  for (i = 0; i < 6; i++)
;29      cout << t1 [i] << endl;
;30  cout << s << endl;
;31
;32  return 0;
;33 }

```

Les fichiers binaires

■ La classe ofstream :

Pour ouvrir en écriture un fichier binaire, il suffit d'appeler le constructeur qui a en paramètre le nom du fichier et le mode d'ouverture et fixer ce deuxième paramètre à ios::out | ios::binary: par exemple `ofstream f("toto.xyz",ios::out | ios::binary);`.

Pour savoir si le fichier a bien été ouvert en écriture la méthode `is_open()` renvoie true si le fichier est effectivement ouvert.

Pour écrire dans le fichier on utilise la méthode `write((char *)buffer , int nb)` pour écrire nb octets dans ce fichier.

■ La classe ifstream :

Pour ouvrir en lecture un fichier binaire, il suffit d'appeler le constructeur qui a en paramètre le nom du fichier et le mode d'ouverture et fixer ce deuxième paramètre à ios::in | ios::binary: par exemple `ifstream f("toto.xyz",ios::in | ios::binary);`.

Pour savoir si le fichier a bien été ouvert en écriture la méthode `is_open()` renvoie true si le fichier est effectivement ouvert.

Pour lire dans le fichier on utilise la méthode `read((char *)buffer , int nb)` pour lire nb octets de ce fichier.

■ Écriture d'un fichier binaire

```

1  #include <iostream>
2  #include <fstream>
3
4  using namespace std;
5
6  int main (void)
7  {
8      int a = 78, t1 [6], i;
9      double b = 9.87;
10     char c = 'W';
11
12     for (i = 0; i < 6; i++)
13         t1 [i] = 10000+i;
14
15     ofstream f ("toto.xyz", ios::out | ios::binary);
16
17     if(!f.is_open())
18         cout << "Impossible d'ouvrir le fichier en écriture !" << endl;
19     else
20     {
21         f.write ((char *)&a, sizeof(int));
22         f.write ((char *)&b, sizeof(double));
23         f.write ((char *)&c, sizeof(char));
24         for (i = 0; i < 6; i++)
25             f.write ((char *)&t1[i], sizeof(int));
26     }
27     f.close();

```

```
};28
};29 return 0;
};30 }
```

■ lecture d'un fichier binaire

```
1 #include <iostream>
2 #include<fstream>
3 using namespace std;
4
5 int main (void)
6 {
7     int a, t1 [6], i;
8     double b;
9     char c;
10    string s;
11    ifstream f ("toto.xyz", ios::in | ios::binary);
12
13    if (!f.is_open())
14        cout << "Impossible d'ouvrir le fichier en lecture !" << endl;
15    else
16        {
17        f.read ((char *)&a, sizeof(int));
18        f.read ((char *)&b, sizeof(double));
19        f.read ((char *)&c, sizeof(char));
20        for (i = 0; i < 6; i++)
21            f.read ((char *)&t1[i], sizeof(int));
22        }
23    f.close();
24
25    cout << "a=" << a << endl
26         << "b=" << b << endl
27         << "c=" << c << endl;
28    for (i = 0; i < 6; i++)
29        cout << t1 [i] << endl;
30
31    return 0;
32 }
```


La librairie standard

La STL

La STL (Standard Template Library) a été mise au point par Alexander Stepanov et Meng Lee. La STL a été proposée au comité ISO de standardisation du C++ qui l'a acceptée en juillet 1994. Les résultats des travaux de recherche ont été publiés officiellement dans un rapport technique en novembre 1995. Ces travaux de recherche ont été une avancée majeure pour le C++, qui était aussi à l'époque le seul langage capable d'offrir les mécanismes de programmation générique nécessaires à la mise au point de cette bibliothèque. Elle a d'ailleurs influencé les autres parties de la future bibliothèque du C++ (notamment la future classe string) et aussi l'évolution du langage.

La STL est axée autour de trois grands thèmes :

- Les **conteneurs** : ce sont les structures de données classiques de l'algorithmique, à savoir les tableaux à accès direct, les listes chaînées, les piles, les files, les ensembles, les dictionnaires. Dans sa version initiale, elle ne contient pas les tables de hachage, qui ne seront d'ailleurs pas présents dans ISO C++98.
- Les **algorithmes** : ce sont les algorithmes classiques de l'algorithmique, essentiellement les algorithmes de tri et de recherche
- Les **itérateurs** : c'est une généralisation du concept de pointeur. D'ailleurs un pointeur est un itérateur particulier. Les itérateurs ont l'avantage de pouvoir parcourir un conteneur sans que ce parcours ne fasse partie de l'état interne du conteneur.

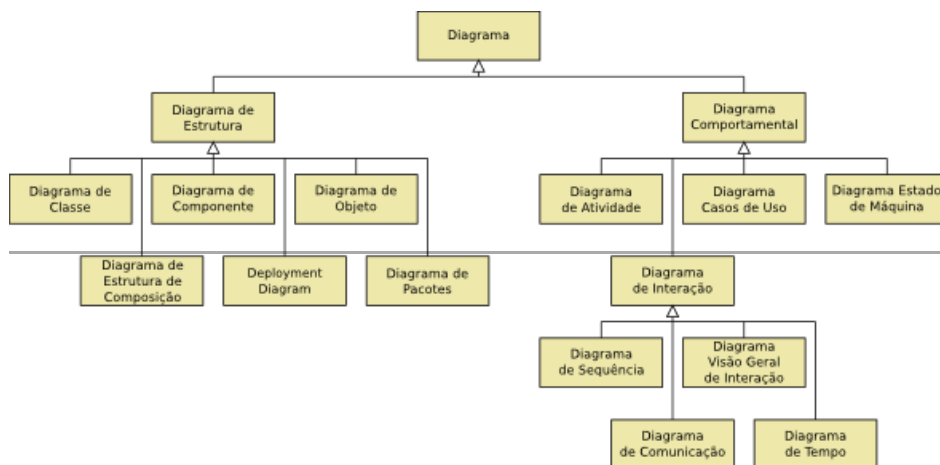
La force de la STL réside dans le fait de pouvoir offrir des conteneurs pouvant accueillir n'importe quel type moyennant quelques contraintes sémantiques qui peuvent varier selon le type conteneur et surtout dans le fait que les algorithmes sont complètement découplés des conteneurs (même si certains conteneurs offrent parfois leur propre algorithme pour des besoins d'optimisation et de performances. Par exemple, le conteneur *list* a une méthode *sort*).

Les conteneurs

En c++, les conteneurs sont des classes offrant au programmeur une implémentation permettant de gérer des collections dynamiques d'objets du même type (on parle de conteneurs homogènes), c'est-à-dire pour lesquels le nombre d'objets contenus peut varier à l'exécution. Les conteneurs sont implémentés dans la bibliothèque standard en tant que *modèles de classe* (templates), ce qui permet de les utiliser avec n'importe quel type d'objets, standard ou défini par le programmeur (à l'exception de **bitset**).

De plus les conteneurs sont conçus de manière à être compatible avec les algorithmes de la bibliothèque standard.

Comme toute classe standard, il faut les inclure avec un en-tête (header) spécifique, qui est souvent le nom de la classe avec des < > autour. Exemple : pour utiliser la classe vector il faut préciser dans le programme un `#include <vector>` .



Pour leur présentation il est d'usage de les séparer en trois catégories :

Les conteneurs séquentiels

Ils offrent des fonctionnalités suffisantes de base pour gérer simplement des collections d'objets. Tous permettent une insertion et un accès à tous les éléments par des itérateurs.

Il s'agit de **vector**, **deque**, **list** et **bitset**.

vector est un tableau dynamique où il est particulièrement aisé d'accéder directement aux divers éléments par un index, et d'en ajouter ou en retirer à la fin. A la manière des tableaux de type C, l'espace mémoire alloué pour un objet de type `vector` est toujours continu, ce qui permet des algorithmes rapides d'accès aux divers éléments. A noter qu'il existe une version spéciale de `vector`, `vector<bool>`, spécialement conçue pour économiser de la mémoire

deque (de "Double ended **queue**", file à double entrée) ressemble beaucoup à `vector`, mis à part qu'il est tout aussi efficace d'y insérer ou de supprimer des éléments au début de la liste. De plus, les éléments contenus ne sont pas forcément stockés de manière contigüe en mémoire, ce qui permet d'éviter de lourdes réallocations lors de changements importants dans le conteneur s'il est grand.

list est une liste doublement chaînée. L'insertion et la suppression d'élément ou de groupes continus d'éléments y est efficace partout dans la liste, mais il n'est pas possible d'accéder directement (par un index) aux différents éléments. On est forcé de les parcourir avec les itérateurs. De la même manière que `deque`, une liste chaînée ne voit pas ses différents éléments stockés en mémoire continument.

bitset est un cas particulier de conteneur. On peut l'assimiler à un tableau classique de `bool`. Le chiffre binaire étant la plus petite unité de mémoire possible, alors que le type de base du c++ le plus petit (`char`) est de taille 8 bits (habituellement), ces tableaux sont donc particulièrement optimisés pour stocker et manipuler une grande quantité d'informations qui peuvent être traduites par 1 ou quelques bits seulement.

Les conteneurs associatifs

Les conteneurs associatifs supposent l'utilisation d'une "clé de recherche" (un numéro d'identification, ou des chaînes classées par ordre alphabétique par exemple) et implémentent des fonctions efficaces de tri et de recherche. Ils sont habituellement triés continuellement (lors de chaque insertion d'éléments).

Il s'agit de **set**, **multiset**, **map**, **multimap**

set ne contient que des objets du type clé de recherche tandis que **map** contient des clés associées avec un autre objet de type différent (exemple : Les numéros de téléphone dans un annuaire sont triés selon le nom de l'abonné correspondant au numéro). Toutes les clés sont supposées uniques.

Les conteneurs **multiset** et **multimap** sont les versions correspondantes qui autorisent la présence de plusieurs clés identiques (avec des algorithmes correspondants nécessairement un peu moins performants).

Les adaptateurs de conteneurs

Les adaptateurs de conteneurs ne s'utilisent qu'en complément à un des conteneurs précédents, afin de lui ajouter des fonctionnalités plus précises.

Il s'agit de **stack**, **queue**, **priority_queue**

stack implémente une interface de pile (LIFO, Last In First Out : dernier arrivé, premier sorti).

queue implémente un interface de file d'attente (FIFO, First In First Out : premier arrivé premier sorti).

priority_queue implémente un interface de file d'attente où les éléments peuvent être comparés entre eux (par niveau de *priorité*), et sont classés dans la file suivant l'ordre spécifié. Ainsi elle permettent de traiter des données suivant des niveaux de priorité de manière efficace.

Les algorithmes

Les algorithmes de séquences non modifiants

- **for_each**
- **find**
- **find_if**
- **find_end**
- **find_first_of**
- **adjacent_find**
- **count**
- **count_if**
- **mismatch**
- **equal**

- **search**
- **search_n**

Les algorithmes de séquences modifiants

copies

- **copy**
- **copy_backward**

échanges

- **swap**
- **swap_ranges**
- **iter_swap**

transformations

- **transform**

remplacements

- **replace**
- **replace_if**
- **replace_copy**
- **replace_copy_if**

remplissages

- **fill**
- **fill_n**

générations

- **generate**
- **generate_n**

suppressions

- **remove**
- **remove_if**
- **remove_copy**
- **remove_copy_if**

éléments uniques

- **unique**
- **unique_copy**

ordre inverse

- **reverse**
- **reverse_copy**

rotations

- **rotate**

- **rotate_copy**

permutations aléatoires

- **random_shuffle**

répartitions

- **partition**
- **stable_partition**

Les algorithmes de tri et les opérations apparentées

tris

- **sort**
- **stable_sort**
- **partial_sort**
- **partial_sort_copy**
- **nth_element**

recherches dichotomiques

- **lower_bound**
- **upper_bound**
- **equal_range**
- **binary_search**

fusions

- **merge**
- **inplace_merge**

opérations d'ensemble

- **includes**
- **set_union**
- **set_intersection**
- **set_difference**
- **set_symmetric_difference**

opérations de tas

- **push_heap**
- **pop_heap**
- **maxmake_heap**
- **sort_heap**

minimum et maximum

- **min**
- **max**
- **min_element**
- **max_element**
- **lexicographical_compare**

permutations

- `next_permutation`
- `prev_permutation`

Les chaînes de caractères

La classe string

- Il s'agit d'une classe standard qui permet de représenter une chaîne de caractères.
- Pour l'utiliser, il faut rajouter `#include <string>`
- Cette classe encapsule des données pour pouvoir effectuer toutes les opérations de base sur les chaînes.
- Ces opérations sont assez complexes notamment la gestion de la mémoire : l'encapsulation permet de masquer à l'utilisateur de la classe toutes les difficultés techniques.

Différentes opérations sur la classe string

- **Déclaration et initialisation** : `string s1; string s2= "BONJOUR";`
- **Affichage et saisie** : `cout<<s2; cin>>s1;`
- **Concaténation** : `string s3=s2+s1;`

Exemple 1 : la classe string

Fichier main.cpp

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 int main (void)
7 {
8     string s1, s2, s3;
9
10    cout << "Tapez une chaîne : "; cin >> s1;
11    cout << "Tapez une chaîne : "; cin >> s2;
12    s3 = s1 + s2;
13    cout << "Voici la concaténation des 2 chaînes : " << endl;
14    cout << s3 << endl;
15    return 0;
16 }
```

Explications

- Dans cet exemple, nous étudions l'utilisation de la classe string.
- On peut saisir le contenu d'une chaîne en utilisant cin.
- On peut concaténer 2 chaînes grâce à l'opérateur +.
- On peut afficher une chaîne grâce à cout.
- Dans cet exemple, on demande à l'utilisateur de saisir 2 chaînes de caractères s1 et s2 et on affiche s3 la concaténation de s1 et de s2.

Exécution

Lorsqu'on exécute ce programme, il s'affiche à l'écran :

Tapez une chaîne : **AZERTY**

Tapez une chaîne : **QSDFGH**

Voici la concaténation des deux chaînes :

AZERTYQSDFGH

Séparateurs

- Par défaut, lorsqu'on saisit une chaîne de caractères en utilisant cin, le séparateur est l'espace : cela empêche de saisir une chaîne de caractères comportant un espace.
- La fonction `getline(iostream &,string)` permet de saisir une chaîne de caractères en utilisant le passage à la ligne comme séparateur : notre chaîne de caractères peut alors comporter des espaces.

Exemple 2 : string avec des espaces

```

1 #include <iostream>
2 using namespace std;
3 #include<string>
4
5 int main (void)
6 {
7     string s1, s2, s3;
8
9     cout << "Tapez une chaîne : "; getline (cin, s1);
10    cout << "Tapez une chaîne : "; getline (cin, s2);
11    s3 = s1 + s2;
12    cout << "Voici la concaténation des 2 chaînes :" << endl;
13    cout << s3 << endl;
14    return 0;
15 }
```

Explications

- Dans cet exemple, la chaîne de caractères `s1` est saisie grâce à l'instruction `getline(cin,s1)` : cela permet de saisir au clavier la chaîne `s1`, la fin de la chaîne est alors indiquée lorsqu'on tape sur ENTREE.
- On saisit ensuite la chaîne `s2` et on affiche la concaténation des deux chaînes.

Exécution

Lorsqu'on exécute ce programme, il s'affiche à l'écran :

Tapez une chaîne : **AZ ERTY**

Tapez une chaîne : **QS DFGH**

Voici la concaténation des deux chaînes :

AZ ERTYQS DFGH

Analyse de chaînes

- **Nombre de caractères d'une chaîne** : `size()` est une méthode de la classe `string` qui renvoie le nombre de caractères utiles.
- **Récupération du i-ième caractère** : la méthode `const char at(int i)` permet de récupérer le `i+1`ème caractère. (`0 = 1er`)

Exemple 3 : analyse de chaînes

```

1 #include <iostream>
2 #include<string>
3
4 using namespace std;
5
6 int main (void)
7 {
8     string s= "BONJOUR";
9     int taille = s.size ();
10
11    cout << "La chaîne comporte " << taille << " caractères." << endl;
12
13    for (int i = 0 ; i < taille ; i++)
```

```

;14     cout << "caractère " << i << " = " << s.at(i) << endl;
;15     return 0;
;16 }
    
```

Explications

- La méthode `size()` sur un `string` permet de connaître la taille d'une chaîne de caractères.
- Si `i` est un entier `s.at(i)` permet de connaître le $(i+1)$ -ième caractère de la chaîne (`s.at(0)` étant le premier caractère).
- Dans ce programme, on initialise la chaîne `s` à "BONJOUR" : on affiche ensuite la taille de la chaîne et on affiche ensuite un à un chaque caractère.

Exécution

Lorsqu'on exécute ce programme, il s'affiche à l'écran :

```

La chaîne comporte 7 caractères
;caractère 0 = B
;caractère 1 = O
;caractère 2 = N
;caractère 3 = J
;caractère 4 = O
;caractère 5 = U
;caractère 6 = R
    
```

Compatibilité avec les `char *` et les tableaux de `char`

- **Transformation de chaîne de type C en `string`** : on peut utiliser le constructeur `string(char *)` ou l'affectation grâce au symbole `=` d'un `char *` vers une `string`.
- **Transformation d'une `string` en chaîne de type C** : il suffit d'utiliser la méthode : `c_str()` qui renvoie un `char *` qui est une chaîne de type C.

Exemple 4 : compatibilité avec les tableaux de `char` et les `char *`

```

1 #include <iostream>
2 using namespace std;
3 #include<string>
4
5 int main (void)
6 {
7     string s1, s2;
8     char c1 []= "BONJOUR";
9     const char * c2;
;10
;11     s1 = c1;
;12     cout << s1 << endl;
;13     s2 = "AU REVOIR";
;14     c2 = s2.c_str();
;15     cout << c2 << endl;
;16     return 0;
;17 }
    
```

Explications

- Dans cet exemple, `c1` est un tableau de 8 `char` contenant la chaîne "BONJOUR " (n'oubliez pas le caractère de fin de chaîne `'\0'`).
- Le pointeur `c2` est un pointeur vers un tableau non modifiable de `char`.
- Les variables `s1` et `s2` sont des `string`.

- On peut affecter directement `s1=c1` : le tableau de char sera transformé en string.

Dans `c2`, on peut récupérer une chaîne « de type C » identique à notre string en écrivant `c2=s2.c_str()`.

- On peut transformer aisément une string en tableau de char et inversement.

Exécution

Lorsqu'on exécute ce programme, il s'affiche à l'écran :

```
BONJOUR
AU REVOIR
```

Transformation d'une chaîne en int ou double

- Pour transformer une chaîne en double ou en int, il faut transformer la chaîne en flot de sortie caractères : il s'agit d'un `istringstream`.
- Ensuite, nous pourrons lire ce flot de caractères en utilisant les opérateurs usuels `>>`.
- La méthode `eof()` sur un `istringstream` permet de savoir si la fin de la chaîne a été atteinte.
- Chaque lecture sur ce flot grâce à l'opérateur `>>` renvoie un booléen qui nous indique d'éventuelles erreurs.

Exemple 5 : transformation de string en int

```
1 #include <iostream>
2 #include <sstream>
3 #include <string>
4 using namespace std;
5 int main (void)
6 {
7     string s;
8     cout << "Tapez une chaîne : "; getline (cin, s);
9     istringstream istr(s);
10    int i;
11
12    if (istr >> i) cout << "VOUS AVEZ TAPE L'ENTIER " << i << endl;
13    else cout << "VALEUR INCORRECTE" << endl;
14    return 0;
15 }
```

Explications

- Dans cet exemple, `s` est une chaîne : on saisit une chaîne au clavier en utilisant `getline(cin,s)`.
- On crée ensuite un `istringstream` appelé `istr` et construit à partir de `s`.
- On peut lire un entier `i` à partir de `istr` en utilisant : `istr>>i` .
- (`istr>>i`) renvoie `true` si un entier valide a pu être lu et renvoie `false` sinon.
- De la même manière, on pourrait lire des données d'autres types, double par exemple.

Exécution 1

Lorsqu'on exécute ce programme, il s'affiche à l'écran :

```
Tapez une chaîne : 12345
VOUS AVEZ TAPE L'ENTIER 12345
```

Exécution 2

Lorsqu'on exécute ce programme, il s'affiche à l'écran :

```
Tapez une chaîne : BJ9UYU
VALEUR INCORRECTE
```

Les chaînes de caractères de type C

Le header `cstring` définit un certain nombre de fonction permettant la manipulation de chaînes de caractères de type C.

`memcpy`

`memmove`

`strcpy`

`strncpy`

`strcat`

`strncat`

`memcmp`

`strcmp`

`strcoll`

`strncmp`

`strxfrm`

`strcspn`

`strspn`

`strtok`

`memset`

`strerror`

`strlen`

Lorsqu'une chaîne est enregistrée avec des unités de code de huit bits (soit un octet) la fonction `strlen` donne le nombre d'octets de la chaîne.

Si jamais le codage de caractère utilisé est l'un de ses anciens codage de caractères où le nombre d'octet correspond au nombre de caractères, ce nombre est alors aussi le nombre de caractères.

```
int strlen(char *str)
{
    int i = 0;
    while(str[i] != '\0')
        i++;
    return (i);
}
```

`memchr`

```
char *pch;
int pospex;

pch = (char*) memchr (Trame, '!', strlen(Trame));
if (pch!=NULL)
{
    pospex = pch - Trame;
}
```

memchr(void* p, int c, size_t n)

memchr scrute les n premiers octets de p à la recherche du caractère c.

si succès, renvoie un pointeur sur la première occurrence de c dans p, sinon, renvoie NULL.

inline char* strchr(char* s1, int _n)**inline char* strpbrk(char* s1, const char* s2)****inline char* strrchr(char* s1, int n)****inline char* strstr(char* s1, const char* s2)**

Voir aussi

- [Libstdcpp](#)

Expressions rationnelles

Les expressions rationnelles peuvent être analysées et testées via un débogueur en ligne comme <https://regex101.com/>.

Expressions rationnelles courantes

Caractère	Type	Explication
.	Point	n'importe quel caractère
[...]	crochets	<u>classe de caractères</u> : tous les caractères énumérés dans la classe
[^...]	crochets et circonflexe	<u>classe complémentée</u> : tous les caractères sauf ceux énumérés
^	circonflexe	marque le début de la chaîne, la ligne...
\$	dollar	marque la fin d'une chaîne, ligne...
	barre verticale	alternative - ou reconnaît l'un ou l'autre
(...)	parenthèses	<u>groupe de capture</u> : utilisée pour limiter la portée d'un masque ou de l'alternative
*	astérisque	0, 1 ou plusieurs occurrences
+	le plus	1 ou plusieurs occurrences
?	interrogation	0 ou 1 occurrence
{...}	accolades	comptage : détermine un nombre de caractères remplissant les critères qu'il suit (ex : a{2} deux occurrences de "a", a{1, 10} entre une et dix)

Classes de caractères POSIX^[4]

Classe	Signification
[:alpha:]	n'importe quelle lettre
[:digit:]	n'importe quel chiffre
[:xdigit:]	caractères hexadécimaux
[:alnum:]	n'importe quelle lettre ou chiffre
[:space:]	n'importe quel espace blanc
[:punct:]	n'importe quel signe de ponctuation
[:lower:]	n'importe quelle lettre en minuscule
[:upper:]	n'importe quelle lettre capitale
[:blank:]	espace ou tabulation
[:graph:]	caractères affichables et imprimables
[:cntrl:]	caractères d'échappement
[:print:]	caractères imprimables exceptés ceux de contrôle

Expressions rationnelles Unicode^[5]

Expression	Signification
\A	Début de chaîne
\b	Caractère de début ou fin de mot
\d	Chiffre
\D	Non chiffre
\n	Fin de ligne
\s	Caractères espace
\S	Non caractères espace
\t	Tabulation
\w	Caractère alphanumérique : lettre, chiffre ou underscore
\W	Caractère qui n'est pas lettre, chiffre ou underscore
\X	Caractère Unicode
\Z	Fin de chaîne

Constructeurs spéciaux : Ces fonctions précèdent l'expression à laquelle elles s'appliquent, et le tout doit être placé entre parenthèses.

- `?:` : groupe non capturant. Ignorer le groupe de capture lors de la numérotation des backreferences. Exemple : `((?:sous-chaîne_non_renvoyée|autre).)`.
- `?>` : groupe non capturant indépendant.
- `?<=` : positive lookbehind.
- `?<!` : negative lookbehind.
- `?=` : positive lookahead.
- `?!` : negative lookahead. Exclusion d'une chaîne. Il faut toujours la faire suivre d'un point. Exemples :
 - `((?!sous-chaîne_exclue).)`
 - `<(?!body).*>` : pour avoir toutes les balises HTML sauf "body".
 - `début((?!\mot_exclu).)*fin[6]` : pour rechercher tout ce qui ne contient pas un mot entre deux autres.

Remarques :

- Les caractères de débuts et fin de chaînes (^ et \$) ne fonctionnent pas dans [] où ils ont un autre rôle.
- Les opérateurs * et + sont toujours avides, pour qu'ils laissent la priorité il faut leur apposer un ? à leur suite.

Références

1. name=Un caractère doit juste être codé sur au moins 8 bits.
2. <http://www.developpez.com/c/megacours/c3770.html>
3. <https://msdn.microsoft.com/fr-fr/library/yeby3zcb.aspx>
4. <https://www.regular-expressions.info/posixbrackets.html>
5. <http://www.regular-expressions.info/unicode.html>
6. <https://www.regextester.com/15>

Interfaces graphiques

Il est possible en C++ de réaliser des interfaces graphiques portables c'est-à-dire fonctionnant à la fois sous Linux, sous Windows et sous Mac OS.

On parle alors de *toolkits*.

Citons par exemple :

- wxWidgets, disponible sous la licence "wxWindows license" qui autorise la création de programmes commerciaux sans pour autant payer de licence, et bien évidemment de créer des logiciels libres ;
- Qt, disponible de manière gratuite pour l'utilisation dans des projets openSources mais son utilisation dans des projets commerciaux nécessite l'achat d'une licence ;
- gtkmm (<http://www.gtkmm.org/>), tout comme wxWidgets, il est possible de concevoir des applications propriétaires ou openSources ;
- GTK+, il est possible de concevoir des applications open source ou propriétaires avec GTK+, il a été initialement créé pour les besoins de GIMP puis a été réutilisé pour réaliser le gestionnaire de fenêtre GNOME.

Ces *toolkits* comportent des composants graphiques « standards » ayant la même apparence quelle que soit la plateforme de développement. Pour que cela soit portable, un *toolkit* existe en différentes versions. Chaque version supporte un système d'exploitation différent et implémente les composants pour celui-ci.

Bibliographie et liens

Bibliographie

- (français) *Bjarne Stroustrup* - **Le langage C++** - Éditions Pearson Education - 2004 - 1100 pages - (ISBN 2-744-07003-3)
- (français) *Ivor Horton* - **Visual C++ 6** - Éditions Eyrolles - 1999 - 1290 pages - (ISBN 2-212-09043-9)

Liens sur les autres projets

- (français) C++ sur Wikipédia

Liens Externes

- (français) Cours de C/C++ (<http://www.developpez.com/c/megacours/book1.html>)
- (anglais) C/C++ Reference (<http://www.cppreference.com/>)
- (français) Apprenez à programmer en C++ (<http://www.siteduzero.com/tuto-3-5395-0-apprenez-a-programmer-en-c.html>)

Dessiner des formes : les bibliothèques graphiques

Le C++ peut être utilisé pour la création de jeux ou plus généralement d'interface graphiques, qui nécessitent de pouvoir dessiner et créer des formes, entre autres.

Cette partie traite de la programmation graphique *basique* du C++ en 2D. Pour ce faire, il sera utilisé la bibliothèque graphique SFML.

Téléchargement de la SFML

- Allez sur <http://www.sfml-dev.org/download/sfml/2.4.0/index-fr.php> et téléchargez la version qui correspond à votre configuration. Le site de la SFML est très bien détaillé et apporte beaucoup d'informations sur quelle version télécharger.
- Enregistrer où vous voulez le dossier, le principal étant de choisir un répertoire dont on se souvient facilement.

Mise en place de la SFML

Pour utiliser la SFML, il faut ajouter à votre projet un chemin d'inclusion qui va jusqu'au dossier *include* de votre dossier SFML : par exemple `"C:\SFML-2.4.0\include"` (sur Windows).

Puis il faut joindre les fichiers *.hpp* correspondant aux besoins du programme. La plupart du temps, `#include <SFML/Graphics.hpp>` suffit.

Pour la compilation, il faut d'abord compiler le fichier source principal du programme, par habitude `"main.cpp"` : `"g++ -c main.cpp -I<chemin/du/dossier/include/SFML>".` Cela génère un fichier `"main.o"`. Puis il faut lier le fichier compilé aux bibliothèques SFML : `"main.o -o sfml.exe -L<chemin/du/dossier/lib/SFML> -lsfml-graphics -lsfml-window -lsfml-system"`. Enfin, il faut ajouter les fichiers *.dll* du dossier `"bin"` du dossier SFML dans le répertoire d'exécution du programme compilé.

On peut se référer aux pages d'aides du site de la SFML pour effectuer les actions décrites ci-avant : <http://www.sfml-dev.org/tutorials/2.4/index-fr.php> --> Catégorie « Démarrer ». En ce qui concerne *Visual Studio Code*, on peut configurer ainsi le fichier `"tasks.json"` :

```
{
  // See https://go.microsoft.com/fwlink/?LinkId=733558
  // for the documentation about the tasks.json format
  "version": "0.1.0",
  "command": "g++",
  "isShellCommand": true,
  "suppressTaskName": true,
  "tasks": [
    {
      "taskName": "Compilation",
      "isBuildCommand": true,
      "args": ["-c", "${workspaceRoot}\\main.cpp", "-IC:\\SFML-2.4.0\\include"]
    },
    {
      "taskName": "Liaison du fichier compilé aux bibliothèques SFML",
      "args": ["${workspaceRoot}\\main.o", "-o", "sfml.exe", "-LC:\\SFML-2.4.0\\lib", "-lsfml-graphics", "-lsfml-window", "-lsfml-system"]
    }
  ],
  "showOutput": "always"
}
```

Utilisation de la SFML



Cette section est vide, pas assez détaillée ou incomplète.

Opérateurs

On peut facilement utiliser des opérateurs sur une ou des classes. Prenons un exemple très simple: La classe Q (nombre rationnel)!

```

#include <iostream>
#include <cstdlib>

class Q{
public:
    int num; // stocke le numérateur
    int den; // stocke le dénominateur
    Q(int a=0,int b=1); // le constructeur de la classe Q (le nombre 0 par défaut)
    Q operator+(Q a); // Définit l'opérateur +
    void show(); // Affiche le nombre rationnel
};

Q::Q(int a,int b){ // Le constructeur
    num=a;
    den=b;
}

Q Q::operator+(Q a){
    Q t; // crée un nouveau nombre rationnel
    t.num=num*a.den+den*a.num; // stocke le numérateur des nombres additionnés
    t.den=den*a.den; // stocke le dénominateur des nombres additionnés.
    return t; // retourne le résultat de l'addition
}

void Q::show(){
    std::cout << num << " / " << den << " = " << (float)(num)/(float)(den) << "\n";
}

int main(){
    Q a(3,4),b(1,2); // définit deux nouveaux nombres rationnels a et b
    a=a+b; // additionne a et b est stocke le résultat sous a.
    a.show(); // affiche a
    return 0;
}

```

Voilà ! Il est en effet bien pratique d'avoir une classe Q qui permet de stocker des nombres rationnels, mais elle n'est pas très utile, si l'on ne peut pas faire d'opération avec ces objets. L'exemple ci-dessus montre comment il est possible d'utiliser l'opérateur + entre deux objets de la classe Q (nombre rationnel).

Héritage

L'héritage est le fait de faire descendre une classe d'une autre. Par exemple, la classe rectangle descend de la classe polygone, le rectangle est un polygone et hérite donc de tous les attributs de polygone, mais il a quelques propriétés de plus.

Pour déclarer que la classe rectangle descend de polygone, on écrit :

```
class rectangle : public polygone {  
    //définition de la classe rectangle  
};
```

Polymorphisme

Polymorphisme

Le polymorphisme est une fonctionnalité de l'héritage : la capacité d'appeler une méthode en fonction du type réel d'un objet (sa classe).

Remarque : ne pas confondre avec la surcharge, où les fonctions peuvent avoir un même nom, mais différents types de paramètres.

Exemple

Soit une classe A implémentée de la manière suivante :

```
class A
{
public:
    int compare(A autre); // compare deux objets A
    static void sort(A** tableau_de_a,int nombre); // tri
};
```

La méthode statique `sort` effectue le tri du tableau en appelant la méthode `compare`.

Soit une classe B héritant de A, ayant un critère supplémentaire de comparaison. Il faut donc une nouvelle version de la méthode de comparaison :

```
class B : public A
{
public:
    int compare(A autre); // compare deux objets A
};
```

Problème

Si on a un tableau de pointeur sur des objets B, la méthode `compare` appelée par la fonction de tri est celle de la classe A, car la fonction a pour paramètre un tableau de pointeur sur des objets A. Ce comportement n'est pas polymorphique.


Solution

Pour que la méthode appelée soit celle de la classe B, il faut qu'elle soit déclarée virtuelle (`virtual`) dans la classe de base :

```
class A
{
public:
    virtual int compare(A autre); // compare deux objets A
    static void sort(A** tableau_de_a,int nombre); // tri
};
```

La fonction de tri appellera alors la méthode selon le type réel des objets, même en utilisant un pointeur sur des objets A.

En fait, le compilateur construit une table des fonctions virtuelles pour les classes A et B et les autres sous-classes, et chaque objet créé possède un pointeur vers cette table.



Vous avez la permission de copier, distribuer et/ou modifier ce document selon les termes de la **licence de documentation libre GNU**, version 1.2 ou plus récente publiée par la Free Software Foundation ; sans sections inaltérables, sans texte de première page de couverture et sans texte de dernière page

de couverture.

Récupérée de « https://fr.wikibooks.org/w/index.php?title=Programmation_C%2B%2B/Version_imprimable&oldid=568695 »

La dernière modification de cette page a été faite le 28 août 2017 à 00:15.

Les textes sont disponibles sous [licence Creative Commons attribution partage à l'identique](#) ; d'autres termes peuvent s'appliquer.

Voyez les [termes d'utilisation](#) pour plus de détails.