



Beste de savoir

La programmation en C++ moderne

31 août 2020

Table des matières

I. Remerciements	14
II. Le début du voyage	16
II.1. Le C++, qu'est-ce que c'est ?	18
II.1.1. Petite mise au point	18
II.1.1.1. Développer, un métier à part entière	18
II.1.1.2. Votre part du travail	18
II.1.1.3. Les mathématiques, indispensables?	19
II.1.1.4. L'anglais, indispensable?	19
II.1.2. Tour d'horizon du C++	19
II.1.2.1. L'histoire du C++	19
II.1.2.2. Pourquoi apprendre le C++?	20
II.1.3. La documentation	21
II.1.3.1. En résumé	21
II.2. Le minimum pour commencer	22
II.2.1. Des outils en ligne	22
II.2.2. Des outils plus poussés	23
II.2.2.1. Visual Studio Community	24
II.2.2.2. Qt Creator	28
II.2.2.3. Pour les autres outils	30
II.2.3. Un mot concernant Windows	31
II.2.4. Un mot concernant GNU/Linux	31
II.2.4.1. Un éditeur de texte	32
II.2.4.2. Un compilateur	32
II.2.4.3. En résumé	32
II.3. Rencontre avec le C++	33
II.3.1. Compilons notre premier programme	33
II.3.2. Démystification du code	34
II.3.2.1. Inclure des fonctionnalités déjà codées	34
II.3.2.2. Le point d'entrée	34
II.3.2.3. Voici mes instructions...	35
II.3.3. Les commentaires	36
II.3.3.1. La syntaxe	37
II.3.3.2. En résumé	37
Contenu masqué	38

II.4. Une super mémoire	39
II.4.1. Les littéraux	39
II.4.1.1. Les caractères	39
II.4.1.2. Les nombres	42
II.4.2. Les variables	44
II.4.2.1. Comment créer des variables en C++?	44
II.4.2.2. Un peu de constance, voyons!	49
II.4.2.3. Manipulation de variables	50
II.4.3. Qu'en déduisez-vous?	52
II.4.3.1. Avec <code>const</code>	53
II.4.3.2. Le cas de <code>std::string</code>	54
II.4.3.3. Les avantages	54
II.4.4. Les entrées	55
II.4.4.1. Gestion des erreurs	55
II.4.4.2. En résumé	57
Contenu masqué	57
II.5. Le conditionnel conjugué en C++	60
II.5.1. Les booléens	60
II.5.2. <code>if</code> —Si, et seulement si...	62
II.5.2.1. À portée	63
II.5.3. <code>else</code> —Sinon...	64
II.5.4. <code>else if</code> —La combinaison des deux précédents	66
II.5.5. Conditions imbriquées	67
II.5.6. [T.P] Gérer les erreurs d'entrée—Partie I	69
II.5.7. La logique booléenne	70
II.5.7.1. AND—Tester si deux conditions sont vraies	70
II.5.7.2. OR—Tester si au moins une condition est vraie	71
II.5.7.3. NOT—Tester la négation	72
II.5.8. Tester plusieurs expressions	73
II.5.8.1. Évaluation en court-circuit	75
II.5.9. Exercices	76
II.5.9.1. Une pseudo-horloge	76
II.5.9.2. Score	76
II.5.9.3. XOR—Le OU exclusif	77
II.5.9.4. En résumé	77
Contenu masqué	77
II.6. Des boucles qui se répètent, répètent, répètent...	84
II.6.1. <code>while</code> —Tant que...	84
II.6.2. Exercices	87
II.6.2.1. Une laverie	87
II.6.2.2. PGCD	87
II.6.2.3. Somme de nombres de 1 à n	87
II.6.3. <code>do while</code> —Répéter ... tant que	88
II.6.4. [T.P] Gérer les erreurs d'entrée—Partie II	89
II.6.5. <code>for</code> —Une boucle condensée	89
II.6.6. Boucles imbriquées	91

II.6.7.	Convention de nommage	92
II.6.8.	Contrôler plus finement l'exécution de la boucle	92
II.6.8.1.	<code>break</code> —Je m'arrête là	92
II.6.8.2.	<code>continue</code> —Saute ton tour!	94
II.6.8.3.	En résumé	94
	Contenu masqué	95
II.7.	Au tableau !	101
II.7.1.	Un tableau c'est quoi?	101
II.7.2.	<code>std::vector</code> —Mais quel dynamisme!	101
II.7.2.1.	Déclarer un <code>std::vector</code>	101
II.7.2.2.	Manipuler les éléments d'un <code>std::vector</code>	103
II.8.	Accès aux éléments	104
II.8.1.	Exercices	112
II.8.1.1.	Calcul de moyenne	112
II.8.1.2.	Minimum et maximum	112
II.8.1.3.	Séparer les pairs des impairs	113
II.8.1.4.	Compter les occurrences d'une valeur	113
II.8.2.	<code>std::array</code> —Le tableau statique	113
II.8.2.1.	Déclarer un <code>std::array</code>	113
II.8.2.2.	Remplir un tableau	114
II.8.2.3.	Accéder aux éléments	115
II.8.2.4.	Connaître la taille	116
II.8.2.5.	Vérifier si le tableau est vide	116
II.8.3.	<code>std::string</code> —Un type qui cache bien son jeu	117
II.8.3.1.	Connaître la taille d'une chaîne	117
II.8.3.2.	Accéder à un caractère	118
II.8.3.3.	Premier et dernier caractère	118
II.8.3.4.	Vérifier qu'une chaîne est vide	119
II.8.3.5.	Ajouter ou supprimer un caractère à la fin	119
II.8.3.6.	Supprimer tous les caractères	120
II.8.3.7.	Boucler sur une chaîne	120
II.8.3.8.	En résumé	121
	Contenu masqué	121
II.9.	Déployons la toute puissance des conteneurs	126
II.9.1.	Le socle de base: les itérateurs	126
II.9.1.1.	Déclaration d'un itérateur	126
II.9.1.2.	Début et fin d'un conteneur	127
II.9.1.3.	Accéder à l'élément pointé	127
II.9.1.4.	Se déplacer dans une collection	128
II.10.	Déplacement vers l'élément suivant	129
II.10.0.1.	<code>const</code> et les itérateurs	131
II.10.0.2.	Itérer depuis la fin	132
II.10.0.3.	Utilisation conjointe avec les conteneurs	134
II.10.1.	Des algorithmes à gogo!	135
II.10.1.1.	<code>std::count</code> —Compter les occurrences d'une valeur	135

II.10.1.2.	<code>std::find</code> —Trouver un certain élément	136
II.10.1.3.	<code>std::sort</code> —Trier une collection	138
II.10.1.4.	<code>std::reverse</code> —Inverser l'ordre des éléments d'une collection	139
II.10.1.5.	<code>std::remove</code> —Suppression d'éléments	140
II.10.1.6.	<code>std::search</code> —Rechercher un sous-ensemble dans un ensemble	140
II.10.1.7.	<code>std::equal</code> —Vérifier l'égalité de deux ensembles	141
II.10.1.8.	Et tant d'autres	142
II.10.2.	Personnalisation à la demande	142
II.10.2.1.	Le prédicat, à la base de la personnalisation des algorithmes	142
II.10.2.2.	Trier une liste dans l'ordre décroissant	142
II.10.2.3.	Somme et produit	144
II.10.2.4.	Prédicats pour caractères	144
II.10.3.	Exercices	146
II.10.3.1.	Palindrome	146
II.10.3.2.	<code>string_trim</code> —Suppression des espaces	146
II.10.3.3.	Couper une chaîne	147
II.10.3.4.	En résumé	147
	Contenu masqué	147
II.11.	Des flux dans tous les sens	152
II.11.1.	Avant-propos	152
II.11.1.1.	Prendrez-vous une extension?	152
II.11.1.2.	Vois sur ton chemin...	153
II.11.1.3.	Un mot sur Windows	153
II.11.1.4.	Pas d'interprétation, on est des brutes	154
II.11.2.	<code>std::ofstream</code> —Écrire dans un fichier	154
II.11.2.1.	Ouvrir le fichier	155
II.11.2.2.	Écriture	155
II.11.2.3.	Ouvrir sans effacer	158
II.11.3.	<code>std::ifstream</code> —Lire dans un fichier	159
II.11.3.1.	Ouvrir le fichier	159
II.11.3.2.	Lecture	159
II.11.3.3.	Tout lire	162
II.11.4.	Exercice	163
II.11.4.1.	Statistiques sur des fichiers	163
II.11.5.	Encore plus de flux!	163
II.11.5.1.	Un flux c'est quoi?	163
II.11.5.2.	Un <i>buffer</i> dites-vous?	164
II.11.5.3.	Les modificateurs de flux	165
II.11.5.4.	Même les chaînes y passent!	166
II.11.5.5.	En résumé	168
	Contenu masqué	169

III. On passe la deuxième !	171
III.1. Découpons du code— Les fonctions	173
III.1.1. Les éléments de base d'une fonction	173
III.1.1.1. Une fonction, c'est quoi?	173
III.1.1.2. Une fonction bien connue	175
III.1.1.3. Les composants d'une fonction	175
III.2. Schéma d'une fonction	176
III.2.1. Exercices	179
III.2.1.1. Afficher un rectangle	179
III.2.1.2. Distributeur d'argent	180
III.2.1.3. Parenthésage	180
III.2.2. Quelles sont vos références?	181
III.2.2.1. La problématique	181
III.2.2.2. Les références	182
III.2.2.3. Paramètres de fonctions	184
III.2.2.4. Valeur de retour	187
III.2.2.5. Un mot sur la déduction de type	188
III.2.3. Nos fonctions sont surchargées!	190
III.2.4. [T.P] Gérer les erreurs d'entrée—Partie III	191
III.2.5. Dessine-moi une fonction	192
III.2.5.1. Le problème	192
III.2.5.2. La solution	193
III.2.5.3. Utilisons les prototypes	194
III.2.5.4. En résumé	194
Contenu masqué	195
III.3. Erreur, erreur, erreur...	201
III.3.1. L'utilisateur est un idiot	201
III.3.2. À une condition... ou plusieurs	202
III.3.2.1. Contrats assurés par le compilateur	202
III.4. Le typage	203
III.4.0.1. Vérifier nous-mêmes	203
III.4.1. Le développeur est un idiot	203
III.4.1.1. Préconditions et assertions	206
III.4.2. Les tests unitaires à notre aide	207
III.4.2.1. Pourquoi tester?	207
III.4.2.2. Un test unitaire, qu'est-ce que c'est?	207
III.4.2.3. Écrire des tests unitaires	208
III.4.2.4. Les tests unitaires et les postconditions	209
III.4.3. [T.P] Gérer les erreurs d'entrée—Partie IV	209
III.4.4. L'exception à la règle	210
III.4.4.1. La problématique	210
III.4.4.2. C'est quoi une exception?	212
III.4.4.3. Lancement des exceptions dans 3, 2, 1...	212
III.4.4.4. Attends que je t'attrape!	214
III.4.4.5. Attrapez-les tous!	215

III.4.4.6. Un type pour les gouverner tous et dans le catch les lier	216
III.4.5. [T.P] Gérer les erreurs d'entrée—Partie V	218
III.4.5.1. En résumé	219
Contenu masqué	219
III.5. Des fonctions somme toute lambdas	224
III.5.1. Une lambda, c'est quoi?	224
III.5.1.1. Pourquoi a-t-on besoin des lambdas?	224
III.5.2. Syntaxe	225
III.5.2.1. Quelques exemples simples	225
III.5.3. Exercices	228
III.5.3.1. Vérifier si un nombre est négatif	228
III.5.3.2. Tri par valeur absolue	228
III.5.4. On va stocker ça où?	229
III.5.5. Paramètres génériques	230
III.5.6. [T.P] Gérer les erreurs d'entrée—Partie VI	231
III.5.7. [T.P] Gérer les erreurs d'entrée—Partie VII	232
III.5.8. Capture en cours...	233
III.5.8.1. Capture par valeur	233
III.5.8.2. Capture par référence	235
III.5.8.3. En résumé	236
Contenu masqué	236
III.6. Envoyez le générique!	241
III.6.1. Quel beau modèle!	241
III.6.1.1. La bibliothèque standard: une fourmilière de modèles	246
III.6.1.2. Un point de vocabulaire	247
III.6.2. [T.P] Gérer les erreurs d'entrée - Partie VIII	247
III.6.3. Instanciation explicite	248
III.6.4. Ce type là, c'est un cas!	253
III.6.4.1. En résumé	257
Contenu masqué	258
III.7. De nouvelles structures de données	261
III.7.1. struct—Un agrégat de données	261
III.7.1.1. Stockage d'informations personnelles ¹	261
III.7.1.2. Analyse de l'exercice	262
III.7.1.3. Introduction aux structures	262
III.7.1.4. Exercice	264
III.7.2. std::tuple—Une collection hétérogène	265
III.7.2.1. Déclaration	265
III.7.2.2. Accès aux éléments	266
III.7.2.3. std::tuple et fonctions	267
III.7.2.4. Équations horaires	269
III.7.3. std::unordered_map—Une table associative	271
III.7.3.1. Un dictionnaire de langue Zestienne	271
III.7.3.2. Toujours plus de clés	273
III.7.3.3. Cette clé est unique!	275
III.7.3.4. Cherchez un élément	277

III.7.3.5. Un exemple concret	279
III.7.4. std::unordered_set—Représenter un ensemble	280
III.7.4.1. Inscription sur Zeste de Savoir	280
III.7.5. Un peu d'ordre, voyons!	282
III.7.6. Une longue énumération	284
III.7.6.1. En résumé	286
Contenu masqué	286
III.8. Reprendrez-vous un peu de sucre syntaxique ?	290
III.8.1. Ce type est trop long!	290
III.8.2. Décomposons tout ça	290
III.8.2.1. En pleine décomposition	291
III.8.2.2. Et avant, on faisait comment?	293
III.8.3. La surcharge d'opérateurs	295
III.8.3.1. Le problème...	295
III.8.3.2. ...et la solution	297
III.8.3.3. Cas concret—Les fractions	297
III.9. Les opérateurs arithmétiques	298
III.9.0.1. Et la bibliothèque standard?	305
III.9.0.2. Exercice—Calculs avec des durées	306
III.9.0.3. Sur le bon usage de la surcharge	307
III.9.0.4. En résumé	307
Contenu masqué	307
III.10. [T.P] Un gestionnaire de discographie	311
III.10.1. L'énoncé	311
III.10.1.1. Ajout de morceaux	311
III.10.1.2. Affichage de la discographie	312
III.10.1.3. Enregistrement et chargement d'une discographie	314
III.10.1.4. Quitter le programme	314
III.10.1.5. Gestion des erreurs	314
III.10.1.6. Dernières remarques	315
III.10.2. Guide de résolution	315
III.10.2.1. Analyse des besoins	315
III.10.2.2. Attaquons le code!	316
III.10.3. Corrigé complet	319
III.10.4. Conclusion et pistes d'améliorations	320
III.10.4.1. En résumé	320
Contenu masqué	321
III.11. Découpons du code—Les fichiers	347
III.11.1. Le principe	347
III.11.1.1. Le fichier d'en-tête	347
III.11.1.2. Le fichier source	353
III.11.1.3. Créons un lien	353
III.11.2. La sécurité, c'est important	355
III.11.3. Découpons le TP!	356

III.11.4. Les avantages du découpage en fichiers	358
III.11.4.1. Une structure de projet plus visible	358
III.11.4.2. Une meilleure abstraction	358
III.11.4.3. Une meilleure modularité	359
III.11.5. Le cas des templates	359
III.11.5.1. En résumé	363
Contenu masqué	364
IV. Interlude - Être un développeur	386
IV.1. Avant-propos	388
IV.1.1. Ce que nous avons appris	388
IV.1.2. Différents paradigmes	389
IV.1.2.1. Le paradigme impératif	389
IV.1.2.2. Le paradigme fonctionnel	390
IV.1.2.3. Le paradigme générique	391
IV.1.2.4. La programmation par contrat	392
IV.1.3. Brancher le cerveau avant tout	393
IV.1.4. Savoir se débrouiller	394
IV.1.4.1. En résumé	394
IV.2. Mais où est la doc ?	396
IV.2.1. Lire une page de doc	396
IV.2.1.1. À l'arrivée	396
IV.2.1.2. <code>vector</code> — Retour sur le plus célèbre des conteneurs	397
IV.3. Présentation	398
IV.3.0.1. Les algorithmes	400
IV.3.0.2. Les chaînes de caractère au grand complet	401
IV.3.1. Exercices	402
IV.3.1.1. Remplacer une chaîne de caractère par une autre	402
IV.3.1.2. Norme d'un vecteur	402
IV.3.1.3. Nombres complexes	403
IV.3.1.4. Transformations	404
IV.3.2. Documenter son code avec Doxygen	404
IV.3.2.1. Installation des outils	404
IV.3.2.2. Écrire la documentation	408
IV.3.3. Quelques bonnes pratiques	415
IV.3.3.1. Ce qu'il faut documenter	415
IV.3.3.2. Les fichiers d'en-tête ou source?	415
IV.3.3.3. Commentaire vs documentation	416
IV.3.3.4. En résumé	418
Contenu masqué	419
IV.4. Compilation en cours...	423
IV.4.1. Le préprocesseur	423
IV.4.1.1. Inclure des fichiers	423
IV.4.1.2. Conditions	423

IV.4.1.3. <i>Debug ou release?</i>	424
IV.4.2. La compilation	426
IV.4.2.1. Les templates	426
IV.4.2.2. <code>constexpr</code>	427
IV.4.2.3. La compilation à proprement parler	429
IV.4.2.4. Une étape intermédiaire cachée	430
IV.4.2.5. Influencer sur la compilation	431
IV.4.3. Le linker	436
IV.4.3.1. Une question de symboles	437
IV.4.4. Schéma récapitulatif	440
IV.4.4.1. En résumé	441
IV.5. Chasse aux bugs!	443
IV.5.1. Le principe	443
IV.5.2. Un code d'exemple	444
IV.5.3. Visual Studio	444
IV.5.3.1. Interface	445
IV.5.3.2. Pas-à-pas	447
IV.5.3.3. Point d'arrêt conditionnel	448
IV.5.4. Qt Creator	449
IV.5.4.1. Interface	449
IV.5.4.2. Pas-à-pas	451
IV.5.4.3. Point d'arrêt conditionnel	452
IV.5.4.4. Aller plus loin	454
IV.5.5. En ligne de commande avec gdb	454
IV.5.5.1. Poser un point d'arrêt	454
IV.5.5.2. Supprimer des points d'arrêt	455
IV.5.5.3. Désactiver des points d'arrêt	455
IV.5.5.4. Afficher l'état d'une variable	456
IV.5.5.5. Pas-à-pas	456
IV.5.5.6. Conditions	457
IV.5.5.7. Aller plus loin	458
IV.5.5.8. En résumé	458
IV.6. Une foule de bibliothèques	459
IV.6.1. Quelles bibliothèques choisir?	459
IV.6.1.1. Boost	459
IV.6.1.2. SFML	460
IV.6.2. Généralités	460
IV.6.2.1. Statique ou dynamique?	460
IV.6.2.2. <i>Debug ou release?</i>	461
IV.6.3. Installer Boost	461
IV.6.3.1. GNU/Linux	461
IV.6.3.2. Windows	461
IV.6.3.3. Tester l'installation	464
IV.6.4. Installer SFML	466
IV.6.4.1. GNU/Linux	466
IV.6.4.2. Windows	467

IV.6.4.3.	Tester l'installation	470
IV.6.4.4.	En résumé	471
IV.7.	Améliorer ses projets	473
IV.7.1.	git—Sauvegarder et versionner son code	473
IV.7.1.1.	La problématique	473
IV.7.1.2.	Installation	473
IV.7.1.3.	Initialiser git	474
IV.7.1.4.	Créer un dépôt	474
IV.7.1.5.	Connaître l'état de ses fichiers	475
IV.7.1.6.	Ajouter un fichier	476
IV.7.1.7.	Valider les modifications	477
IV.7.1.8.	Voir l'historique	478
IV.7.1.9.	Bloquer certains fichiers	478
IV.7.1.10.	Aller plus loin	480
IV.7.1.11.	Visual Studio	480
IV.7.2.	GitHub—Partager son code	481
IV.7.2.1.	La problématique	481
IV.7.2.2.	Création d'un projet	482
IV.7.2.3.	Récupérer localement les modifications	484
IV.7.2.4.	Pousser nos modifications	485
IV.7.2.5.	Aller plus loin	486
IV.7.3.	CMake—Automatiser la compilation de nos programmes	486
IV.7.3.1.	La problématique	486
IV.7.3.2.	Un exemple simple	487
IV.7.3.3.	Lier des bibliothèques externes	491
IV.7.3.4.	Définir des variables au lancement	504
IV.7.3.5.	Aller plus loin	504
IV.7.4.	Aller plus loin	504
IV.7.4.1.	TravisCI—De l'intégration continue	504
IV.7.4.2.	CppCheck—Vérification du code	505
IV.7.4.3.	StackOverflow—La réponses à quasiment toutes les questions	506
IV.7.4.4.	En résumé	506
	Contenu masqué	507
V.	La Programmation Orientée Objet	515
V.1.	Premiers pas avec la POO	517
V.1.1.	Le principe: des objets bien serviables	517
V.1.1.1.	Le principe	517
V.2.	Penser en terme de données...	518
V.2.0.1.	Exemple concret	519
V.2.1.	Un peu de vocabulaire	519
V.2.2.	En C++, ça donne quoi?	520
V.2.2.1.	Penser services	520
V.2.2.2.	Penser tests	520
V.2.2.3.	Définition de la classe	521

V.3. Fonctions membres	522
V.3.1. Désolé, cet objet n'est pas modifiable	525
V.3.2. On ne fait pas d'exception	528
V.3.3. Découpage en fichiers	529
V.3.4. Exercices	532
V.3.4.1. Cercle	532
V.3.4.2. Informations personnelles	533
V.3.4.3. En résumé	533
Contenu masqué	533
V.4. Qui c'est qui va construire tout ça ?	537
V.4.1. Encapsulation et invariants	537
V.4.1.1. Cette classe, c'est <i>open bar</i>	537
V.4.1.2. Tout est question d'invariant	538
V.4.1.3. Encapsulons tout ça	538
V.4.1.4. Modifier sa visibilité	539
V.4.2. On cherche un constructeur	540
V.4.2.1. La syntaxe	542
V.4.3. Constructeur par défaut	543
V.4.3.1. Initialiser une instance par défaut a du sens	544
V.4.3.2. Initialiser une instance par défaut n'a pas de sens	545
V.4.4. Soyons explicites	546
V.4.5. En toute amitié	550
V.4.5.1. Attributs publics	553
V.4.5.2. Accesseurs	553
V.4.5.3. La troisième, c'est la bonne	554
V.4.6. Exercices	555
V.4.6.1. Un autre constructeur pour la classe Fraction	555
V.4.6.2. Pile	556
V.4.6.3. En résumé	557
Contenu masqué	558
V.5. Une classe de grande valeur	561
V.5.1. Une histoire de sémantique	561
V.5.2. La sémantique de valeur, c'est quoi?	562
V.5.3. Égalité	563
V.5.3.1. Libre ou membre?	563
V.5.3.2. Soyons intelligents, réutilisons	568
V.5.4. Le retour des opérateurs	570
V.5.4.1. Les opérateurs d'affectation intégrés	570
V.5.4.2. Du deux en un	572
V.5.4.3. Les opérateurs de manipulation des flux	573
V.5.4.4. Quelques bonnes pratiques	577
V.5.4.5. Les autres opérateurs	577
V.5.5. Copier des objets	577
V.5.5.1. Constructeur de copie	578
V.5.5.2. Opérateur d'affectation par copie	578
V.5.5.3. En résumé	579

V.6. [T.P] Entrons dans la matrice !	580
V.6.1. Qu'est-ce qu'une matrice?	580
V.6.1.1. Addition	580
V.6.1.2. Multiplication par un nombre	581
V.6.1.3. Transposition	581
V.6.1.4. Multiplication de deux matrices	581
V.6.1.5. D'autres propriétés	582
V.6.2. L'énoncé	582
V.6.2.1. Les services	582
V.6.2.2. Détails d'implémentation	582
V.6.2.3. Les tests	583
V.6.3. Correction détaillée	587
V.6.3.1. Les services	587
Contenu masqué	587
V.7. Constructeurs	596
V.7.0.1. Les détails d'implémentation	597
V.8. Fichiers d'en-tête standards	598
V.8.1. Aller plus loin	603
V.8.1.1. Le plus grand nombre	603
V.8.1.2. Des polynômes	604
V.8.1.3. En résumé	605
V.9. La sémantique de déplacement	606
V.10. Classes à sémantique d'entités	607
V.11. Le besoin de handles	608
V.12. [T.P] Le Javaquarium	609
V.13. Oh, le bel héritage	610
V.14. Les piliers de la POO	611
V.15. Classes templates	612
V.16. [T.P] Des polynômes génériques	613
VI. Annexes	614
VI.1. Index	615

Bienvenue dans ce cours sur l'apprentissage de la programmation avec le langage C++. Vous débutez complètement et n'avez qu'une vague idée de ce que peut bien être la programmation? On vous a dit d'en apprendre plus sur le C++? Alors **ce tutoriel est fait pour vous**.

Pour rendre ce cours **pédagogique**, nous alternerons entre théorie, explications et détails techniques d'un côté et exercices, travaux pratiques et questions-réponses de l'autre. L'accent

sera mis sur **la rigueur**, **la qualité** et les **bonnes pratiques**. Le rythme sera **progressif**, chaque concept étant introduit en temps voulu et en lien avec ce que nous aurons déjà vu.

i

Prérequis

Prérequis

Lire [cette introduction à la programmation](#) ↗ afin d’avoir des bases pour aborder l’apprentissage du C++.

Objectifs

Vous former à la programmation en vous donnant des bases solides sur lesquelles continuer.
Vous former au C++ moderne.

Première partie

Remerciements

I. Remerciements

Car ce cours n'aurait jamais pu voir le jour sans l'aide et le soutien de plusieurs personnes.

- Un merci particulier à @gbdivers pour son cours [Débuter en C++ moderne](#) qui aura été une source d'inspiration pour l'écriture du présent cours, à qui nous avons emprunté plusieurs passages très bien écrits et qui n'aura cessé de nous aider à améliorer ce cours par ses nombreuses remarques et suggestions.
- Tous les membres qui ont participé à [la bêta](#) et qui ont grandement aidé à améliorer la qualité de ce tutoriel, par leurs nombreuses remarques, suggestions et corrections. Toutes vos participations, petites et grandes, nous ont été extrêmement précieuses et utiles pour produire un cours meilleur.
- Toute [l'équipe](#) de Zeste de Savoir; en particulier, un grand merci à @Taurre et @Glordim qui se sont chargés de la validation.
- Et l'ensemble des lecteurs, pour avoir choisi ce cours.




À venir

La partie POO est toujours en rédaction. Les mises à jour viendront au fil de l'eau.

Deuxième partie

Le début du voyage

II. Le début du voyage

Vous y voilà. Grâce au tutoriel sur [les bases de la programmation](#)  , vous en savez un peu plus sur le merveilleux monde de la programmation. Maintenant, vous avez hâte d'appliquer tout ça concrètement.

Le but de cette première partie est de vous introduire aux concepts de bases de la programmation. Ce que vous y apprendrez vous permettra de faire des programmes assez utiles. Bien entendu, votre apprentissage ne s'arrêtera pas de si tôt. 🍊

II.1. Le C++, qu'est-ce que c'est ?

Voilà, vous êtes décidés, vous voulez apprendre à programmer à l'aide du langage C++. Mais vous devez toujours avoir **quelques questions** qui trottent dans votre esprit.

- Concrètement, C++, c'est quoi?
- Et puis pourquoi commencer par C++?
- Que vais-je savoir faire en sortant de ce tutoriel?
- Il y a-t-il des conditions, des prérequis, pour pouvoir suivre ce cours?

II.1.1. Petite mise au point

J'ai indiqué en introduction que le seul prérequis était de lire [l'introduction à la programmation](#) et c'est vrai. Mais si peu de connaissances sont nécessaires pour comprendre tout ce qui sera dit, cela ne veut pas dire pour autant que ce sera du tout cuit, prémâché, directement dans la bouche. C'est pour cela que je veux prendre quelques minutes pour clarifier ceci.

II.1.1.1. Développer, un métier à part entière

Eh oui! Créer des logiciels, **c'est un métier**. D'ailleurs, il y a même des écoles et des études spécialisées pour ça. C'est que développer, ce n'est pas simplement écrire du code. Il y a aussi des phases de réflexion, de conceptions, d'écriture, de validation, de tests, de réécriture d'anciennes portions, etc.

Par ailleurs, même si cela n'est pas directement lié à la programmation, peut également s'ajouter à cela: la gestion de base de données, l'usage du réseau, le management, la gestion de projet, etc. En bref, **être développeur c'est beaucoup de compétences différentes dans des domaines variés**.

La cruelle vérité qui se cache derrière tout ça, c'est que **ce cours ne fera pas de vous des experts, ni des développeurs professionnels**. Par contre, une fois fini, vous aurez des bases solides pour continuer votre apprentissage. La route du savoir est infinie.

II.1.1.2. Votre part du travail

Le cours est écrit de façon à être le plus clair possible, sans vous noyer sous un flot de détails et d'explications, mais il arrivera parfois que vous ne compreniez pas un morceau de code ou une explication. C'est tout à fait normal, ça fait partie de l'apprentissage. Reprenez le cours à tête reposée, aidez-vous de schémas ou de dessins, demandez de l'aide sur [les forums](#), et vous ne resterez jamais bloqués longtemps. 🍊

II. Le début du voyage

Par contre, il faut être prêt à **fournir des efforts** de votre côté. Cela signifie ne pas se ruer sur les forums au bout de quelques minutes, sans même avoir essayé de trouver une solution.

II.1.1.3. Les mathématiques, indispensables ?

Une des grandes appréhensions, qui revient régulièrement dans la bouche des débutants, est de savoir si les mathématiques sont un prérequis à l'apprentissage de la programmation. La réponse est **non**. Il est tout à fait possible d'**apprendre à programmer tout en ayant un faible niveau en mathématique**. Ce cours ne demande aucune connaissance en mathématiques plus poussées que les opérations de base et, quelques fois, l'utilisation de sinus et cosinus, en guise d'exemples.

Bien sûr, certains aspects de la programmation, comme la sécurité, la cryptographie ou les applications scientifiques vont demander un bagage mathématique solide. Mais cela ne rentre pas dans le cadre de ce cours.




II.1.1.4. L'anglais, indispensable ?

À strictement parler, dans le cadre de ce cours, **pas besoin de savoir parler anglais**. Même si nous examinerons régulièrement de la documentation écrite dans la langue de Shakespeare, je serai là pour donner des explications et éclaircissements. Et quand bien même il y aurait quelque chose sur lequel vous butez, vous pouvez vous servir d'un traducteur automatique.

D'un point de vue plus général, si vous souhaitez continuer dans l'informatique et la programmation, il sera très difficile d'échapper à l'anglais. Beaucoup de cours, de documents, de forums sont en anglais, ou en font un usage massif. L'anglais est tout simplement **la langue de l'informatique**. Après, sachez que l'anglais informatique est **simple à comprendre**, car souvent écrit par des gens dont l'anglais n'est pas la langue maternelle. Ainsi, inutile d'être bilingue, un bon dictionnaire ou un traducteur vous aideront.

II.1.2. Tour d'horizon du C++

II.1.2.1. L'histoire du C++

Faisons un plongeon dans l'histoire et revenons dans les années 1970. À cette époque, [Dennis Ritchie](#) , programmeur aux laboratoires AT&T aux États-Unis, invente [le langage C](#) , conçu pour programmer le système d'exploitation UNIX. Ce langage devint très populaire à tel point qu'il est encore beaucoup utilisé aujourd'hui, dans l'écriture de Linux par exemple. Puis un peu plus tard, au début des années 1980, [Bjarne Stroustrup](#) , lui aussi développeur aux laboratoires AT&T, décida de prendre le langage C comme base et de lui ajouter des fonctionnalités issues d'un autre langage appelé Simula. Ce langage devint alors le **C with classes**.

Finalement, en 1983, son créateur, estimant que le nom de son langage était trop réducteur aux vues de tous les ajouts faits par rapport au C, décida de le renommer **C++**. Mais l'histoire ne s'arrête pas là. Au contraire, le C++ continue d'évoluer à tel point qu'on décide au début des années 1990 de **le normaliser**, c'est-à-dire d'en établir les règles officielles. Ce travail de longue

II. Le début du voyage

haleine s'acheva en 1998; cette version est ainsi souvent nommée C++98. Ensuite, en 2003, des corrections ont été apportées et l'on obtint C++03.

Puis de nouveau un chantier titanesque est mis en place pour améliorer encore plus le C++, ce qui aboutit 8 ans plus tard, en 2011, à la sortie de C++11, jugée par beaucoup de développeurs comme étant la renaissance du C++. Ensuite, de nouvelles corrections et quelques ajustements ont été apportés pour donner C++14. Enfin, à l'heure actuelle, la norme C++17, nouvelle version majeure et apportant tout un lot de choses intéressantes, est sortie en fin d'année et C++20 est déjà en chantier.



Point vocabulaire

Beaucoup de programmeurs utilisent le terme «**C++ historique**» pour désigner les normes C++98 et C++03 et le terme «**C++ moderne**» pour parler de C++11 et au-delà.

II.1.2.2. Pourquoi apprendre le C++ ?

- Sa **popularité**: le C++ est un langage qui est utilisé dans de nombreux projets important (citons [Libre Office](#) , [7-zip](#) ou encore [KDE](#)). Il est au programme de beaucoup de formations informatiques. Il possède une communauté très importante, beaucoup de documentation et d'aide, surtout sur l'internet anglophone.
- Sa **rapidité**: C++ offre un grand contrôle sur la rapidité des programmes. C'est cette caractéristique qui fait de lui un des langages de choix pour les programmes scientifiques, par exemple.
- Sa **facilité d'apprentissage**: depuis sa version de 2011, C++ est beaucoup plus facile à apprendre que par le passé. Et ça tombe bien, c'est sur cette version et les suivantes que va se baser ce cours.
- Son **ancienneté**: C++ est un langage ancien d'un point de vue informatique (30 ans, c'est énorme), ce qui donne une certaine garantie de maturité, de stabilité et de pérennité (il ne disparaîtra pas dans quelques années).
- Son **évolution**: C++11 est un véritable renouveau de C++, qui le rend plus facile à utiliser et plus puissant dans les fonctionnalités qu'il offre aux développeurs. C++14 et C++17 améliorent encore la chose.
- Il est **multi-paradigme**: il n'impose pas une façon unique de concevoir et découper ses programmes mais laisse le développeur libre de ses choix, contrairement à d'autres langages comme [Java](#) ou [Haskell](#) .

Bien entendu, tout n'est pas parfait et C++ a aussi ses défauts.

- Son **héritage du C**: C++ est un descendant du langage C, inventé dans les années 1970. Certains choix de conception, adaptés pour l'époque, sont plus problématiques aujourd'hui, et C++ les traîne avec lui.
- Sa **complexité**: il ne faut pas se le cacher, avoir une certaine maîtrise du C++ est très long et demandera des années d'expérience, notamment parce que certaines des fonctionnalités les plus puissantes du C++ requièrent de bien connaître les bases.
- Sa **bibliothèque standard**: bien qu'elle permette de faire beaucoup de choses (et d'ailleurs, nous n'aurons même pas le temps d'en faire un tour complet dans ce cours), elle n'offre pas de mécanisme natif pour manipuler des bases de données, faire des

II. Le début du voyage

programmes en fenêtres, jouer avec le réseau, etc. Par rapport à d'autres langages comme [Python](#) , C++ peut paraître plus «limité».

II.1.3. La documentation

En programmation, il y a un réflexe à adopter le plus rapidement possible: si on ne sait pas comment utiliser un outil, il faut aller consulter **la documentation** de l'outil concerné, et ce avant de demander de l'aide sur un forum par exemple. Voici un lien vers [une excellente documentation](#) C++. Elle est en anglais, mais pas de soucis, je suis là avec vous. Je vous donnerai des liens, vous expliquerai comment comprendre et exploiter les informations fournies pour que, par la suite, vous puissiez le faire vous-mêmes.

Il y a aussi un autre outil, très utile pour rechercher une information et que vous connaissez déjà, **les moteurs de recherches** (Google, Bing, DuckDuckGo, Qwant, etc). Sachez aussi que **les forums** sont une mine d'informations. Vous pouvez par exemple utiliser ceux de [Zeste de Savoir](#) , en n'oubliant bien évidemment pas d'utiliser le tag `[c++]` lors de la création d'un sujet.

Enfin, sachez qu'il existe une référence ultime appelée **la norme**, produit par un organisme de validation international appelé l'**ISO**, qui explique tous les détails et les règles du C++ mais qui est un document complexe et très largement hors de portée pour vous.

La dernière version de ce document est sortie en 2017 et explique les règles de fonctionnement du C++ dans sa version de 2017. Je le mentionne simplement pour que vous soyez au courant de son existence, sans être surpris si, lors de vos recherches sur Internet, des réponses mentionnent ou citent la norme.

II.1.3.1. En résumé

- La programmation est une activité parfois complexe, mais très enrichissante et accessible.
- Le cours se veut le plus accessible possible, mais vous devez jouer le jeu et faire des efforts de votre côté.
- Les mathématiques et l'anglais ne sont pas requis pour suivre ce cours, bien qu'un niveau acceptable en anglais soit extrêmement utile par la suite.
- Le C++ a une longue histoire, mais le C++ moderne a véritablement vu le jour en 2011, avec des améliorations et corrections apportées en 2014 et 2017.
- Le C++ possède des forces qui en font un des langages les plus utilisés sur la planète.
- En contrepartie, le C++ est un langage assez complexe qui demande de nombreuses années avant d'être «maîtrisé».
- La documentation nous aide à mieux comprendre le langage, son fonctionnement, etc. Elle est en anglais, mais illustrée par des exemples.

II.2. Le minimum pour commencer

Maintenant que nous sommes bien au fait de ce que nous allons apprendre, il est temps de se pencher sur quelques outils que nous utiliserons. Nous verrons tant les **outils en ligne** pour tester rapidement un morceau de code que les **outils de développement complets**, à installer sur votre ordinateur.

II.2.1. Des outils en ligne

Nous l'avons vu dans l'[introduction à la programmation](#) [↗](#), il existe des langages compilés et C++ en fait partie. Nous allons donc avoir besoin d'un **compilateur**. Mais des compilateurs C++, il en existe beaucoup. Et vous, lecteurs, n'avez pas tous les mêmes systèmes d'exploitation (Windows, Mac OS, GNU/Linux, BSD, etc) ni les mêmes versions (Windows XP, 7, 8, 10, distributions GNU/Linux, etc).

Aujourd'hui, grâce à Internet, nous pouvons accéder à de nombreuses ressources dont des compilateurs C++ dans leur dernière version. L'un d'eux s'appelle **Wandbox** [↗](#). Ce site fournit des compilateurs en ligne pour de nombreux langages, dont deux pour C++ qui sont parmi les plus connus: Clang et GCC.

Je suis conscient que cela demande un accès constant à Internet. La contrepartie est que l'on peut ainsi pratiquer partout (école, travail, ordinateur de quelqu'un d'autre) puisque aucune installation n'est requise.

II. Le début du voyage

II.2.1.0.1. Survol de Wandbox

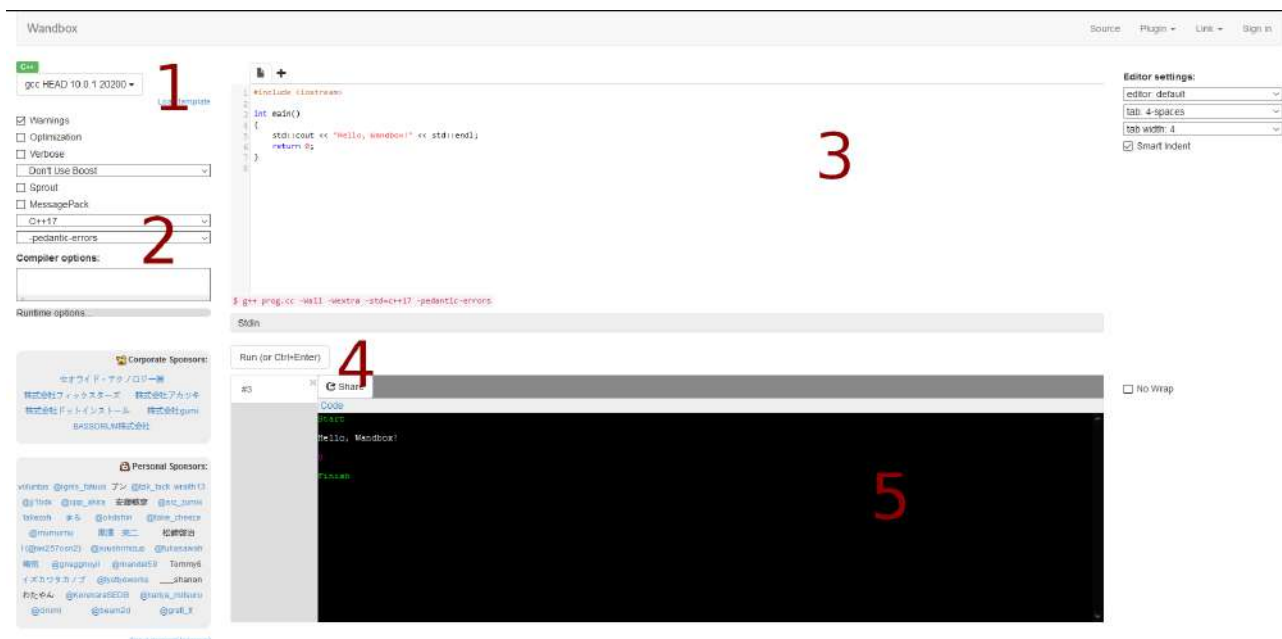


FIGURE II.2.1. – L'interface de Wandbox.


1. En haut à gauche, dans un encart vert, c'est le **langage de programmation utilisé**. Si un autre nom que C++ est écrit, pas de soucis, il suffit de cliquer sur la liste déroulante en dessous, puis sur «C++» et choisir «**GCC HEAD**» ou «**Clang HEAD**» (qui sont les toutes dernières versions de ces compilateurs).
2. Un peu plus bas, la liste déroulante commençant par C++ permet de **choisir la version du langage**. Dans ce tutoriel, nous utiliserons la version «**C++17**», qui correspond à la version 2017 du langage. Que vous choisissiez «C++17» ou «C++17 (GNU)», cela n'a aucune importance à notre niveau.
3. La grande zone de texte blanche est **là où nous écrirons le code**.
4. Enfin, le bouton **Run (or Ctrl+Enter)** servira à lancer le compilateur et lui faire compiler le code écrit dans l'encadré précédent.
5. Quand la compilation est terminée, le résultat apparait ici.

II.2.2. Des outils plus poussés

Les outils en ligne sont très bien pour des petits codes rapides ou pour faire quelques tests, mais ils sont quand même limités. Sachez qu'il existe cependant des outils très complets tant pour Windows que pour GNU/Linux. Comme mentionné dans le [cours d'introduction à la programmation](#) [↗](#), on peut utiliser un ensemble d'outils et de logiciels différents ou bien utiliser un IDE.

II.2.2.1. Visual Studio Community

L'IDE par excellence de Microsoft, qui permet de programmer non seulement en C++, mais aussi en Python, en JavaScript, en C#, etc. Configurable, extensible, léger par défaut dans sa version 2019, il est en plus **100% gratuit dans le cadre de l'apprentissage**, de la recherche ou des projets *open source*. Vous n'aurez donc pas besoin de déboursier quoi que ce soit pour l'utiliser. Inconvénient? Il n'est **disponible que pour Windows** (dans sa version 7 au minimum).

Pour le télécharger et l'installer, je vous invite à suivre [la documentation fournie par Microsoft](#) . L'installation demande une connexion Internet.

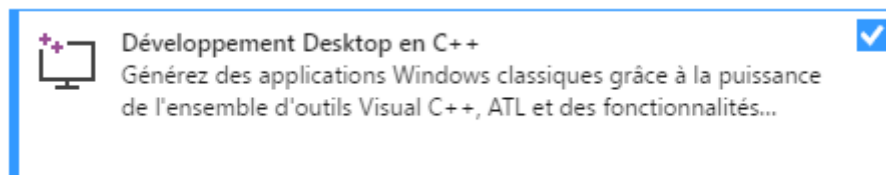


FIGURE II.2.2. – Vérifiez que la case C++ est bien cochée.

Une fois que Visual Studio est installé pour la première fois et configuré, on arrive sur l'écran d'accueil.

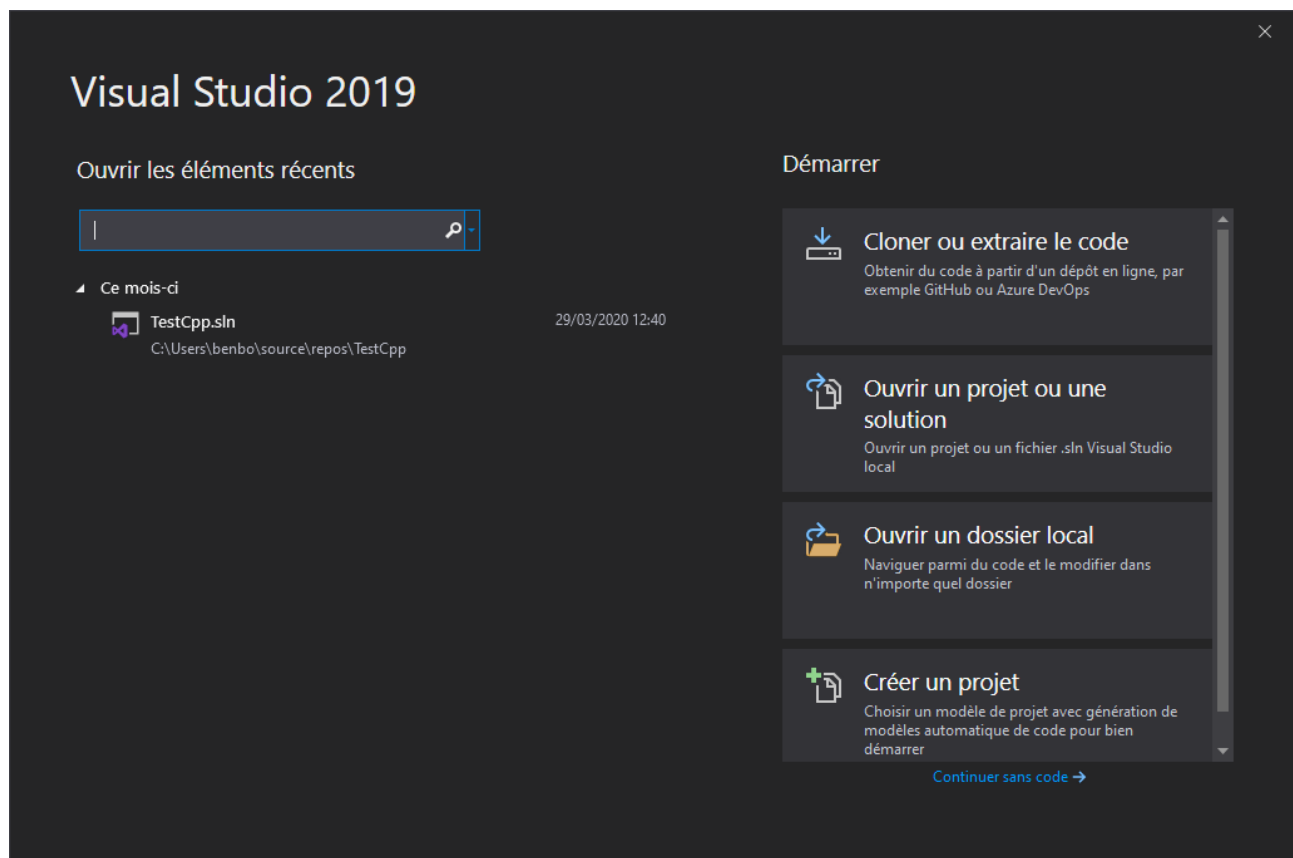


FIGURE II.2.3. – L'écran d'accueil de Visual Studio 2019.

On peut directement créer un nouveau projet, ou bien aller directement à la page principale, vide.

II. Le début du voyage

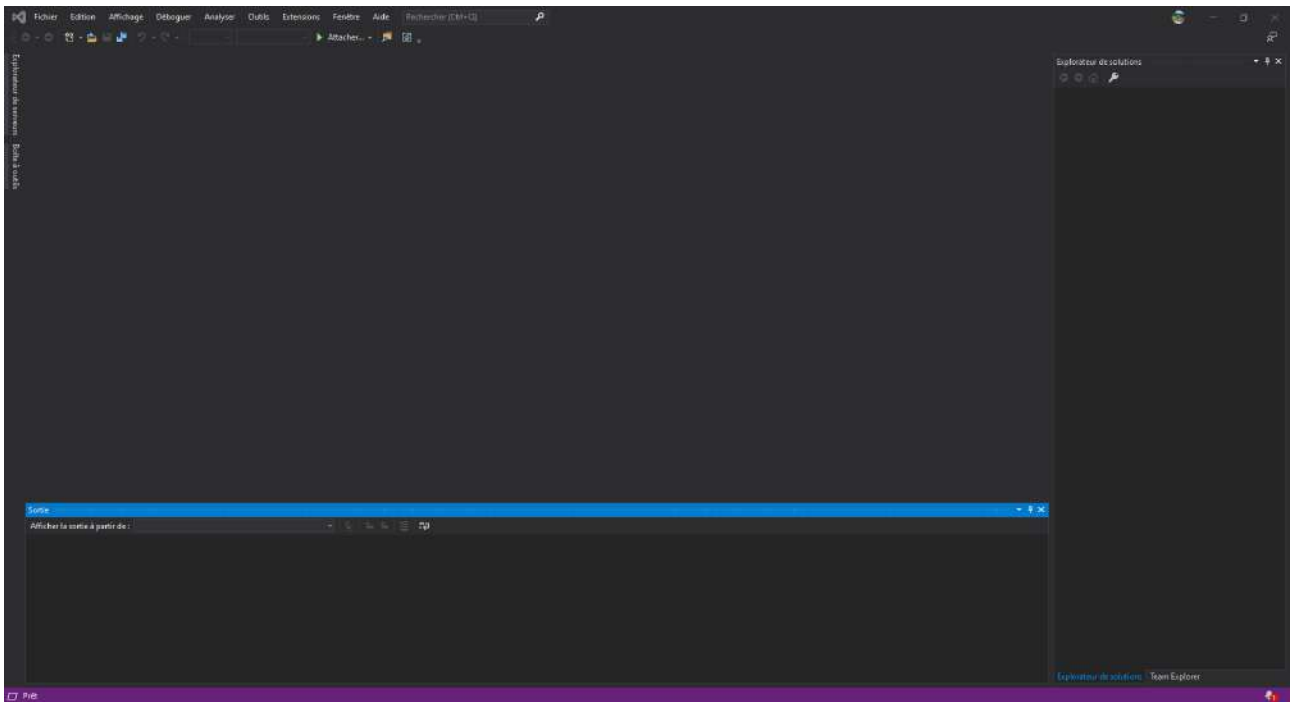


FIGURE II.2.4. – La page principale de Visual Studio 2019 est bien vide.

Que l'on choisisse de créer directement un nouveau projet, ou bien que, depuis la page d'accueil, on clique sur **Fichier -> Nouveau -> Projet...**, on se retrouve sur le même écran.

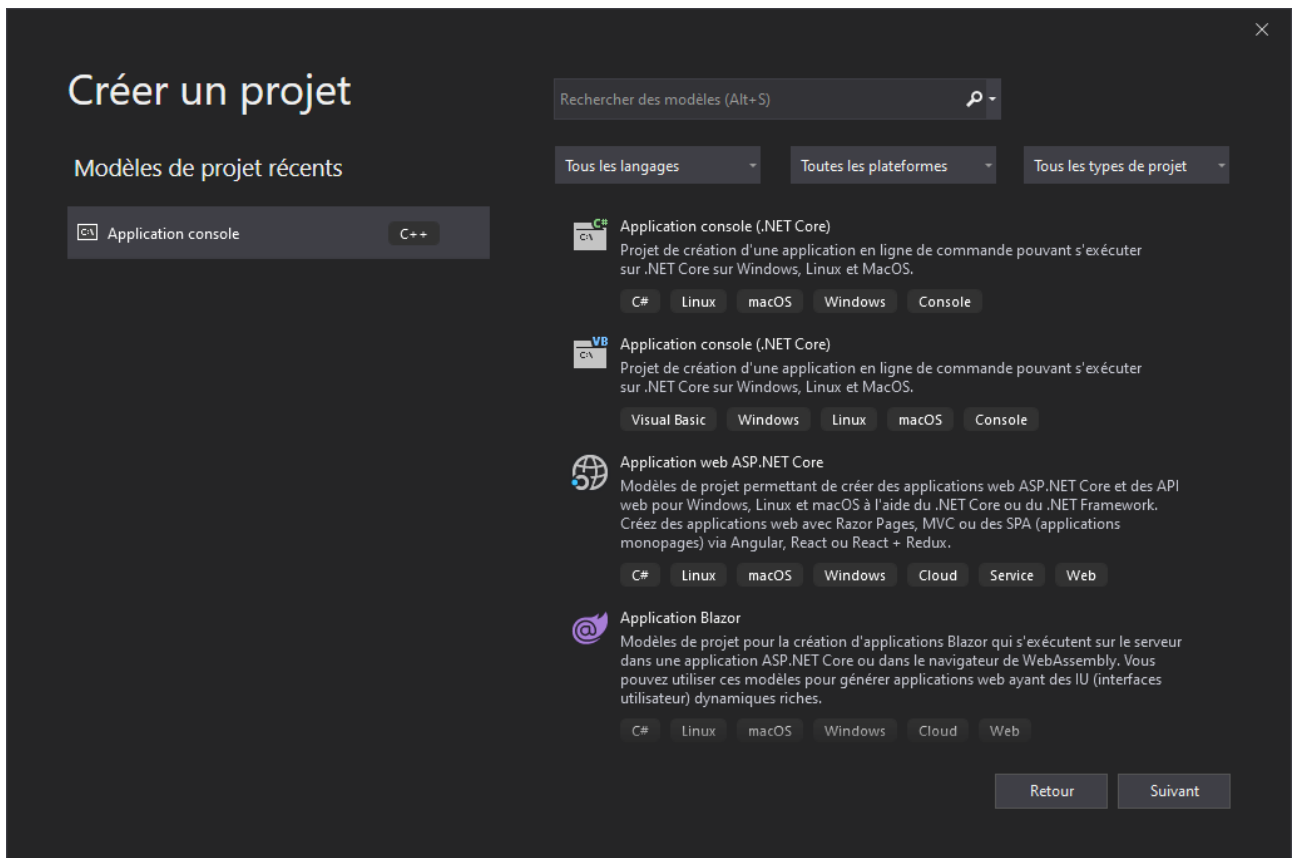


FIGURE II.2.5. – On a le choix de créer tout un tas de projets différents.

II. Le début du voyage

Dans notre cas, nous allons créer un projet vide. Sélectionnez **C** comme langage et **Console** comme type. Sur l'écran suivant, nous pouvons choisir son nom et son emplacement avant de cliquer sur **Créer**.

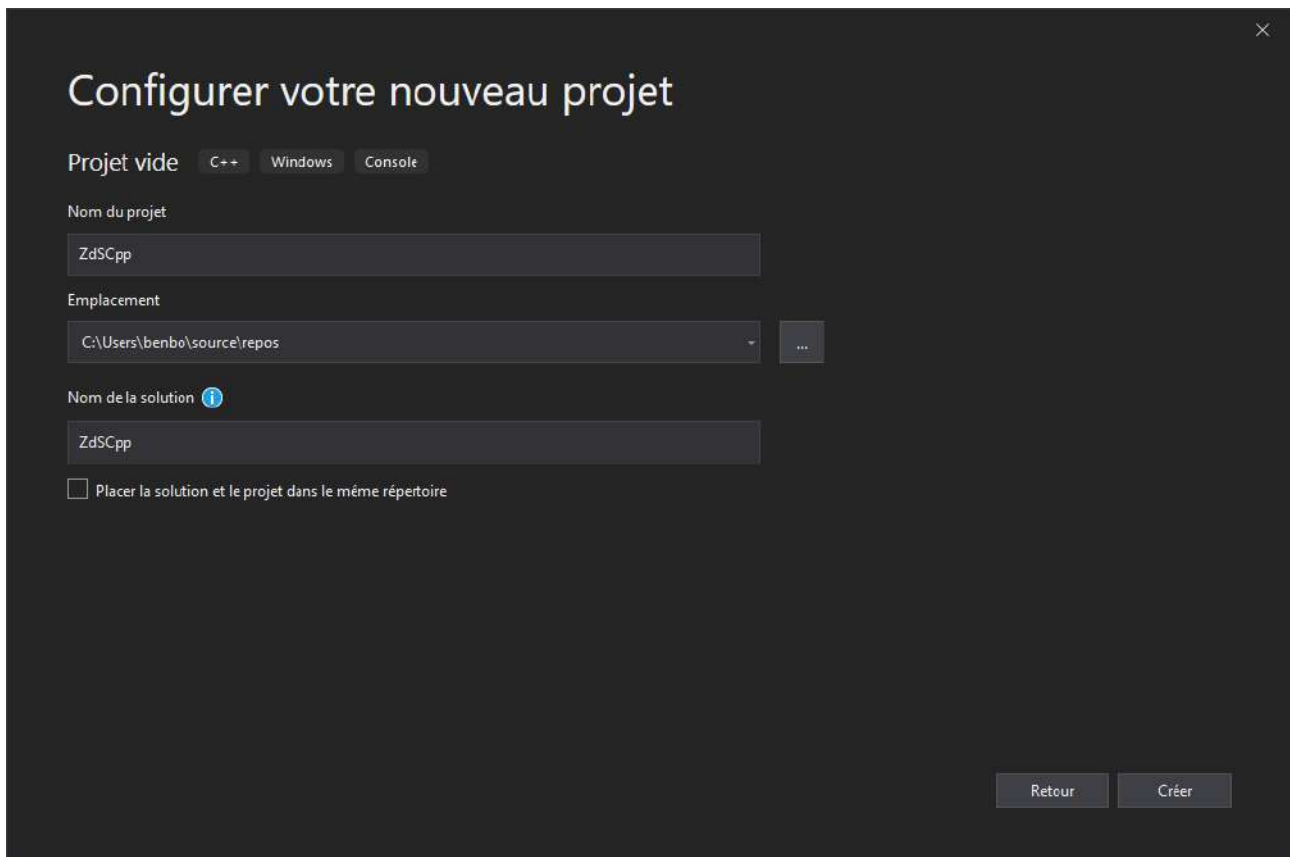


FIGURE II.2.6. – Donnez le nom que vous souhaitez à votre projet.

Une fois que votre projet est créé, faites un clic-droit sur **Fichiers sources**, à droite de l'écran, puis **Ajouter -> Nouvel élément...**. Cliquez sur **Fichier C** + **(.cpp)**, nommez le fichier **main.cpp** et validez.

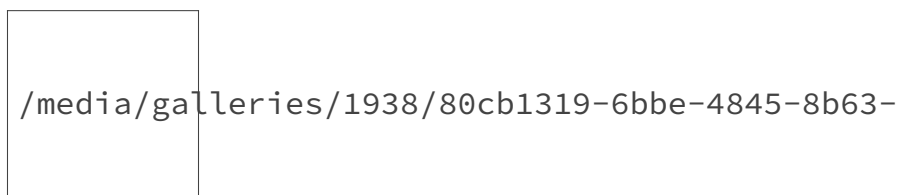
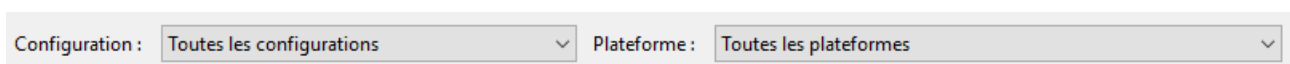


FIGURE II.2.7. – Ajout d'un fichier C++.

Il ne reste plus qu'à régler quelques paramètres du projet pour utiliser la dernière norme de C++ et nous serons bons. Pour cela, à droite, faites un clic-droit sur le nom de votre projet, puis descendez pour cliquer sur **Propriétés**. Tout en haut, mettez l'option **Configuration** à la valeur **Toutes les configurations** et l'option **Plateforme** à la valeur **Toutes les plateformes**.



II. Le début du voyage

FIGURE II.2.8. – Ce que vous devez avoir comme valeurs en haut de la fenêtre.

Dans **Propriétés de configuration**, déroulez **C/C** puis **Langage**. À la ligne **Norme du langage C++**, sélectionnez **Norme ISO C++17 (/std:c++17)** et validez.

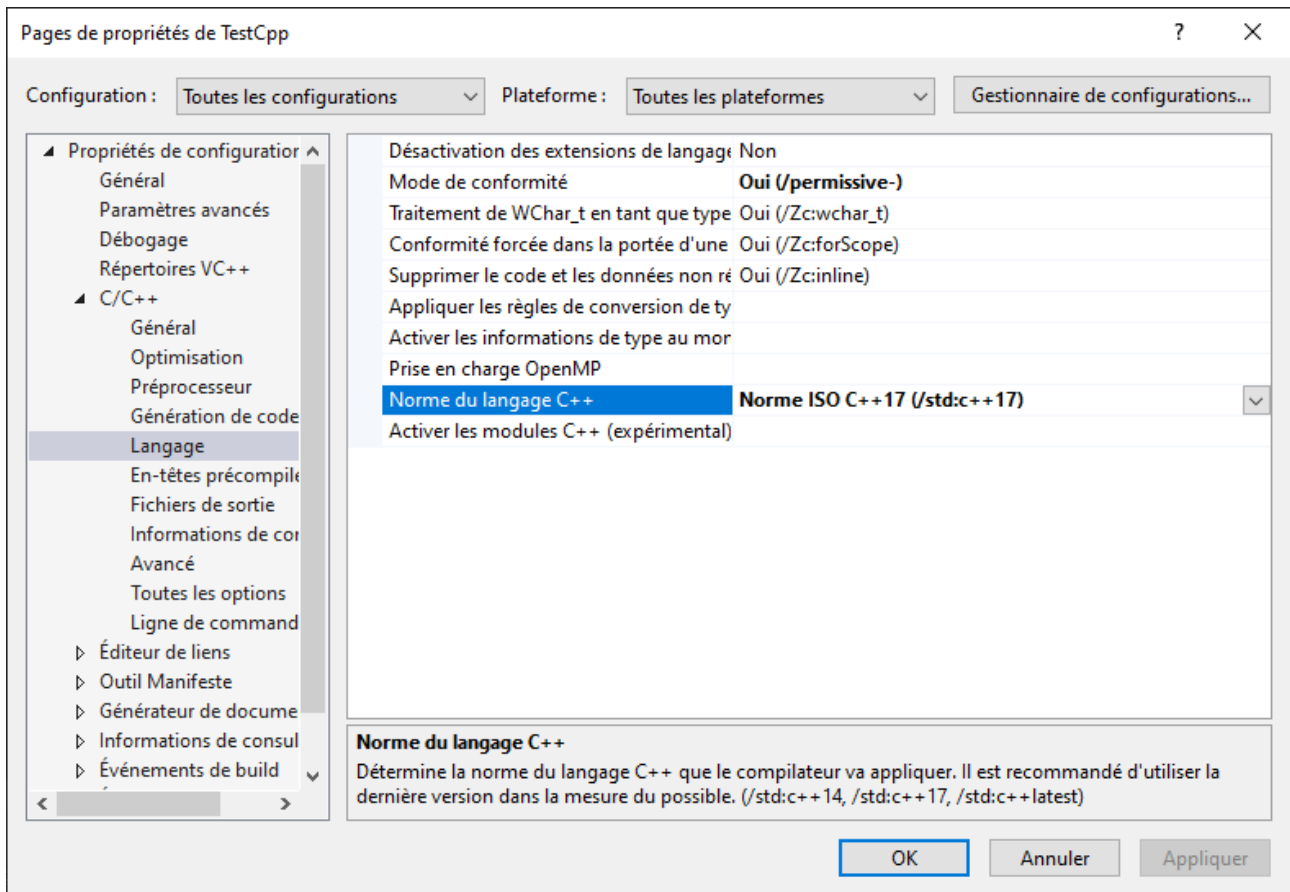


FIGURE II.2.9. – Voici la fenêtre de sélection de la norme C++.

Un deuxième paramètre à modifier se trouve dans la rubrique **C/C** + **-> En-têtes précompilés**. À la rubrique **En-têtes précompilés**, attribuez la valeur **Sans utiliser les en-têtes précompilés**.

Le dernier est toujours dans **Propriétés de configuration**, mais sous la rubrique **Éditeur de liens -> Système**. Là, attribuez à la rubrique **Sous-système** la valeur **Console (/SUBSYSTEM:CONSOLE)**, puis validez.

II.2.2.2. Qt Creator



Non supporté

Nous vous proposons ci-dessous Qt Creator car c'est un autre poids-lourd des IDE, mais **aucun de nous deux ne l'utilise ni ne saurait vous aider en cas de problème avec.**

Les explications qui suivent seront peut-être périmées ou inexactes. Nous en sommes désolé, mais cela demande beaucoup de travail que de les maintenir constamment à jour. Vous pourrez néanmoins trouver un tutoriel français [ici](#) , sur le site de @gbdivers, un spécialiste de Qt.

Un autre grand IDE pour programmer en C++ se nomme [Qt Creator](#) , disponible sous Windows, MacOS et GNU/Linux. Il est fourni par une compagnie du nom de Qt, qui édite également une bibliothèque C++ très populaire et répandue, que vous aurez peut-être l'occasion de croiser ou d'apprendre une fois ce cours terminé. 🍊

Téléchargez la version Open Source, puis installez-la. Sélectionnez la **dernière version de Qt disponible** et, dans *Tools*, prenez **la version la plus à jour de MinGW**. L'installation peut être plus ou moins longue, en fonction de la qualité de votre connexion Internet notamment.

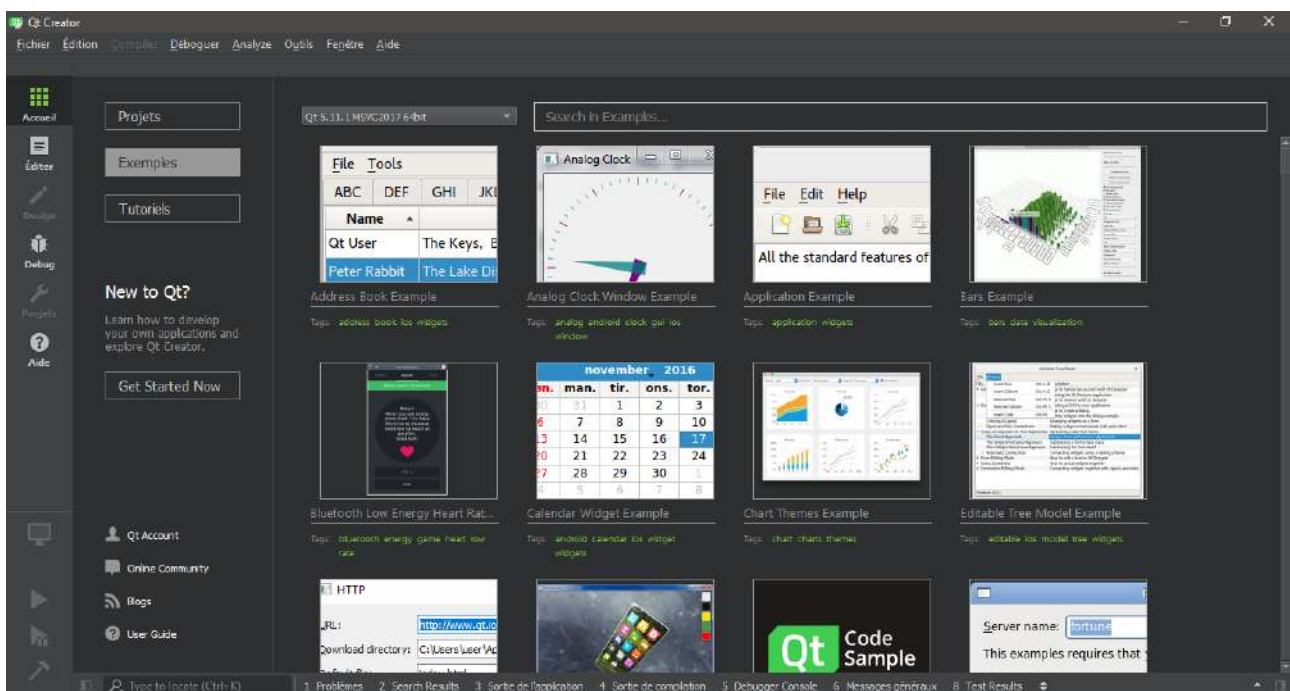


FIGURE II.2.10. – Voici l'écran d'accueil, au lancement de Qt Creator.

Pour créer un nouveau projet, cliquez sur **Fichier -> Nouveau fichier ou projet...**, puis dans **Projets -> Non-Qt Project -> P + Application**.

II. Le début du voyage

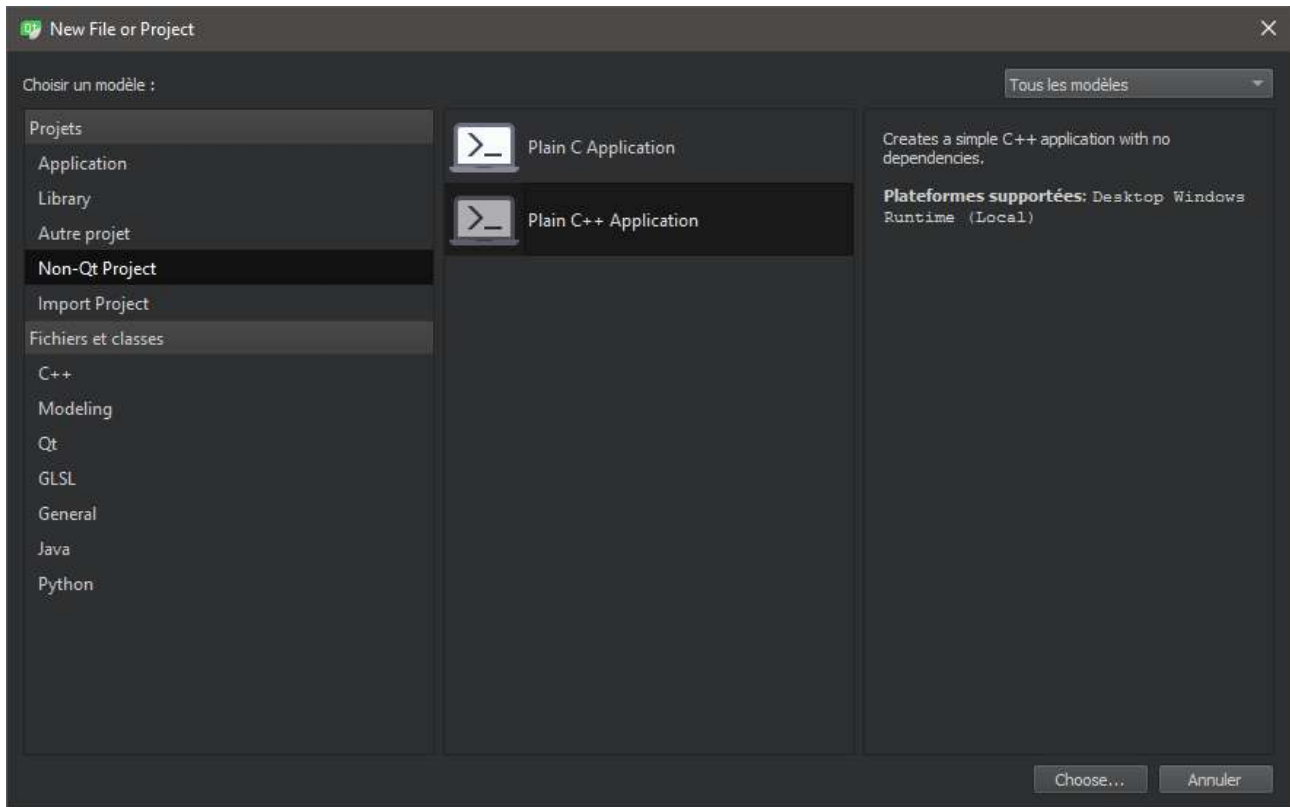


FIGURE II.2.11. – Écran de choix du projet.

Cliquez ensuite sur **Choose**. L'écran d'après vous demande le nom du projet et où il sera localisé.

II. Le début du voyage

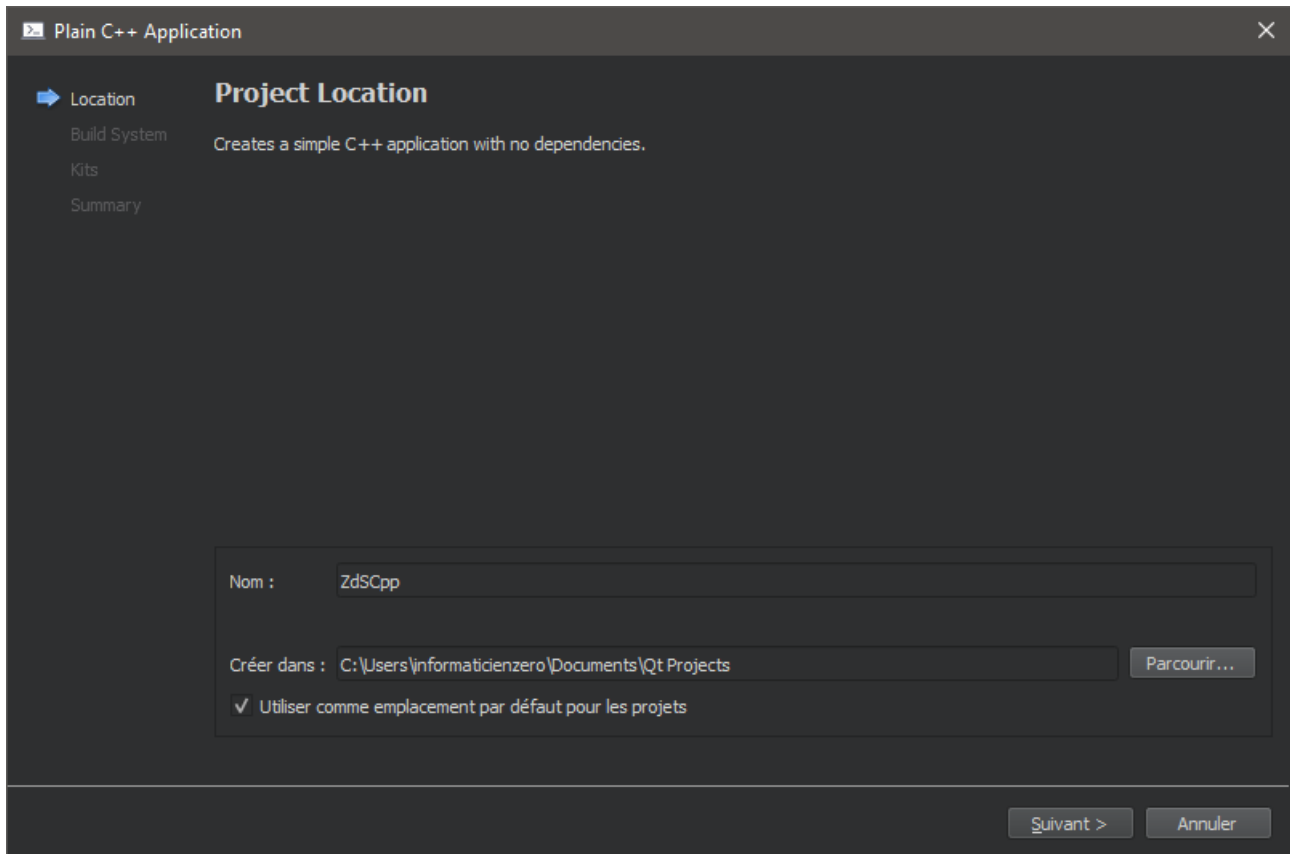
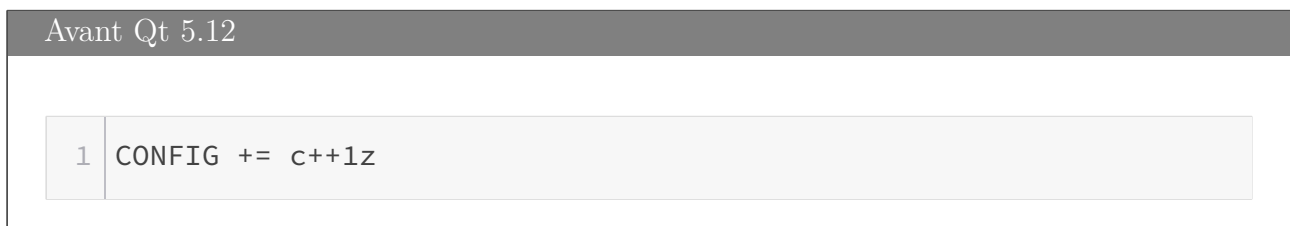


FIGURE II.2.12. – Choix du nom et de la localisation du projet.

Sur les écrans d'après, laissez les choix par défaut et cliquez à la fin sur **Terminer**. Ouvrez enfin le fichier se terminant par **.pro** et ajoutez la première ligne pour une version de Qt inférieur à 5.12, la deuxième à partir de Qt 5.12.



II.2.2.3. Pour les autres outils

Si vous souhaitez utiliser un autre outil que ceux cités plus haut, vous pouvez, bien entendu. Par contre, vous devrez vous débrouiller pour le configurer, car on ne peut malheureusement

II. Le début du voyage

pas expliquer la procédure pour tous les outils existants.

Pour les utilisateurs de XCode, vous avez de la chance, un lecteur a fait [un post décrivant la procédure à suivre](#) . 🍊

II.2.3. Un mot concernant Windows

Un petit paragraphe pour ceux d'entre vous qui tournent sous Windows, et ce peu importe quel IDE vous utilisez. Par défaut, la **console Windows** ne sait pas bien gérer les accents de notre belle langue française. Heureusement, c'est très simple à changer.

Dans le menu Démarrer, tapez `Cmd` et ouvrez l'application. Ensuite, faites un clic droit sur la barre des tâches puis cliquez sur `Propriétés`, puis `Police` et choisissez «Consolas» ou «Lucida Console» ainsi que la taille, suivant vos envies.

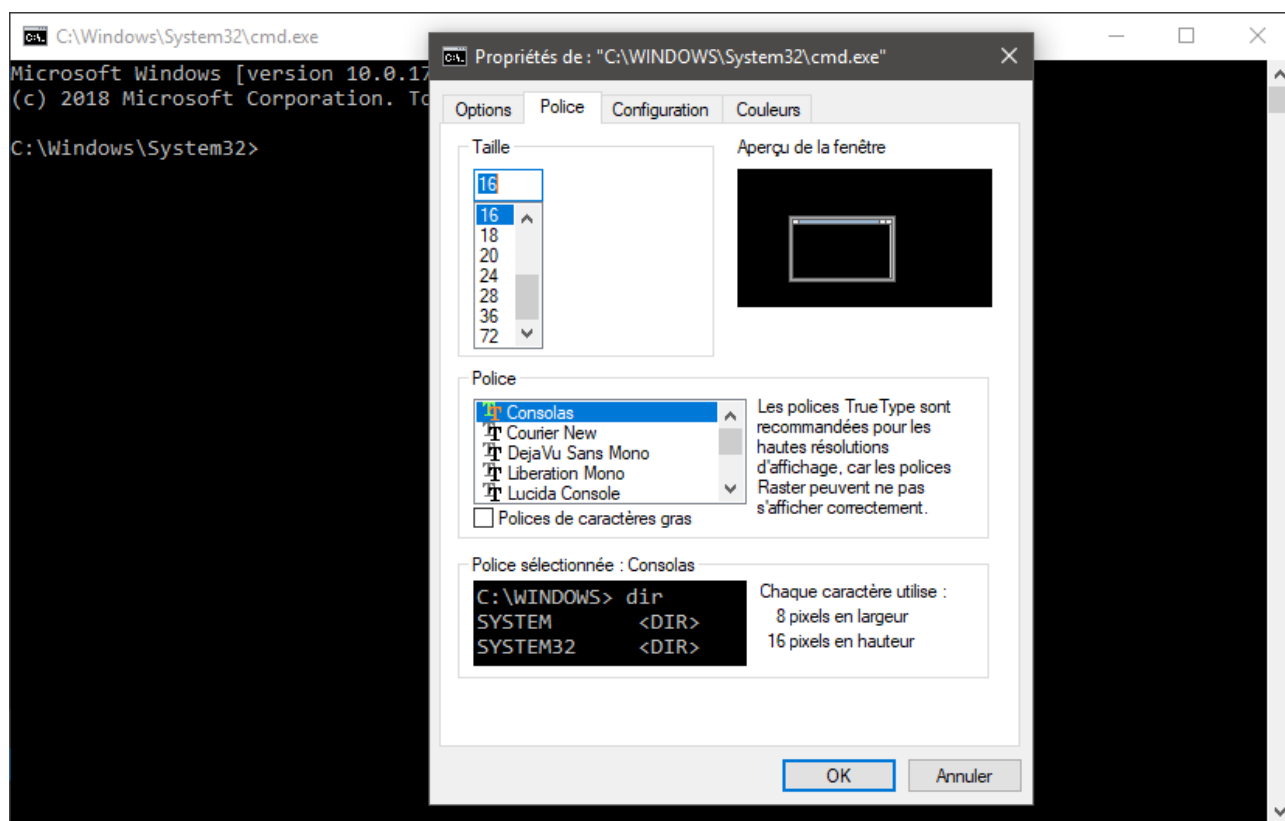


FIGURE II.2.13. – Personnellement, j'utilise Consolas en taille 16, que je trouve lisible et agréable.

II.2.4. Un mot concernant GNU/Linux

Pour nos amis les Linuxiens, il y a Qt Creator, mais ce n'est pas la seule solution. Comme expliqué dans le tutoriel introductif à la programmation, il est également possible d'**utiliser des outils séparés**. La plupart de ceux qui utilisent GNU/Linux au quotidien sont certainement déjà à l'aise avec la console et savent déjà utiliser les outils que je vais présenter. Pour les autres, suivez-moi.

II.2.4.1. Un éditeur de texte

Ce choix est le votre et peut-être que vous en avez déjà un qui vous convient très bien. Si c'est le cas, très bien, continuez dessus. Sinon, choisissez un éditeur de texte qui au moins possède **la coloration syntaxique**, c'est à dire la capacité de changer la couleur du texte en fonction de ce qu'un mot signifie. Parmi les plus connus, on peut citer [Visual Studio Code](#) , [Atom](#) ou encore [Sublime Text](#) .

II.2.4.2. Un compilateur

Ici, le choix est plus limité. Les deux compilateurs C++ dont vous entendrez le plus souvent parler, parce qu'ils sont de qualité et à jour, sont [GCC](#) et [Clang](#) . Pour la version C++ de ces compilateurs, il faut installer soit `g++`, soit `clang++`. Je vous laisse faire, en fonction des commandes d'installation de paquets de votre distribution préférée.

Une fois ceci fait, reste à vous montrer **comment compiler un code C++**. La commande suivante est illustrée avec GCC, mais si vous utilisez Clang, il n'y a qu'à changer le compilateur, le reste étant identique.

```
1 g++ -std=c++17 fichier.cpp -o fichier.out
```

La commande demande à GCC de compiler le fichier `fichier.cpp` et de générer le programme final avec `fichier.out` comme nom. L'option `-std=c++17` indique au compilateur qu'on veut utiliser la version 2017 de C++.

Ne reste plus qu'à écrire un code pour le compiler, ce que nous allons justement voir dans le chapitre suivant. 🍊

II.2.4.3. En résumé

- Nous avons la possibilité de faire des tests très rapidement avec des outils en ligne, qui ne demandent rien d'autre qu'une connexion Internet.
- Nous avons aussi des outils très complets qui nous facilitent la vie, comme Visual Studio 2017 ou Qt Creator. Ils peuvent être utilisés sans connexion Internet et font partie des outils les plus utilisés professionnellement, donc vous gagnez à apprendre à les utiliser.
- Pour que la console Windows soit capable d'afficher les accents, il ne faut pas oublier de changer la police.
- Concernant GNU/Linux, si l'on dispose d'un choix énorme d'éditeurs de texte, les deux grands compilateurs majoritairement utilisés aujourd'hui pour C++ sont GCC et Clang.

II.3. Rencontre avec le C++

Que diriez-vous de rentrer directement dans le vif du sujet **en écrivant votre premier code C++**? Nous allons commencer par l'un des exemples les plus connus et les plus utilisés: **afficher un message**. Et, vous verrez, il y a déjà de quoi dire.

II.3.1. Compilons notre premier programme

Allez donc sur [Wandbox](#) , ou bien ouvrez votre projet C++, et copiez-collez ce code dans la zone de texte.

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello World !" << std::endl; // affiche Hello
6     World !
7     return 0;
8 }
```

Cliquez ensuite sur **Run** (pour WandBox), faites **Ctrl+F5** (Visual Studio Community), **Ctrl+R** (Qt Creator), ou compilez et exécutez en ligne de commande, et sous vos yeux émerveillés s'affiche le résultat suivant.

```
1 Hello World !
```

Félicitations, vous venez de **compiler votre tout premier programme C++**! Mais je devine que vous voulez comprendre ce qui s'est passé et avoir des explications sur ce que signifie ce code.

II.3.2. Démystification du code

II.3.2.1. Inclure des fonctionnalités déjà codées

```
1 #include <iostream>
```

Le but de notre programme est d'afficher un message. Des développeurs experts ont déjà créé un outil qui permet de le faire facilement. Il se trouve dans un fichier nommé «*iostream*», acronyme de «*Input Output Stream*», soit «Flux d'Entrées Sorties» dans la langue de Molière. Ce fichier fait partie de la **bibliothèque standard C++**, un ensemble de fonctionnalité déjà pré-codées et incluses partout avec chaque compilateur C++.

Pour utiliser les fonctionnalités offertes par ce fichier, notamment écrire un message, on doit l'importer dans notre programme. On dit qu'on l'inclut, d'où l'anglais «*include*». Nous utiliserons beaucoup cette fonctionnalité tout au long du cours.

Enfin, dernière remarque, avez-vous noté que la ligne commence par le symbole **#**? C'est ce qu'on appelle une **directive de préprocesseur**. Le préprocesseur est un programme qui se lance automatiquement au début de la compilation, notamment pour importer les fichiers qu'on lui demande. Nous le détaillerons plus loin dans ce cours. Retenez que **#include nous permet d'importer des fichiers** que des programmeurs experts ont déjà écrits avant nous.

II.3.2.2. Le point d'entrée

```
1 int main()  
2 {  
3  
4 }
```

Quant l'ordinateur exécute un programme, il lui faut bien un endroit où commencer, un début, un point d'entrée. À partir de ce point d'entrée, **il exécutera des instructions, des ordres**, que nous aurons au préalable écrits. Cet endroit s'appelle **main** (prononcé «*mèïne*») et est la fonction principale de tout programme C++.

Qu'est-ce qu'une fonction? C'est un **un regroupement d'instructions**. En C++, vous trouverez ainsi des fonctions pour calculer une racine carrée, pour comparer des durées, trier une liste d'objets, afficher des messages, etc. Nous en utiliserons beaucoup déjà existantes et, plus tard, apprendrons à créer nos propres fonctions.

Dans notre cas, la fonction **main** regroupe deux instructions différentes.

II.3.2.3. Voici mes instructions...

```
1 std::cout << "Hello World !" << std::endl; // Affiche le message  
    Hello World !
```

La première instruction est justement celle qui nous permet d’afficher un message. Elle débute avec `std::cout` (à prononcer «ci-aoûte»), qui représente **la sortie standard en C++**, c’est-à-dire la communication du programme vers l’utilisateur, par l’écran dans notre cas. Le préfixe `std::` indique que vous utilisez une fonctionnalité déjà programmée et fournie dans la bibliothèque standard C++.

On note ensuite la présence de deux chevrons `<` qui permettent **d’injecter, d’envoyer, du contenu dans la sortie standard**, dans `std::cout`. Dans notre code d’introduction, nous injectons deux choses.

- D’abord, `"Hello World !"`, qui est le **message que l’on voit s’afficher** à l’écran.
- Puis `std::endl`, qui permet d’**aller à la ligne**, comme son nom l’indique («*end line*» que l’on traduit par «retour à la ligne»). De même que pour `std::cout`, `std::endl` est fourni par la bibliothèque standard.

Enfin, elle se termine par un point-virgule `;` qui est **obligatoire**. Chaque instruction doit être terminée par un point-virgule. Ce sont les règles du C++ qui veulent ça.



Chevrons

On peut combiner les chevrons `<` autant que l’on souhaite. Essayez donc d’afficher la phrase «Bonjour, j’écris du C++» suivi d’un retour à la ligne, suivi de «Et j’aime beaucoup.»

👁 Contenu masqué n°1



Un mot sur Windows

Si vous êtes sous Windows et que vous tentez d’afficher du texte avec des accents, vous allez obtenir des caractères bizarres. Pour afficher du code avec des accents, vous allez devoir ajouter les lignes 2, 3 et 8 dans votre code. Nous en reparlerons plus tard.



```
1 #include <iostream>
2 #define NOMINMAX
3 #include <Windows.h>
4
5 int main()
6 {
7     // À utiliser avant d'afficher du texte.
8     SetConsoleOutputCP(1252);
9     std::cout << "Oui, je peux utiliser éèàï !" << std::endl;
10 }
```

Sous GNU/Linux et MacOS, vous n'aurez aucun problème, donc ignorez ces lignes.

```
1 return 0;
```

La deuxième instruction conclut la fonction `main` en renvoyant la valeur `0`. Le mot anglais pour renvoyer est *return*; vous croiserez d'ailleurs souvent l'anglicisme retourner.

Mais à qui est transmise cette valeur? Au système d'exploitation. Le zéro, par convention, signifie que tout s'est bien passé.

Nous aurons l'occasion de comprendre mieux tout cela quand nous aborderons le chapitre sur les fonctions. Ne vous inquiétez donc pas de `return 0`; **sachez simplement qu'il termine la fonction `main`.**

II.3.3. Les commentaires

Cela ne vous a pas échappé, mais j'ai omis de traiter un bout de code. En effet, cette partie bizarre avec des slashes.

```
1 // Affiche le message Hello World !
```

C'est ce qu'on appelle **un commentaire**. Les commentaires sont complètement ignorés par le compilateur, **ils ne servent qu'aux humains**. On s'en sert pour documenter son code, expliquer des passages un peu ardu, décrire des passages un peu moins lisibles ou tout simplement pour offrir quelques compléments d'information.

En effet, un code est bien plus souvent lu qu'écrit ou modifié. C'est pourquoi il est important **qu'il soit lisible**. Les commentaires aident à gagner en lisibilité.

Enfin, contrairement au code qui contient beaucoup de mots anglais, les commentaires peuvent être écrits dans la langue que l'on souhaite. Si beaucoup de projets collaboratifs ont des

II. Le début du voyage

commentaires écrits en anglais, dans ce tutoriel, je les écrirai en français afin de vous aider à bien comprendre.

II.3.3.1. La syntaxe

On peut écrire des commentaires de deux façons. La première, c'est celle que nous avons vu avec les deux slashes `//`. Un commentaire ainsi créé s'arrête à la fin de la ligne où il est créé. On dit que c'est un **commentaire mono-ligne**.

```
1 // Cette ligne affiche Bonjour.
2 std::cout << "Bonjour";
3
4 // Cette ligne permet d'afficher un retour à la ligne.
5 std::cout << std::endl;
```

L'autre syntaxe permet d'écrire des blocs de commentaires bien délimités entre les symboles `/*` et `*/`. On peut ainsi écrire des commentaires sur plusieurs lignes, voire entre plusieurs instructions!

```
1 /* Ceci est un commentaire sur une seule ligne. */
2 std::cout << "Hello World !" << std::endl; /* Ceci est un
   commentaire à la fin d'une ligne du code. */
3
4 /* Ceci...
5 ...est un commentaire
6 écrit sur plusieurs lignes...
7 ...car il est très long. */
8
9 std::/* Ceci est un commentaire noyé dans le code. */cout <<
   "Hello World !";
```

Vu que les commentaires sont ignorés par le compilateur, on peut tout à fait en écrire en plein milieu d'une ligne de code. Mais ce n'est pas lisible, aussi je vous déconseille fortement de le faire, car un code est, je vous le rappelle, plus souvent lu qu'écrit.

II.3.3.2. En résumé

- Un programme en C++ est composé d'instructions, qui sont les ordres que l'on donne à l'ordinateur.
- Ces instructions sont regroupées dans des blocs appelés fonctions. Celle par laquelle commence tout programme C++ s'appelle `main`.
- Nous avons utilisé la bibliothèque standard pour la première fois, avec le fichier `iostream`, afin d'afficher un message à l'écran.

II. Le début du voyage

- Les commentaires sont ignorés par le compilateur mais nous sont très utiles à nous, humains.

Contenu masqué

Contenu masqué n°1

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Bonjour, j'écris du C++" << std::endl <<
6         "Et j'aime beaucoup.";
7     return 0;
8 }
```

[Retourner au texte.](#)

II.4. Une super mémoire

Vous savez désormais afficher des messages à l'écran. C'est un bon début, mais on est quand même assez limités, n'est-ce pas? Rassurez-vous, C++ nous permet de faire bien plus que ça. Grâce à ce langage, nous allons nous amuser avec cette **super calculatrice** qu'est notre ordinateur et nous découvrirons qu'il possède aussi **une super mémoire**!

Ce chapitre va nous faire découvrir les **littéraux** et les **variables**.

II.4.1. Les littéraux

Que sont les littéraux? **Une valeur écrite littéralement dans le code**, d'où son nom. Nous avons rencontré un littéral dans le chapitre précédent: `"Hello World !"`. Ce littéral est ce qu'on appelle une **chaîne de caractères**. Mais ce n'est pas le seul littéral possible en C++. Examinons tout ça.

II.4.1.1. Les caractères

Nous avons fait connaissance avec les chaînes de caractères lors du chapitre précédent. Les chaînes de caractères ne sont rien d'autre que du texte. On les reconnaît parce qu'elles commencent et finissent par des doubles guillemets `"`.

Entraînez-vous donc à afficher des chaînes de caractères, par exemple un littéral souhaitant une bonne journée.

👁 Contenu masqué n°2

Tous comme les mots de la langue française sont des regroupements de lettres, les chaînes de caractères sont une suite de **caractères simples**. En C++, un caractère est encadré par des guillemets simples `'`.

```
1 #include <iostream>
2
3 int main()
4 {
5     // Un caractère peut être une lettre.
6     std::cout << 'A' << std::endl;
7     // Ou bien un chiffre.
```

II. Le début du voyage

```
8     std::cout << '7' << std::endl;
9     // Ou même de la ponctuation.
10    std::cout << '!' << std::endl;
11
12    return 0;
13 }
```

Les chaînes de caractères permettent de regrouper des caractères et nous facilitent ainsi la vie pour écrire du texte.

```
1 #include <iostream>
2
3 int main()
4 {
5     // Que c'est fastidieux d'écrire ça !
6     std::cout << 'S' << 'a' << 'l' << 'u' << 't' << ' ' << 't' <<
7         'o' << 'i' << ' ' << '!' << std::endl;
8     // C'est tellement mieux comme ça.
9     std::cout << "Salut toi !" << std::endl;
10
11    return 0;
12 }
```

?

J'ai mis plusieurs caractères entre guillemets simples, comme ceci 'abc' et mon code compile. Ça veut dire que ça marche?

Si vous avez essayé de compiler, vous avez remarqué un message en rouge dans la fenêtre de résultat disant `warning: multi-character character constant`, ainsi qu'un nombre au lieu de nos caractères.

Le **warning** est un message d'avertissement que le compilateur vous envoie, car il ne sait pas si ce que vous avez fait est une erreur d'inattention ou bien une manœuvre volontaire de votre part. Dans notre cas, la norme C++ n'interdit pas de faire ça, mais **ce n'est pas considéré comme une bonne pratique**, comme étant du bon code.

!

N'ignorez pas les warnings!

Les *warnings* sont des messages importants signalant d'éventuels problèmes. Il ne faut surtout pas les ignorer sous prétexte que le code compile!

II.4.1.1.1. Des caractères spéciaux


Il existe quelques caractères qui sont un peu particuliers et, pour les introduire, je vais vous demander d'afficher un message contenant un chemin de dossier Windows (C:\Program Files

II. Le début du voyage

(x86) par exemple).

👁 Contenu masqué n°3

Boum ba da boum! Le compilateur vous crache à la figure un message de type `warning: unknown escape sequence: '\P'`. Que s'est-il passé?

Il existe en C++ des séries de caractères, appelées **séquences d'échappement**, qui commencent toutes par `\` et dont la liste complète se trouve [ici](#) . Et pour comprendre l'intérêt de ces séquences d'échappement, essayez donc de compiler le code suivant.

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Il m'a demandé " Comment vas-tu ? "" <<
6         std::endl;
7     return 0;
8 }
```

Vous devez certainement avoir une erreur proche de celle-ci: `'error: expected ';' before 'Comment'`. C'est tout à fait normal. En effet, nous avons vu plus haut que les chaînes de caractères sont délimitées par les doubles guillemets `"`. Donc tout ce qui est entre cette paire de guillemets est considéré par le compilateur comme faisant partie de la chaîne de caractères.

Dans notre exemple, cela veut dire que **Comment vas-tu ? ne fait plus partie** de la chaîne. Donc le compilateur l'interprète comme des instructions C++ et comme il ne les comprend pas, il ne peut pas compiler le programme.

?

Mais quel rapport avec ces fameux caractères commençant par `\`?

Eh bien, les séquences d'échappement permettent de dire au compilateur «Écoute, ce caractère est spécial, il faut l'afficher et non l'interpréter», ce qui permet d'afficher des doubles guillemets dans une chaîne de caractères par exemple.

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Il m'a demandé \" Comment vas-tu ? \"" <<
6         std::endl;
7     std::cout << "Dossier principal : C:\\Program Files (x86)" <<
8         std::endl;
```

II. Le début du voyage

```
7  
8     return 0;  
9 }
```

De toutes les séquences d'échappement qui existe, les plus utilisées et celles que vous verrez le plus souvent sont les suivantes.

- `\'` qui permet d'afficher un guillemet simple `'`.
- `\"` qui permet d'afficher un guillemet double `"`.
- `\n` qui permet d'aller à la ligne, comme `std::endl`.
- `\t` qui permet de faire une tabulation horizontale.
- `\\` qui permet d'afficher un antislash `\`.

Entraînez-vous donc en affichant un guillemet simple sans échappement, puis avec échappement. Et faites donc mumuse avec les tabulations et les retours à la ligne.

👁 Contenu masqué n°4

II.4.1.2. Les nombres

Il existe d'autres types de littéraux en C++: les nombres entiers et les nombres à virgule, appelés **flottants**.

```
1 std::cout << -1 << std::endl;  
2 std::cout << 0 << std::endl;  
3 std::cout << 1 << std::endl;  
4  
5 std::cout << -1.6027 << std::endl;  
6 std::cout << 3.14159 << std::endl;  
7 std::cout << 2.71828 << std::endl;
```

On remarque déjà qu'on peut utiliser des nombres négatifs sans aucun problème. Ensuite, dû à l'origine américaine de C++, **les flottants s'écrivent avec un point et non une virgule**. Même si l'on se fiche des chiffres après la virgule, il faut mettre le point `.` sinon C++ interprétera le nombre comme un entier.

Sinon, rien de bien étonnant, ce sont des nombres et on peut faire des opérations dessus.

```
1 #include <iostream>  
2  
3 int main()  
4 {  
5     std::cout << "1 + 2 = " << 1 + 2 << std::endl;
```

II. Le début du voyage

```
6      std::cout << "1 - 4 = " << 1 - 4 << std::endl;
7      std::cout << "7.5 * 2 = " << 7.5 * 2 << std::endl;
8
9      std::cout << "9 / 2 = " << 9 / 2 << std::endl;
10     std::cout << "9. / 2 = " << 9. / 2 << std::endl;
11     std::cout << "9 % 2 = " << 9 % 2 << std::endl;
12
13     return 0;
14 }
```

?

Attends, qu'est-ce que c'est que ça? $9 / 2$ et $9. / 2$ ne donne pas la même chose? Et c'est quoi %?

En fait, la raison est très simple: pour C++, si on fait une opération sur deux nombres entiers, le résultat est un nombre entier. Si l'on veut que le résultat soit un nombre à virgule, il faut qu'au moins un des deux nombres soit un flottant.

Dans le cas de la division, si les deux nombres sont des entiers, on obtient le quotient de la division euclidienne, le reste s'obtenant avec l'opérateur %, appelé **modulo** (c'est de l'[arithmétique](#) [↗](#)). Si l'un des deux nombres est un flottant, alors on obtient le résultat de la division réelle, donc un nombre réel.

Également, comme en maths, les calculs respectent la [distributivité](#) [↗](#), l'[associativité](#) [↗](#), la [commutativité](#) [↗](#) et la [priorité des opérateurs](#) [↗](#). L'exemple suivant est tiré du cours de @gbdivers.

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Commutativité :" << std::endl;
6      std::cout << "2 + 3 = " << 2 + 3 << std::endl;
7      std::cout << "3 + 2 = " << 3 + 2 << std::endl;
8
9      std::cout << "Associativité :" << std::endl;
10     std::cout << "2 + (3 + 4) = " << 2 + (3 + 4) << std::endl;
11     std::cout << "(2 + 3) + 4 = " << (2 + 3) + 4 << std::endl;
12
13     std::cout << "Distributivité :" << std::endl;
14     std::cout << "2 * (4 + 3) = " << 2 * (4 + 3) << std::endl;
15     std::cout << "2 * 4 + 2 * 3 = " << 2 * 4 + 2 * 3 << std::endl;
16
17     std::cout << "Priorité des opérateurs :" << std::endl;
18     std::cout << "2 * 6 + 3 = " << 2 * 6 + 3 << std::endl;
19     std::cout << "2 * (6 + 3) = " << 2 * (6 + 3) << std::endl;
20     std::cout << "2 * 4 - 6 / 2 = " << 2 * 4 - 6 / 2 << std::endl;
```

II. Le début du voyage

```
21  
22     return 0;  
23 }
```

Nous nous sommes amusés avec les littéraux, mais on est quand même rapidement limité. Imaginons que nous voulions multiplier le résultat précédent par 2 et ce, trois fois. Nous sommes obligés d'écrire un code comme ceci.

```
1  #include <iostream>  
2  
3  int main()  
4  {  
5      std::cout << "1 * 2 = " << 1 * 2 << std::endl;  
6      std::cout << "1 * 2 * 2 = " << 1 * 2 * 2 << std::endl;  
7      std::cout << "1 * 2 * 2 * 2 = " << 1 * 2 * 2 * 2 << std::endl;  
8  
9      return 0;  
10 }
```

Maintenant, imaginons qu'on ne multiplie plus par 2 mais par 4, et non plus trois fois mais cinq. Vous visualisez bien tous les changements à faire? C'est pénible, n'est-ce pas? N'y a-t-il pas un moyen de se souvenir de valeurs et de calculs?

II.4.2. Les variables

La réponse à la question soulevée dans la section précédente se trouve dans le titre. Il s'agit des **variables**. C'est un concept commun à beaucoup de langages de programmation qui permet de **stocker une valeur** et de **lui associer un nom**, afin de faciliter tant l'écriture que la lecture du code. On peut les voir comme des enveloppes, des tiroirs, des conteneurs, bref, une zone où est stockée une valeur, à laquelle on associe un nom.

?

Elle est où, cette «zone de stockage»?

Toutes les variables sont stockées dans la mémoire vive de l'ordinateur, [la RAM](#) [↗](#). Son fonctionnement est un peu complexe et ne nous intéresse pas (dans le cadre de ce cours tout du moins). Les curieux trouveront beaucoup de ressources sur Internet pour combler leur soif de savoir, allant de la [vulgarisation](#) [↗](#) simple à des [explications complètes](#) [↗](#).

II.4.2.1. Comment créer des variables en C++?

Pour déclarer une variable en C++, il faut trois éléments obligatoires.

II. Le début du voyage

- D'abord, un **type**, qui indique ce que la variable va stocker (un entier, une chaîne de caractères, etc).
- Ensuite, un **identificateur**, c'est-à-dire le nom associé à la variable.
- Enfin, il faut bien donner **une valeur** à stocker à notre variable. Ceci se fait en mettant cette valeur entre accolades { }.

Examinons un cas réel que nous détaillerons.

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     int reponse { 42 };
7     std::cout << "La réponse à la Grande Question est " << reponse
8         << std::endl;
9
10    double pi { 3.1415926 };
11    std::cout << "Voici la valeur du célèbre nombre pi : " << pi <<
12        std::endl;
13
14    char lettre { 'A' };
15    std::cout << "La première lettre de l'alphabet français est "
16        << lettre << std::endl;
17
18    std::string phrase { "Bonjour tout le monde !" };
19    std::cout << "En entrant dans la salle, il s'écria : " <<
20        phrase << std::endl;
21
22    return 0;
23 }
```

```
1 La réponse à la Grande Question est 42
2 Voici la valeur du célèbre nombre pi : 3.14159
3 La première lettre de l'alphabet français est A
4 En entrant dans la salle, il s'écria : Bonjour tout le monde !
```



Syntaxe héritée

Il existe une syntaxe alternative, de la forme `type identificateur = valeur;`. Essayez, vous verrez que ça marche.



```
1 #include <iostream>
2
3 int main()
4 {
5     int variable = 42;
6     std::cout << variable << std::endl;
7
8     return 0;
9 }
```

Cette syntaxe est héritée du C. Elle est toujours valable en C++, ne soyez donc pas surpris si vous la voyez un jour dans des codes divers. Dans ce cours, nous utiliserons la forme dite «moderne» utilisant les accolades `{}`.

II.4.2.1.1. C'est tout à fait mon type

Les plus attentifs ont remarqué que l'on retrouve les mêmes littéraux que dans la section précédente.

- Pour les **nombres entiers**, nous avons le mot-clé **int**, abréviation de l'anglais *integer* signifiant ... nombre entier. Grâce à ce type, on peut stocker des entiers négatifs ou positifs.
- Pour les **flottants**, les nombres à virgule, nous avons le mot-clé **double**, qui permet de stocker des nombres réels très grands.
- Pour les **caractères simples**, nous avons **char**, qui est tellement transparent que je ne vous ferai pas l'affront de le traduire.
- Enfin, pour les **chaînes de caractères**, nous avons le type **std::string**. Ce type est un peu particulier car il n'existe pas nativement en C++. Ce sont des programmeurs experts qui l'ont codé afin de manipuler aisément des chaînes de caractères. Afin de pouvoir manipuler des **std::string**, il faut donc inclure le bon fichier, ce que l'on fait grâce à la ligne `#include <string>`.

II.4.2.1.2. Quel est ton nom, petite variable ?

D'autre, puristes de la langue française, auront remarqué qu'il manque l'accent aigu dans l'identificateur **reponse**. Ce n'est pas un oubli de ma part. Si je suis obligé de faire cette entorse à notre belle langue, c'est parce que **C++ m'y force**. Étant un langage inventé aux États-Unis dans les années 1980, **C++ n'autorise que les 26 lettres composant l'alphabet anglais** (plus les chiffres et l'underscore `_`), à savoir donc les mêmes que les nôtres, accents non compris. Dans la réalité, les programmeurs utilisent l'anglais pour nommer leurs variables, ce n'est donc pas un problème.



Langue du cours

Je n'utilise le français que dans le cadre de votre apprentissage. Quand vous aurez plus d'expérience, je vous encouragerai à utiliser l'anglais.

Mais que ce soit en anglais ou en français, un identificateur doit **respecter des règles bien précises**.

- Il doit **commencer par une lettre**. Il ne peut pas commencer par un chiffre, c'est interdit. Il ne doit pas commencer non plus par un underscore `_` : leur utilisation répond à des règles précises, donc il est plus simple de ne pas les employer comme premier caractère.
- Les espaces et les signes de ponctuation sont **interdits** (`'`, `?`, etc).
- On **ne peut pas utiliser un mot-clé** du langage comme identificateur. Ainsi, il est interdit de déclarer une variable s'appelant `int` ou `return`, par exemple.

```
1 #include <iostream>
2
3 int main()
4 {
5     // Bon exemple.
6     int avec_underscore { 0 };
7     // Erreur : espace interdit.
8     int avec espace { 0 };
9
10    // Bon exemple.
11    int variable1 { 42 };
12    // Erreur : ne peut pas commencer par un chiffre.
13    int 1variable { 42 };
14
15    // Bon exemple.
16    char lettre { 'A' };
17    // Erreur : ponctuation interdite.
18    char autre_lettre! { 'B' };
19
20    // Bon exemple.
21    double retour { 2.71 };
22    // Erreur : mot-clé réservé par C++.
23    double return { 2.71 };
24
25    return 0;
26 }
```

II.4.2.1.3. De l'importance d'un beau nom

Donner un nom clair et précis à une variable est un grand défi auquel sont confrontés même les programmeurs ayant de l'expérience. En effet, même si un identificateur respecte les règles

II. Le début du voyage

imposées par C++, cela ne veut pas dire que la variable est bien nommée.

- `variable`: ne veut rien dire. Que stocke-t-elle? Quel est son but? Pourquoi existe-t-elle?
- `variable_contenant_la_multiplication_de_pi_par_e_au_cube`: beaucoup trop long.
- `cIrCoNfErEnCe_CeRcLe`: le nom définit bien le pourquoi de l'existence de notre variable, mais le mélange majuscule / minuscule le rend illisible.
- `Brzeczyszczkiewicz`: aucun sens, sauf si vous aimez [l'humour polonais](#) [↗](#).

Avec le temps, en progressant, vous arriverez à trouver plus rapidement et plus facilement des identificateurs qui soient clairs et simples. Dans le cours, je m'efforcerai de le faire afin de vous donner des exemples concrets.

II.4.2.1.4. Sinon, par défaut...

Abordons un dernier point. Je vous ai dit qu'on donne une valeur entre accolades pour initialiser notre variable. Mais que se passe-t-il si on écrit simplement les accolades, sans aucune valeur dedans? Le code va-t-il toujours compiler?

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     int entier {};
7     std::cout << "Mon entier vaut " << entier << std::endl;
8
9     double reel {};
10    std::cout << "Mon réel vaut " << reel << std::endl;
11
12    char lettre {};
13    std::cout << "Mon caractère vaut " << lettre << std::endl;
14
15    std::string phrase {};
16    std::cout << "Ma chaîne vaut " << phrase << std::endl;
17
18    return 0;
19 }
```

```
1 Mon entier vaut 0
2 Mon réel vaut 0
3 Mon caractère vaut
4 Ma chaîne vaut
```

Non, le programme n'a pas de *bugs*, il fonctionne très bien. En fait, si aucune valeur n'est spécifiée, nos variables sont **initialisées à une valeur par défaut**. Pour les entiers et les réels,

II. Le début du voyage

il s'agit de zéro. Pour les caractères, c'est une valeur spéciale signifiant «pas de caractère», de même que pour les chaînes de caractères, initialisées avec du vide.

i

À vous de choisir

C++ vous laisse libre de vos choix. Dans ce cours, pour commencer, nous écrirons toujours les valeurs explicitement. Au fur et à mesure que vous progresserez, nous utiliserons de plus en plus souvent l'initialisation par défaut.

II.4.2.2. Un peu de constance, voyons !

Terminons cette partie en présentant un mot-clé important qui a pour nom `const`. Il permet d'empêcher toute modification de la variable sur laquelle ce mot-clé s'applique. On dit qu'on crée **une constante**. Et si l'on essaye de modifier une constante, le compilateur refuse clair et net.

```
1 int main()
2 {
3     double const pi { 3.141592 };
4     pi = 4; // Ouch, badaboum ça ne compile pas.
5
6     return 0;
7 }
```

```
1 [Visual Studio]
2 C3892 'pi' : vous ne pouvez pas assigner une variable const.
3
4 -----
5
6 [GCC]
7 prog.cc: In function 'int main()':
8 prog.cc:4:8: error: assignment of read-only variable 'pi'
9     4 |     pi = 4; // Ouch, badaboum ça ne compile pas.
10      |     ~~~^~~
11
12 -----
13
14 [Clang]
15 prog.cc:4:8: error: cannot assign to variable 'pi' with
16     const-qualified type 'const double'
17     pi = 4; // Ouch, badaboum ça ne compile pas.
18     ~~ ^
19 prog.cc:3:18: note: variable 'pi' declared const here
20     double const pi { 3.141592 };
```

II. Le début du voyage

```
20 ~~~~~^~~~~~
21 1 error generated.
```

Certains choisissent de les écrire entièrement en majuscule pour bien les différencier des autres variables. Ce n'est absolument pas une obligation et nous n'imposons sur ce point aucune règle.

Quelle en est l'utilité? Il sert pour la qualité du code. Tout ce qui n'est pas destiné à être modifié est ainsi protégé. Cela nous permet également de donner des garanties fortes à notre code. C'est pour ça que nous allons l'utiliser dans tous les exemples. Vous verrez plus tard dans le cours d'autres cas utiles.

i

Ordre du `const`

Vous verrez souvent des codes qui inversent l'ordre de `const` et écrivent `const int constante {};`. Cela est légal et possible en C++, car `const` respecte une règle simple: **il s'applique à ce qui est à sa gauche immédiate**, sauf s'il n'y a rien, auquel cas il s'applique à **ce qu'il y a à droite**. Ici, cela ne change rien, mais plus tard cela aura des conséquences. Nous verrons cela en temps voulu.

II.4.2.3. Manipulation de variables

Nous ne sommes absolument pas obligés de nous contenter d'afficher nos variables. Au contraire, nous pouvons faire beaucoup d'opérations dessus, notamment **combinaison des variables entre elles** et **modifier leur valeur**.

```
1  #include <iostream>
2
3  int main()
4  {
5      int entier { 4 };
6      std::cout << "Mon entier vaut : " << entier << std::endl;
7
8      // Je peux tout à fait changer la valeur de ma variable.
9      entier = 4 * (8 + 9) - 1;
10     std::cout << "Finalement non, il vaut : " << entier <<
        std::endl;
11
12     // Je peux même utiliser la valeur de ma variable et la
        réaffecter à la même variable.
13     entier = entier + 7;
14     std::cout << "Et si j'additionne 7 ? " << entier << std::endl;
15
16     int autre_entier { entier * 2 };
17     // Je peux utiliser d'autres variables également.
```

II. Le début du voyage

```
18     entier = autre_entier - 4;
19     std::cout <<
        "Finalement, je décide que mon entier a une autre valeur qui est : "
        << entier << std::endl;
20
21     return 0;
22 }
```

Pour modifier la valeur d'une variable, on utilise l'opérateur d'affectation `=`, précédé de l'identificateur de la variable et suivi de la valeur à affecter: `identificateur = valeur;`. Et comme vous l'avez noté, cette valeur peut être un simple littéral, un calcul plus complexe, une autre variable voire la variable actuelle elle-même.

En effet, C++ s'occupe d'abord de tout ce qui se trouve **à droite du signe égal**. Ainsi, dans la ligne `entier = entier + 7`, il récupère l'ancienne valeur de `entier`, lui ajoute 7 et assigne ce nouveau résultat à `entier`. Essayez donc pour voir.

```
1  #include <iostream>
2
3  int main()
4  {
5      int entier { 4 };
6      std::cout << "Mon entier vaut : " << entier << std::endl;
7
8      entier = entier + 2;
9      std::cout << "Mon entier vaut : " << entier << std::endl;
10
11     entier = entier - 2;
12     std::cout << "Mon entier vaut : " << entier << std::endl;
13
14     entier = entier * 2;
15     std::cout << "Mon entier vaut : " << entier << std::endl;
16
17     entier = entier / 2;
18     std::cout << "Mon entier vaut : " << entier << std::endl;
19
20     return 0;
21 }
```

Vous souvenez-vous de l'exemple que nous avons vu à la fin de la section sur les littéraux? Nous faisons des multiplications, mais le fait de devoir répéter de nombreuses fois les mêmes instructions était lourd. Tenez, entraînez-vous donc en faisant un programme qui utilise les variables.

II.4.2.3.1. Prenons un raccourci

Les programmeurs étant fainéants par nature, ils n'aiment pas la répétition. Ils ont donc créé des raccourcis pour aller plus vite. Ainsi, plutôt que d'écrire `a = a + b`, on peut écrire directement `a += b`. Cette nouvelle syntaxe s'applique aux opérateurs d'addition `+`, de soustraction `-`, de multiplication `*`, de division `/` et de modulo `%`.

```
1 #include <iostream>
2
3 int main()
4 {
5     int entier { 4 };
6     std::cout << "Mon entier vaut : " << entier << std::endl;
7
8     entier += 2;
9     std::cout << "Mon entier vaut : " << entier << std::endl;
10
11    entier -= 2;
12    std::cout << "Mon entier vaut : " << entier << std::endl;
13
14    entier *= 2;
15    std::cout << "Mon entier vaut : " << entier << std::endl;
16
17    entier /= 2;
18    std::cout << "Mon entier vaut : " << entier << std::endl;
19
20    entier %= 2;
21    std::cout << "Mon entier vaut : " << entier << std::endl;
22
23    return 0;
24 }
```

Pire, non contents de ça, ils ont raccourci encore la syntaxe dans le cas où l'on souhaite ajouter ou soustraire 1 à un nombre. On parle d'**incrément** et de **décrément**. Il suffit de faire ainsi.

- `a++` ou `++a` pour incrémenter de 1 la valeur de la variable.
- `a--` ou `--a` pour décrémenter de 1 la valeur de la variable.

Les deux syntaxes sont quasiment équivalentes, subsiste une différence mineure que je passe sous silence car elle ne nous intéresse pas à l'heure actuelle.

II.4.3. Qu'en déduisez-vous ?

Lorsque nous déclarons une variable, comme vous le savez maintenant, nous précisons, entre autres, son type et sa valeur. Sauf que cette information est parfois redondante. Regardez le code d'illustration ci-dessous.

II. Le début du voyage

```
1 int main()
2 {
3     // Comme 0 est déjà un littéral entier, pourquoi donc préciser
4     // 'int' ?
5     int entier { 0 };
6
7     // Comme 3.1415 est déjà un littéral flottant, pourquoi donc
8     // préciser 'double' ?
9     double reel { 3.1415 };
10
11     return 0;
12 }
```

Le compilateur peut en effet déduire tout seul le type d'une variable et cela grâce à un nouveau mot-clef: **auto**. Celui-ci s'utilise en lieu et place du type de la variable. Notre code précédent peut ainsi être réécrit de la manière qui suit.

```
1 int main()
2 {
3     // Hop, le compilateur déduit que 'entier' est de type 'int'.
4     auto entier { 0 };
5
6     // Hop, le compilateur déduit que 'reel' est de type 'double'.
7     auto reel { 3.1415 };
8
9     return 0;
10 }
```

II.4.3.1. Avec **const**

Tout comme dans le cas d'une variable dont le type est explicitement écrit, si vous voulez déclarer une constante, il suffit de rajouter **const**.

```
1 int main()
2 {
3     auto const constante { 40 };
4     return 0;
5 }
```

II.4.3.2. Le cas de `std::string`

Dans le cas des chaînes de caractères, c'est un peu particulier. Déjà, il faut rajouter un `s` après les guillemets fermants `"`, et ce pour chaque chaîne de caractères que vous voulez écrire. Ensuite, il faut rajouter une ligne spéciale. Nous expliquerons pourquoi plus loin dans ce cours. L'exemple suivant va vous aider à comprendre.

```
1 #include <string>
2
3 int main()
4 {
5     // Écrivez cette ligne une seule fois.
6     using namespace std::literals;
7
8     // Puis vous pouvez déclarer autant de chaînes de caractères
9     // que vous voulez.
10    auto chaine { "Du texte."s };
11    auto autre_chaine { "Texte alternatif."s };
12    auto encore_une_chaine { "Allez, un dernier pour la route."s };
13
14    return 0;
15 }
```



Histoire et héritage

Si nous sommes obligés de rajouter le `s` en fin de littéral, c'est parce que nous avons affaire ici à un **héritage du C**. Sans celui-ci, le compilateur déduit que notre littéral est une chaîne de caractères comme en C et non comme étant une `std::string`. Nous en reparlerons au moment voulu, mais retenez que ce `s` est **très important** dès que vous utilisez `auto`.

II.4.3.3. Les avantages

Utiliser `auto` présente deux avantages majeurs.

- D'abord, utiliser `auto` permet de **raccourcir le code**. Là, remplacer `double` par `auto` n'est pas un gain de place énorme, mais, plus tard dans ce cours, nous verrons des types longs à écrire et lourds à lire, qu'il sera bien plus agréable de remplacer par un concis `auto`.
- Ensuite, cela permet un **code plus évolutif**. Imaginez des calculs complexes avec des types écrits explicitement. Si vous voulez changer l'un des types, il faut en changer chaque occurrence. Avec `auto`, vous laissez au compilateur le soin de s'occuper de cette tâche.



Je suis perdu. Qu'est-ce que je dois choisir entre `auto` ou le type explicite?

II. Le début du voyage

C++ vous autorise à faire les deux, donc **le choix est vôtre**. Certains programmeurs ne jurent que par `auto` et l'utilisent partout. D'autres préfèrent le réserver aux types longs ou complexes et écrire explicitement les types simples et courts. Dans le cadre de ce cours, j'utiliserai les deux façons de faire.

II.4.4. Les entrées

Dès le premier code C++ que vous avez étudié, il y avait la notion de sortie standard. Il est temps de voir le concept inverse en manipulant **l'entrée standard**. Maintenant que nous connaissons le concept des variables, demander des informations à l'utilisateur est à notre portée. Il faut en effet pouvoir stocker quelque part la valeur qu'a tapée l'utilisateur, ce que permettent justement les variables.

Manipuler les entrées est un peu particulier avec Wandbox. Il faut cliquer sur l'encadré `stdin` et taper la valeur que vous souhaitez rentrer. C'est moins pratique qu'avec un IDE. Peu importe ce que vous préférez, l'essentiel est que vous puissiez manipuler pour comprendre la suite du chapitre.

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Entre ton age : " << std::endl;
6     int age { 0 };
7     std::cin >> age;
8     std::cout << "Tu as " << age << " ans.\n";
9
10    return 0;
11 }
```

Avez-vous remarqué à quel point `cin` est semblable à `cout`? Déjà, pour l'utiliser, il faut le préfixer par `std::`, car `cin` est un objet appartenant à la bibliothèque standard. Et on utilise les chevrons dans le sens inverse de `cout`.

II.4.4.1. Gestion des erreurs

Et si vous rentrez une valeur qui n'a rien à voir? Que se passe-t-il si je décide de taper `Maurice` au lieu de `5`? Essayez donc.

```
1 Entre ton age :
2 Maurice
3 Tu as 0 ans.
4 Appuyez sur une touche pour continuer...
```

II. Le début du voyage

Comme on veut récupérer un entier et qu'on a écrit une chaîne de caractères, notre variable `age` n'est pas modifiée et vaut donc toujours `0`, sa valeur par défaut.

?

Donc c'est bon alors? Y'a pas de soucis particulier, puisque `std::cin` ignore nos bêtises.

Eh bien nous allons voir que si. Essayons donc de demander deux informations à l'utilisateur: son âge et son nom.

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::cout << "Entre ton age : ";
7     int age { 0 };
8     std::cin >> age;
9     std::cout << "Tu as " << age << " ans.\n";
10
11     std::cout << "Entre ton nom : ";
12     std::string nom { "" };
13     std::cin >> nom;
14     std::cout << "Tu t'appelles " << nom << ".\n";
15
16     return 0;
17 }
```

Lancez ce code en faisant exprès de taper une chaîne de caractères plutôt qu'un entier. Vous devriez obtenir le même résultat que si dessous.

```
1 Entre ton age : Mishka
2 Tu as 0 ans.
3 Entre ton nom : Tu t'appelle .
4 Appuyez sur une touche pour continuer...
```

Vous n'avez rien eu le temps de taper que le programme avait fini. C'est parce que, lorsque `std::cin` rencontre une erreur (comme le fait de rentrer du texte alors qu'on attend un entier), **il passe dans un état invalide**, tous les caractères invalides restent mémorisés et toutes les utilisations suivantes de `std::cin` sont foireuses.

Comment allons-nous gérer ça? Comment nettoyer `std::cin` s'il a échoué? Le chapitre suivant vous apprendra un mécanisme basique mais puissant, qu'on trouve tout le temps en programmation et qui nous aidera à régler nos problèmes.



Un mot sur Windows

La gestion des accents et autres sous Windows, avec C++, est compliquée. Malheureusement, la seule solution que j'ai à vous offrir est **de vous passer des accents** et autres caractères français spéciaux. Tant pis pour la beauté de la langue française.

Dans le cadre professionnel, l'anglais est majoritairement utilisé, donc il n'y a pas de problème.

II.4.4.2. En résumé

- C++ nous permet de manipuler des caractères simples, des chaînes de caractères, des nombres entiers et des nombres réels.
- Il existe des caractères spéciaux, comme le retour à la ligne (`'\n'`) ou le caractère nul (`'\0'`).
- Les variables nous permettent d'associer un nom à une valeur, mais il faut respecter certaines règles.
- On peut utiliser `auto` pour laisser le compilateur déduire tout seul le type d'une expression.
- Nous pouvons demander des informations à l'utilisateur grâce à `std::cin`.
- Nous n'avons aucun mécanisme de protection s'il rentre n'importe quoi.

Contenu masqué

Contenu masqué n°2

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Bonne journée à toi lecteur." << std::endl;
6     return 0;
7 }
```

[Retourner au texte.](#)

Contenu masqué n°3

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Dossier principal : C:\Program Files (x86)" <<
6         std::endl;
7     return 0;
8 }
```

[Retourner au texte.](#)

Contenu masqué n°4

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Voici un exemple sans échappement : " << "'" <<
6         " est bien affiché." << std::endl;
7     std::cout << "Maintenant, je vais l'échapper : " << '\\' <<
8         " est bien affiché grâce à l'échappement." << std::endl;
9     std::cout << "La suite, après la tabulation : \tJe suis loin."
10         << std::endl;
11     std::cout <<
12         "Ces mots sont sur une ligne.\nEt ceux là sur la suivante.\n";
13     return 0;
14 }
```

[Retourner au texte.](#)

Contenu masqué n°5

```
1 #include <iostream>
2
3 int main()
4 {
```

II. Le début du voyage

```
5   int nombre { 1 };
6   int const multiplicateur { 2 };
7
8   nombre = nombre * multiplicateur;
9   std::cout << "Maintenant, mon nombre vaut " << nombre <<
    std::endl;
10
11  nombre = nombre * multiplicateur;
12  std::cout << "Maintenant, mon nombre vaut " << nombre <<
    std::endl;
13
14  nombre = nombre * multiplicateur;
15  std::cout << "Maintenant, mon nombre vaut " << nombre <<
    std::endl;
16
17  return 0;
18 }
```



Vous remarquerez qu'on répète quand même plusieurs fois les mêmes instructions. Ce n'est pas grave. Nous verrons très bientôt comment remédier à ce problème. L'essentiel est que vous ayez pu voir l'avantage d'utiliser des variables.

[Retourner au texte.](#)

II.5. Le conditionnel conjugué en C++

«*Le conditionnel est un mode employé pour exprimer un état soumis à une condition.*» Telle est la définition qu'en donne [Wikipédia](#). Cela vous rappelle-t-il vos cours de français? Mais quel peut bien être le rapport avec la programmation? C'est que **C++ permet aussi d'exprimer des conditions** et donc de modifier le comportement de notre programme.

Ce chapitre va donc introduire **les conditions** à notre palette d'outils et permettra de résoudre la problématique soulevée en conclusion du chapitre précédent.

II.5.1. Les booléens

Les conditions en C++ sont régies par un principe simple, qui est qu'**une condition est soit vraie, soit fausse**. Finalement, c'est un peu comme dans la réalité. Si je vous pose la question «Êtes-vous majeurs?», la réponse est soit oui, soit non. De même, «Faîtes-vous 1m80 ou plus?» entraînera soit une réponse positive, soit négative.

C++ nous offre un type conçu exprès pour ça, `bool`. Ce type peut prendre deux valeurs: soit `true`, signifiant vrai, soit `false` qui veut dire faux. Voyez par vous-mêmes l'exemple on ne peut plus bête ci-dessous.

```
1 int main()
2 {
3     bool const vrai { true };
4     bool const faux { false };
5
6     return 0;
7 }
```



D'accord, mais ça sert à quoi ça? Je vois l'intérêt de manipuler des entiers ou du texte, mais stocker juste vrai ou faux, ça sert quand?

L'intérêt semble en effet très faible si l'on se contente du code précédent. Sachez que les booléens sont partout, utilisables grâce à ce que l'on appelle **les opérateurs de comparaisons**. En voici la liste juste ci-dessous.

Opérateur	Signification	Exemple
<code>==</code>	Égalité, compare si deux variables sont égales entre elles.	<code>a == b</code>

II. Le début du voyage

<code>!=</code>	Inégalité , compare si deux variables sont de valeurs différentes.	<code>a != b</code>
<code><</code>	Strictement inférieur , compare si la variable de gauche est strictement inférieure à celle de droite.	<code>a < b</code>
<code><=</code>	Inférieur ou égale , compare si la variable de gauche est inférieure ou égale à celle de droite.	<code>a <= b</code>
<code>></code>	Strictement supérieur , compare si la variable de gauche est strictement supérieure à celle de droite.	<code>a > b</code>
<code>>=</code>	Supérieur ou égale , compare si la variable de gauche est supérieure ou égale à celle de droite.	<code>a >= b</code>



Opérateur d'égalité

Non, `==` n'est pas une erreur. Il faut bien deux signes `=`, pour différencier avec l'opération consistant à modifier la valeur d'une variable.

Le rapport avec les booléens? Chacune des expressions suivantes renvoie `true` ou `false`. Faisons donc un petit test.

```
1 #include <iostream>
2
3 int main()
4 {
5     int const a { 10 };
6     int const b { 20 };
7
8     // Cette directive nous permet d'afficher true ou false. Par
9     // défaut, std::cout affiche 1 ou 0.
10    std::cout << std::boolalpha;
11
12    std::cout << "a == b donne " << (a == b) << std::endl;
13    std::cout << "a != b donne " << (a != b) << std::endl;
14    std::cout << "a < b donne " << (a < b) << std::endl;
15    std::cout << "a <= b donne " << (a <= b) << std::endl;
16
17    // On peut tout à fait stocker le résultat dans une variable
18    // booléenne.
19    bool const plus_grand { a > b };
20    std::cout << "a > b donne " << plus_grand << std::endl;
21
22    bool const plus_grand_ou_egal { a >= b };
23    std::cout << "a >= b donne " << plus_grand_ou_egal <<
24    std::endl;
25
26    return 0;
27 }
```

II. Le début du voyage

Lancez donc ce code, analysez les résultats. Sont-ils logiques? Maintenant, changez la valeur des variables et observez de nouveau. Est-ce que c'est toujours logique? Voyez-vous maintenant un peu plus l'utilité des booléens? 🍏

II.5.2. if— Si, et seulement si...

Maintenant, examinons la façon de poser une question en C++, si l'on peut dire. Nous allons en effet découvrir la première structure de contrôle: `if`. Ce mot-clef est un mot anglais signifiant «*si*», et exécute des instructions si, et seulement si la condition donnée est vraie. Voyez donc ce schéma.

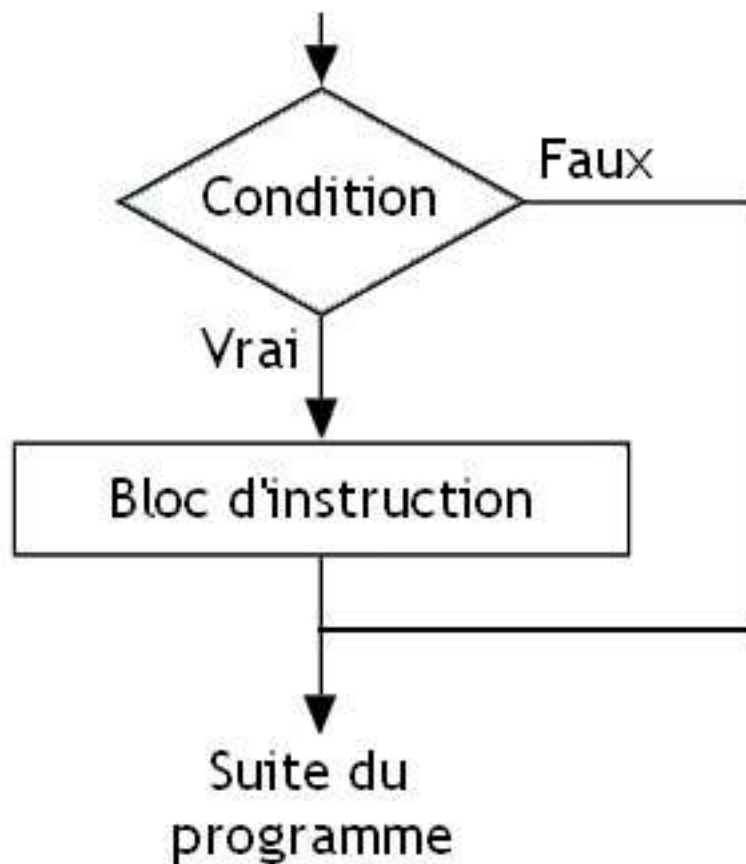


FIGURE II.5.1. – Image originale tirée du tutoriel sur [le langage C](#) .

En C++, voici à quoi ressemble cette instruction. Toutes les instructions entre accolades seront exécutées si `condition` est vraie.

```
1 if (/* condition */)
2 {
3     // Code à exécuter si la condition est vraie.
4 }
```


II. Le début du voyage

Prenons un exemple. Affichons la note à un examen, rentrée par l'utilisateur. Si celle-ci est supérieure ou égale à 16, nous afficherons un message de félicitations.

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Indique-moi ta note : ";
6
7     int note { 0 };
8     std::cin >> note;
9
10    std::cout << "Tu as obtenu " << note << std::endl;
11
12    if (note >= 16)
13    {
14        std::cout << "Félicitations, c'est une très bonne note !"
15            << std::endl;
16    }
17
18    return 0;
19 }
```

Faites-le tourner. Vous verrez, le message de félicitations ne s'affiche que si vous rentrez un nombre supérieur ou égal à 16. Essayez avec 15, vous ne verrez plus le message.

II.5.2.1. À portée

Nous pouvons également **déclarer des variables** au sein du bloc d'accolades. Cependant, celles-ci ne sont utilisables que **jusqu'à l'accolade fermante correspondante**. Ainsi, le code ci-dessous n'est pas correct.

```
1 #include <iostream>
2
3 int main()
4 {
5     if (true)
6     {
7         int const age { 42 };
8         // D'autres instructions.
9
10        // Aucun problème.
11        if (age == 42)
12        {
13            // Ici non plus.
14        }
15    }
16 }
```

II. Le début du voyage

```
15     }
16
17     // NON ! Le compilateur ne connaît pas la variable age.
18     std::cout << age << std::endl;
19
20     return 0;
21 }
```

Ce sont les règles de C++ qui veulent ça. Quand on écrit un nouveau bloc à base d'accolades, on dit qu'on crée **une nouvelle portée**. Et une variable n'est utilisable que dans la portée, ou le bloc d'accolade, où elle a été déclarée.

C'est **une bonne pratique de déclarer ses variables dans la plus petite portée possible**. Il y a plusieurs raisons à ça.

- Dans le cas d'un code long et complexe, déclarer une variable au plus près possible de son utilisation permet de ne pas avoir à parcourir de longues lignes pour en obtenir des détails (quel est son type, sa valeur de base, etc). Cela aide à **la lecture et la compréhension du code**.
- Lorsqu'on atteint la fin du bloc correspondant, le programme **libère dans la mémoire les emplacements qu'il avait réservés** pour les variables qui s'y trouvaient. Dans le cas d'un `int` comme ici, ça ne change rien. Mais dans le cas de variables plus complexes que nous verrons plus tard, cela peut avoir une grande incidence sur les performances du programme.

Ainsi, je vous encourage fortement à déclarer vos variables à l'intérieur de vos blocs de condition **si elles ne sont pas destinées à être utilisées ailleurs**.

II.5.3. else—Sinon...

Maintenant que nous avons vu comment changer le code si une condition est vérifiée, voyons comment faire l'opposé, c'est-à-dire comment agir si celle-ci est fausse. Imaginons un programme demandant si l'utilisateur est majeur ou mineur. On peut imaginer quelque chose comme ce qui suit.

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Donne-moi ton âge: ";
6      int age { 0 };
7      std::cin >> age;
8
9      if (age >= 18)
10     {
11         std::cout << "Tu es majeur." << std::endl;
```

II. Le début du voyage

```
12     }
13
14     if (age < 18)
15     {
16         std::cout << "Tu es mineur." << std::endl;
17     }
18
19     return 0;
20 }
```

Ce programme est fonctionnel mais un peu lourd, puisqu'on répète deux fois une condition quasiment identique. Heureusement, C++ nous offre un autre mot-clef, **else**, qui exécute des instructions si la condition du **if** est fausse. Voyez-vous mêmes le code ci-dessous.

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Donne-moi ton âge: ";
6      int age { 0 };
7      std::cin >> age;
8
9      if (age >= 18)
10     {
11         std::cout << "Tu es majeur." << std::endl;
12     }
13     else
14     {
15         std::cout << "Tu es mineur." << std::endl;
16     }
17
18     return 0;
19 }
```

Le code se lit ainsi en français: «*Si l'âge rentré par l'utilisateur est supérieur ou égal à 18 ans, alors afficher "Tu es majeur", sinon afficher "Tu es mineur".*» Le mot-clef **else** est en effet un mot d'anglais qui existe et qui signifie, comme vous l'avez compris, «*sinon*».

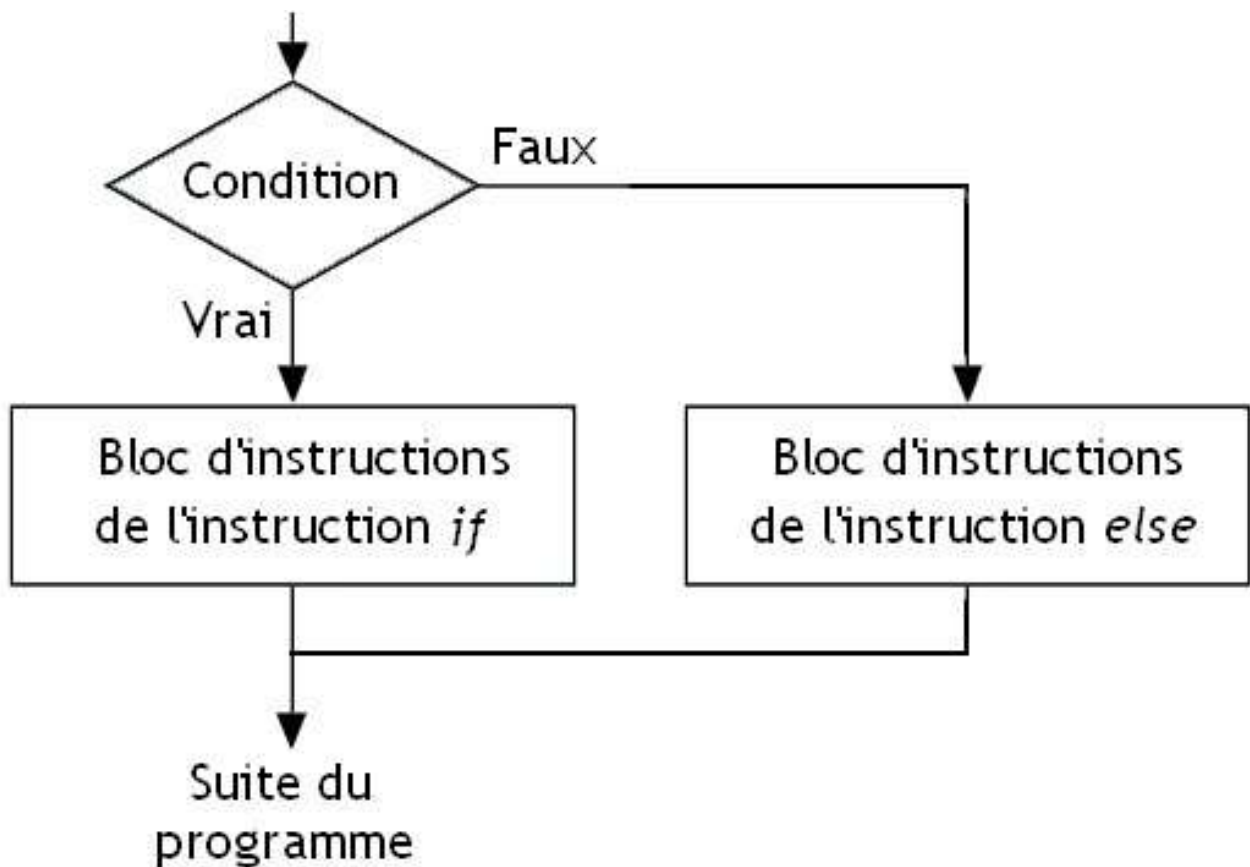


FIGURE II.5.2. – Image originale tirée du [tutoriel sur le langage C](https://zestedesavoir.com/contenus/755/le-langage-c-1/1042_les-bases-du-langage-c/4294_les-selections/#1-12882_la-structure-if).

Notez bien que, contrairement au `if`, `else` n’a pas de parenthèse pour préciser une condition. En effet, rappelons que `else` signifie «tout le reste». Y’a t-il donc un moyen de tester plusieurs conditions différentes avant de faire «tout le reste»? Sommes-nous condamnés à utiliser une suite de `if` et de `else`?

II.5.4. `else if`— La combinaison des deux précédents

Comme vous l’avez vu en lisant le titre de cette section, C++ est un langage bien conçu qui répond à la problématique de la section précédente. La combinaison `else if` s’utilise en effet entre un `if` et un `else` pour dire «ou si cette condition est vraie.» Voyons déjà à quoi ressemble la syntaxe pour vous donner une idée.

```
1 if (/* condition 1 */)
2 {
3     // Si
4     // Code à exécuter si la condition 1 est vraie.
5 }
6 else if (/* condition 2 */)
```

II. Le début du voyage

```
7 {
8     // Ou si.
9     // Code à exécuter si la condition 1 est fausse mais que la
    condition 2 est vraie.
10 }
11 else if (/* condition 3 */)
12 {
13     // Ou si.
14     // Code à exécuter si ni la condition 1 ni la condition 2
    ne sont vraies mais que la condition 3 est vraie.
15 }
16 else
17 {
18     // Sinon.
19     // Code à exécuter si aucune condition n'est vraie.
20 }
```

Vous voyez le principe? Bien, passons donc à un exercice concret. Imaginons un programme qui affiche des messages différents en fonction de l'âge de l'utilisateur. Mettez le texte et choisissez les âges que vous souhaitez. Une solution parmi d'autre se trouve ci-dessous.

👁 Contenu masqué n°6

II.5.5. Conditions imbriquées

Il est tout à fait possible de **tester des conditions au sein d'autres conditions**. Si l'on prend l'exemple d'un menu de restaurant, on peut imaginer demander au client s'il veut de la viande ou un plat végétarien. Dans le premier cas, il faut lui demander la cuisson souhaitée; dans le deuxième, on lui laisse le choix du légume. Cela est tout à fait possible en C++.

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Bienvenue au restaurant \"Chez Clem\"." <<
        std::endl;
6
7     bool const plat_vegetarien { true };
8
9     if (plat_vegetarien)
10    {
11        bool const salade { true };
12        if (salade)
13        {
```

II. Le début du voyage

```
14         std::cout << "Salade grecque ? De pâtes ?" <<
15             std::endl;
16     }
17     else
18     {
19         std::cout <<
20             "Nous avons également de la purée, une macédoine de légumes, un
21             << std::endl;
22     }
23     }
24     else
25     {
26         bool const viande { true };
27         if (viande)
28         {
29             std::cout << "Bleue, saignante, à point, bien cuite ?"
30             << std::endl;
31         }
32         else
33         {
34             std::cout <<
35                 "Sinon niveau poisson il y a pavé de saumon, filet de cabillaud
36                 << std::endl;
37         }
38     }
39     }
40     return 0;
41 }
```



Instruction condensée

L'instruction `else if`, vue plus haut, correspond en fait à un `if` dans un `else`.



```

1  if (/* condition 1 */)
2  {
3  }
4  else if (/* condition 2 */)
5  {
6  }
7
8  // Équivaut à ceci.
9
10 if (/* condition 1 */)
11 {
12 }
13 else
14 {
15     if (/* condition 2 */)
16     {
17     }
18 }

```

II.5.6. [T.P] Gérer les erreurs d'entrée— Partie I

Vous souvenez-vous de la problématique du chapitre précédent? Nous avons des problèmes quand l'utilisateur rentrait du texte à la place d'un entier, par exemple. *Eh bien nous pouvons déjà apporter une première solution* à ce problème, solution que nous améliorerons au fur et à mesure.

Les développeurs ayant écrit la bibliothèque standard C++ ont déjà tout prévu. Trois fonctions, déjà programmées, vont nous intéresser.

- `std::cin > x` est en fait une fonction qui renvoie `true` si tout est correct ou `false` si on a rencontré une erreur lors de la saisie.
- `std::cin.clear()` restaure `std::cin` à un état fonctionnel, sans erreur.
- `std::cin.ignore()` permet d'ignorer un nombre défini de caractères, soit jusqu'à un nombre maximum (exemple 500), soit jusqu'à un caractère précis (exemple `'\n'` ou `'a'`). Dans notre cas, nous allons utiliser ceci:

```
1 std::cin.ignore(255, '\n');
```

Maintenant, vous avez toutes les informations nécessaires pour commencer la protection de nos entrées. À vous de jouer!

Voilà, nous avons déjà une petite protection. Mais elle ne marche qu'une fois: impossible de la faire marcher tant que l'utilisateur n'a pas rentré quelque chose de valide. Pour l'instant, laissez ça de côté, nous en reparlerons dans le prochain chapitre.

II.5.7. La logique booléenne

Nous avons vu les différents moyens de tester des conditions, ainsi que différents opérateurs de comparaisons qui retournent des booléens. Mais ce ne sont pas les seuls. Cette section va introduire [l'algèbre booléenne](#) [↗](#), et rassurez-vous, c'est simple à comprendre, puissant et bien utile.

II.5.7.1. AND—Tester si deux conditions sont vraies

Imaginons un instant que vous souhaitiez faire l'acquisition d'une voiture. Pour cela, il vous faut remplir deux conditions: avoir le permis et suffisamment d'argent. Si l'une des deux conditions n'est pas remplie, ou les deux, vous continuerez à prendre le bus. Le seul moyen de faire hurler le moteur sur l'autoroute, c'est d'avoir assez d'argent **ET** d'avoir le permis.

C++ nous fournit deux moyens d'exprimer cet opérateur logique «ET»: `&&` et `and`. Les deux sont parfaitement valables et interchangeables. Le deuxième était notamment prévu à l'époque où tous les claviers ne possédaient pas le symbole `&`.



Ancienne spécificité Microsoft

Si `and` ne marche pas avec Visual Studio, il faut inclure le fichier `<ciso646>`. Avec une version à jour, ce n'est cependant plus nécessaire.

```
1  #include <iostream>
2
3  int main()
4  {
5      int const prix_voiture { 5000 };
6      int const argent { 2000 };
7      bool const a_le_permis { true };
8
9      if (argent >= prix_voiture && a_le_permis)
10     {
11         std::cout << "Voici les clés, bonne route." << std::endl;
12     }
13     else
14     {
15         std::cout << "Désolé, vous allez devoir prendre le bus." <<
16             std::endl;
17     }
```


II. Le début du voyage

```
18     return 0;
19 }
```

Testez donc ce programme en modifiant le prix de la voiture, l'argent disponible ou la possession ou non du permis. Comme écrit plus haut, il n'y a que si `a_le_permis` est à `true` et qu'il y a plus d'argent que le prix de la voiture que vous serez en mesure d'avoir les clés.

Voici ce qu'on appelle la [table de vérité](#) de l'opérateur AND, qui formalise les entrées et les sorties de cet opérateur. N'hésitez pas à en lire plus sur le sujet s'il vous intéresse. 🍊

Première condition	Seconde condition	Résultat de l'opérateur «AND»
false	false	false
false	true	false
true	false	false
true	true	true

II.5.7.2. OR—Tester si au moins une condition est vraie

Maintenant, nous allons aller visiter un musée. Quand on arrive devant l'entrée, bonne surprise, l'entrée est gratuite si l'on a moins de 26 ans **OU** si l'on est professeur (peu importe la matière). Il suffit de remplir l'une des deux conditions pour être admis gratuitement à l'intérieur. Bien sûr, si vous remplissez les deux, l'offre reste valable.

Cet exemple va servir de base pour l'introduction de l'opérateur C++ «OU»: au choix, `||` (qui se tape sur un AZERTY français en faisant `Alt Gr` + `-` (le numéro 6) ou `or`, présent pour la même raison que précédemment.



Ancienne spécificité Microsoft

Si `or` ne marche pas avec Visual Studio, il faut inclure le fichier `<ciso646>`. Avec une version à jour, ce n'est cependant plus nécessaire.

```
1  #include <iostream>
2
3  int main()
4  {
5      int const age { 25 };
6      bool const est_professeur { true };
7
8      if (age <= 26 || est_professeur)
9      {
10         std::cout << "Bonne visite." << std::endl;
11     }
```

II. Le début du voyage

```
12     else
13     {
14         std::cout << "Désolé, il va falloir payer." << std::endl;
15     }
16
17     return 0;
18 }
```

Encore une fois, faites votre ce programme et modifiez-le. Voyez dans quelles circonstances l'entrée est gratuite et à quels moments il faut payer. Vous allez obtenir des résultats identiques à la table de vérité ci-dessous.

Première condition	Seconde condition	Résultat de l'opérateur «OR»
false	false	false
false	true	true
true	false	true
true	true	true

II.5.7.3. NOT—Tester la négation

Il fait chaud, vous allez à la piscine vous rafraîchir. En arrivant, vous jetez un œil au règlement et vous voyez qu'il est interdit de nager si l'on a un short de bain et non un maillot de bain. Vous raisonnez donc ainsi: *«Si j'ai un maillot de bain, alors tout va bien, l'entrée m'est autorisée, je peux nager. Si j'ai un short de bain, alors l'entrée m'est interdite et je dois en acheter un avant de nager.»* Comment exprimeriez-vous ça en C++? Peut-être avec le code ci-dessous?

```
1  int main()
2  {
3      bool const ai_je_un_maillot { true };
4      if (ai_je_un_maillot)
5      {
6          // Je n'ai rien à faire, tout est OK.
7      }
8      else
9      {
10         // Je dois aller acheter un maillot.
11     }
12
13     // Je nage et je suis heureux.
14     return 0;
15 }
```

Ce code fonctionne, mais il nous pose un problème: on a un `if` vide et c'est moche. L'idéal

II. Le début du voyage

serait de ne traiter que le cas «Pas de maillot de bain». C++ nous permet justement cela avec l'opérateur de négation `!` ou `not`. Celui-ci renvoie la valeur inverse de l'expression testée. Si l'expression vaut `true`, alors cet opérateur renverra `false`; si, au contraire, elle valait `false`, sa négation donnera `true`.



Ancienne spécificité Microsoft

Si `not` ne marche pas avec Visual Studio, il faut inclure le fichier `<ciso646>`. Avec une version à jour, ce n'est cependant plus nécessaire.

```
1 int main()
2 {
3     bool const ai_je_un_maillot { true };
4     if (!ai_je_un_maillot)
5     {
6         // Je dois aller acheter un maillot.
7     }
8
9     // Je nage et je suis heureux.
10    return 0;
11 }
```

Notre code devient plus court, plus concis et plus agréable à lire, il gagne en qualité. Je vous mets en dessous la table de vérité, bien plus courte cette fois.

Valeur de la condition	Résultat de l'opérateur «NOT»
true	false
false	true



Il y a juste quelque chose que je ne comprends pas. Pourquoi ne pas créer tout simplement un booléen inverse, genre `pas_de_maillot`? Pourquoi se casser la tête à tester la négation d'une condition et ne pas tester la condition inverse tout de suite?

C'est une très bonne question. Dans un exemple aussi simpliste que celui-ci, monté de toute pièce, on aurait pu, c'est vrai. Mais d'autres fois nous n'avons pas le choix, car nous utilisons du code qui a été programmé d'une certaine façon.

II.5.8. Tester plusieurs expressions

Nous ne sommes absolument pas limités à tester deux expressions en même temps. On peut le faire avec trois, quatre, dix, soixante (bien que soixante soit un exemple extrême et vraiment trop peu lisible). Le code suivant le prouve.

II. Le début du voyage

```
1 #include <iostream>
2
3 int main()
4 {
5     bool const ai_je_envie_de_nager{ true };
6     bool const ai_je_un_maillot { true };
7     int const argent { 20 };
8     int const prix { 5 };
9
10    if (ai_je_envie_de_nager && ai_je_un_maillot && argent >= prix)
11    {
12        std::cout << "PLOUF !" << std::endl;
13    }
14    else
15    {
16        std::cout << "Huum, un autre jour." << std::endl;
17    }
18
19    return 0;
20 }
```

Dans le cas du même opérateur, ils sont évalués de gauche à droite. C'est le cas du code précédent. Mais que se passe-t-il si l'on mélange plusieurs opérateurs différents dans la même expression? Sauriez-vous dire ce que quelque chose comme `expression1 && expression2 || expression3` va donner? En effet, les opérateurs logiques sont comme les opérateurs mathématiques que nous avons vus dans les chapitres précédents: **ils ont une priorité**.

1. Le plus prioritaire, c'est la négation `!`.
2. Ensuite vient le «ET» `&&`.
3. Enfin, le «OU» `||` est le moins prioritaire.

Ainsi, dans l'exemple `!a && b`, c'est d'abord `!a` qui est évalué, puis la nouvelle valeur et `b` sont testées avec l'opérateur ET `&&`. Avec le code `a && b || c && d`, dans l'ordre, on évalue `a && b`, `c && d` et enfin `a && b || c && d`.

Bien que la priorité des opérateurs soit clairement définie par C++, une bonne pratique consiste à **ajouter des parenthèses** autour des expressions pour rendre le **code plus lisible et plus clair**. Si l'on reprend nos exemples, cela donne `(!a) && b` pour le premier et `(a && b) || (c && d)` pour le deuxième. C'est clair? Prouvez-le-moi en mettant les parenthèses au bon endroit dans ces codes (piqués à @gbdivers).

```
1 a && b && c
2 a || b || c
3 a || b && c
4 a && b || c
5
```

II. Le début du voyage

```
6  !a && b
7  a || !b
8
9  a && b || c && d
10 a || b && c || d
```

👁 Correction

i

Parenthèses

À partir de maintenant, je vais mettre des parenthèses dans le code pour le rendre clair et explicite.

II.5.8.1. Évaluation en court-circuit

Imaginez un formulaire à remplir, avec de multiples questions, pour demander une place de parking dans votre immeuble. Pour ça, il faut déjà habiter l'immeuble en question et avoir une voiture. Si vous habitez ailleurs, pas la peine de réfléchir à la possession ou non d'une voiture, vous êtes inéligible pour cette place de parking.

Eh bien le compilateur mène le même raisonnement quand il évalue une expression constituée de «AND». S'il détecte qu'une condition est fausse, alors **il ne sert à rien d'évaluer le reste de l'expression** puisque le résultat est forcément faux. Si vous ne voyez pas pourquoi, retournez un peu plus haut jeter un œil à la table de vérité de «AND». 🍊

```
1  #include <iostream>
2
3  int main()
4  {
5      bool const ai_je_une_voiture { false };
6      bool const habite_immeuble { true };
7
8      // Comme ai_je_une_voiture vaut false, le compilateur ne va pas
9      // tester l'expression habite_immeuble.
10     if (ai_je_une_voiture && habite_immeuble)
11     {
12         std::cout << "Voici votre place de parking." << std::endl;
13     }
14     else
15     {
16         std::cout << "Désolé, vous n'êtes pas éligible." <<
17         std::endl;
18     }
19 }
```

II. Le début du voyage

```
18     return 0;  
19 }
```

Avec l'opérateur «OR», le même principe s'applique: si l'une des expressions est évaluée à `true`, alors, selon la table de vérité de l'opérateur «OR», le résultat sera `true`, donc pas la peine d'évaluer le reste.

Ce principe d'optimisation par le compilateur s'appelle **l'évaluation en court-circuit**. Il permet d'optimiser l'exécution du code en ne perdant pas de temps sur des instructions qui seront forcément évaluées à `false` ou à `true`. Quand nous avancerons dans notre apprentissage du C++, nous verrons des cas où l'évaluation en court-circuit est bien utile.

II.5.9. Exercices

Allez, c'est l'heure de pratiquer un peu. C'est le meilleur moyen de progresser et de fixer toutes ces nouvelles notions dans votre mémoire. N'hésitez pas à faire cette section à tête reposée et avec le cours sous les yeux en même temps.

i

Un mot sur la licence

Certains exercices sont adaptés du [cours sur le langage C](#) , comme m'y autorisent la licence et mon statut de double auteur. 🍊

II.5.9.1. Une pseudo-horloge

Demandons à l'utilisateur de rentrer un nombre et nous lui afficherons la période de la journée correspondante: nuit, matin, après-midi, soir. Ainsi, entre 8h et 12h, nous sommes le matin. Mais attention, nous voulons aussi gérer les cas un peu spéciaux que sont midi, minuit et un nombre qui n'est pas entre 0 et 24.

👁 Correction pseudo-horloge

II.5.9.2. Score

Imaginez que vous avez un score de jeu vidéo sous la main.

- Si le score est strictement inférieur à deux mille, affichez «C'est la catastrophe!»
- Si le score est supérieur ou égal à deux mille et que le score est strictement inférieur à cinq mille, affichez: «Tu peux mieux faire!»
- Si le score est supérieur ou égal à cinq mille et que le score est strictement inférieur à neuf mille, affichez: «Tu es sur la bonne voie!»
- Sinon, affichez: «Tu es le meilleur!»

👁 Correction score

II.5.9.3. XOR— Le OU exclusif

Le OR que nous avons vu plus tôt dans ce chapitre est dit «inclusif». Ainsi, `a || b` signifie `a`, ou `b`, ou les deux. Le but de cet exercice va être de faire un OU dit «exclusif», c'est-à-dire que si `a` et `b` renvoient la même chose, l'expression est évaluée à `false`, sinon à `true`. Vous pouvez le faire avec les opérateurs que nous avons vus.

👁 Indice

👁 Correction XOR

II.5.9.4. En résumé

- Nous pouvons tester une expression avec `if`. Si celle-ci est fausse, on peut tester d'autres expressions avec autant de `else if` qu'on veut. Enfin, `else` permet de tester le cas général, «tout le reste».
- Nous pouvons comparer des expressions entre elles avec les opérateurs de comparaisons.
- Les opérateurs logiques permettent de tester plusieurs expressions en même temps ou d'en obtenir la négation.
- Le compilateur optimise l'évaluation de ces multiples expressions grâce à l'évaluation en court-circuit.

Contenu masqué

Contenu masqué n°6

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Quel âge as-tu ? ";
6
7     unsigned int age { 0 };
8     std::cin >> age;
9 }
```

II. Le début du voyage

```
10  if (age < 5u)
11  {
12      std::cout <<
          "Tu n'es pas un peu jeune pour naviguer sur internet ?"
          << std::endl;
13  }
14  else if (age < 18u)
15  {
16      std::cout << "Tu es mineur !" << std::endl;
17  }
18  else if (age < 30u)
19  {
20      std::cout << "Vous êtes majeur !" << std::endl;
21  }
22  else if (age < 50u)
23  {
24      std::cout <<
          "Vous n'êtes plus très jeunes, mais vous n'êtes pas encore vieux."
          << std::endl;
25  }
26  else if (age < 70u)
27  {
28      std::cout << "Vous commencez à prendre de l'âge." <<
          std::endl;
29  }
30  else if (age < 120u)
31  {
32      std::cout <<
          "Vous avez du en voir, des choses, durant votre vie !"
          << std::endl;
33  }
34  else
35  {
36      std::cout << "Quelle longévité !" << std::endl;
37  }
38
39  return 0;
40 }
```

[Retourner au texte.](#)

Contenu masqué n°7 :

Correction T.P Partie I

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::cout << "Entre ton age : ";
7     unsigned int age { 0 };
8
9     if (std::cin >> age)
10    {
11        // Tout va bien.
12        std::cout << "Tu as " << age << " ans.\n";
13    }
14    else
15    {
16        // Si std::cin a rencontré une erreur quelconque.
17        std::cout <<
18            "Tu n'as pas rentré un entier, il y a eu une erreur."
19            << std::endl;
20        std::cin.clear(); // On remet std::cin dans un état
21                           fonctionnel.
22        std::cin.ignore(255, '\n'); // On vide les caractères
23                                   mémorisés.
24    }
25
26    std::cout << "Maintenant, vérifions que tout va bien." <<
27        std::endl;
28    std::cout << "Entre ton nom : ";
29    std::string nom { "" };
30    std::cin >> nom;
31    std::cout << "Tu t'appelles " << nom << ".\n";
32
33    return 0;
34 }
```

Si vous avez eu du mal ou n'avez pas trouvé, ce n'est pas grave, vous êtes en train d'apprendre. N'hésitez pas à relire la solution jusqu'à ce que tout vous paraisse limpide. [Retourner au texte.](#)

Contenu masqué n°8 :

Correction

Voici la solution. L'avez-vous trouvée? Vous êtes sur la bonne voie! Vous avez-eu du mal? N'hésitez pas à relire les passages précédents à tête reposée et à faire vos propres exemples.

II. Le début du voyage

```
1 (a && b) && c
2 (a || b) || c
3 a || (b && c)
4 (a && b) || c
5
6 (!a) && b
7 a || (!b)
8
9 (a && b) || (c && d)
10 (a || (b && c)) || d
```

[Retourner au texte.](#)

Contenu masqué n°9 : Correction pseudo-horloge

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Quelle heure est-il ?" << std::endl;
6     int heure { 0 };
7     std::cin >> heure;
8
9     if (heure > 0 && heure < 7)
10    {
11        std::cout << "Zzz..." << std::endl;
12    }
13    else if (heure >= 7 && heure < 12)
14    {
15        std::cout << "C'est le matin !" << std::endl;
16    }
17    else if (heure == 12)
18    {
19        std::cout << "Il est midi !" << std::endl;
20    }
21    else if (heure > 12 && heure < 18)
22    {
23        std::cout << "C'est l'après-midi !" << std::endl;
24    }
25    else if (heure >= 18 && heure < 24)
26    {
27        std::cout << "C'est le soir !" << std::endl;
28    }
29    else if (heure == 24 || heure == 0)
```

II. Le début du voyage

```
30 {
31     std::cout << "Il est minuit, dormez brave gens !" <<
        std::endl;
32 }
33 else
34 {
35     std::cout <<
        "Il est l'heure de réapprendre à lire l'heure !" <<
        std::endl;
36 }
37
38 return 0;
39 }
```

[Retourner au texte.](#)

Contenu masqué n°10 : Correction score

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Quel est le score du joueur ? " << std::endl;
6     int score { 0 };
7     std::cin >> score;
8
9     if (score < 2000)
10    {
11        std::cout << "C'est la catastrophe !" << std::endl;
12    }
13    else if (score >= 2000 && score < 5000)
14    {
15        std::cout << "Tu peux mieux faire !" << std::endl;
16    }
17    else if (score >= 5000 && score < 9000)
18    {
19        std::cout << "Tu es sur la bonne voie !" << std::endl;
20    }
21    else
22    {
23        std::cout << "Tu es le meilleur !" << std::endl;
24    }
25
26    return 0;
27 }
```

[Retourner au texte.](#)

Contenu masqué n°11 : Indice

Si vous bloquez, voici déjà la table de vérité de l'opérateur.

Première condition	Seconde condition	Résultat de l'opérateur «XOR»
false	false	false
false	true	true
true	false	true
true	true	false

[Retourner au texte.](#)

Contenu masqué n°12 : Correction XOR

La solution s'écrit ainsi: `(a && !b) || (b && !a)`. Vous pouvez vérifier avec le programme ci-dessous.

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << std::boolalpha;
6
7     bool const deux_false { (false && !false) || (false && !false)
8         };
9     std::cout << "(false && !false) || (false && !false) == " <<
10         deux_false << std::endl;
11
12     bool const true_false { (true && !false) || (false && !true) };
13     std::cout << "(true && !false) || (false && !true) == " <<
14         true_false << std::endl;
15
16     bool const false_true { (false && !true) || (true && !false) };
17     std::cout << "(false && !true) || (true && !false) == " <<
18         false_true << std::endl;
19
20     bool const deux_true { (true && !true) || (true && !true) };
21     std::cout << "(true && !true) || (true && !true) == " <<
22         deux_true << std::endl;
```

II. Le début du voyage

```
18  
19     return 0;  
20 }
```

Sinon vous pouvez utiliser `xor`, si vous incluez bien `<ciso646>` avec Visual Studio. 🍊 [Retourner au texte.](#)

II.6. Des boucles qui se répètent, répètent, répètent...

Nous sommes maintenant capables d'exécuter des codes différents en fonction de conditions. Mais notre programme reste très linéaire: nous exécutons les instructions l'une après l'autre, du début à la fin, de haut en bas. Dans la droite ligne du chapitre précédent, nous allons donc voir un mécanisme offert par C++ pour **répéter autant de fois que l'on souhaite une série d'instructions**.

Ce chapitre introduira **les boucles** et permettra d'améliorer encore notre gestion des erreurs, vue dans le chapitre précédent.

II.6.1. while— Tant que...

Commençons ce chapitre avec une boucle `while`. En toute originalité, ce mot-clef est aussi un mot anglais qui signifie «*tant que*». Elle exécute une série d'instructions **tant que la condition est vraie**. Par contre, dès que la condition est évaluée à `false`, la boucle s'arrête. On dit aussi qu'on **boucle** ou qu'on **itère**.

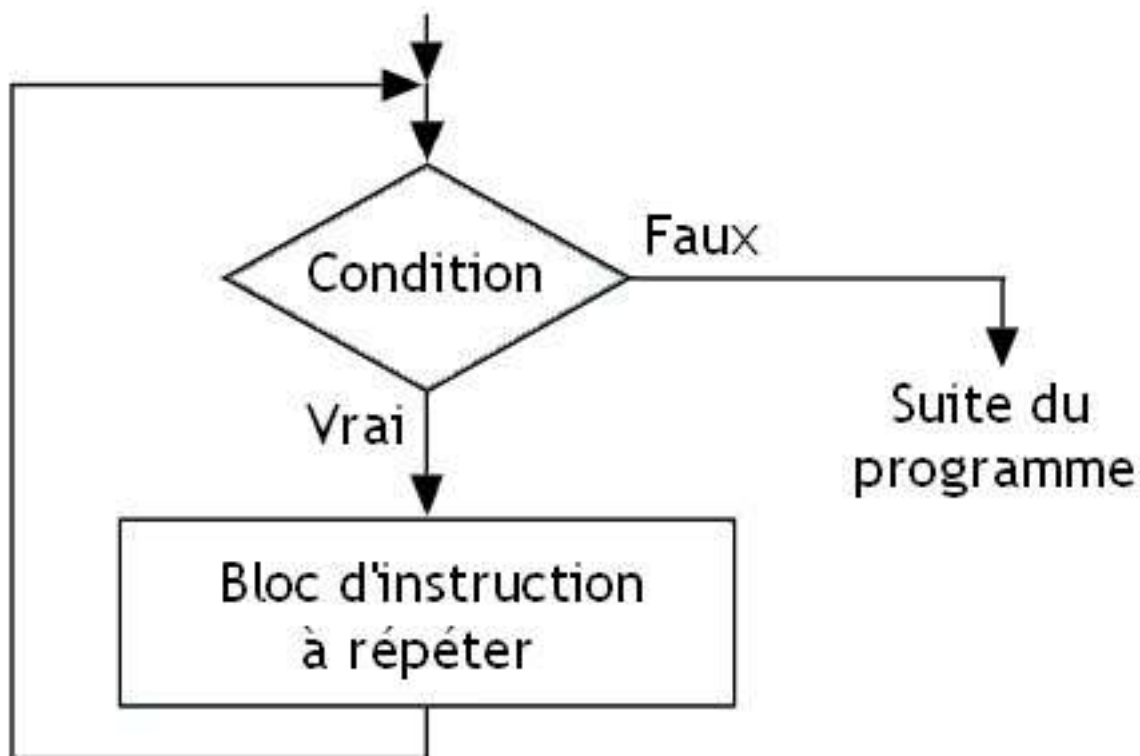


FIGURE II.6.1. – Image originale tirée du [tutoriel sur le langage C](#) .

Prenons un exemple tout bête: notre programme va compter de 0 à 9. Voici le code correspondant.

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Salut, je vais compter de 0 à 9." << std::endl;
6     int compteur { 0 };
7
8     while (compteur < 10)
9     {
10         std::cout << compteur << std::endl;
11         ++compteur;
12     }
13
14     std::cout << "Je m'arrête, après je sais pas faire." <<
15         std::endl;
16     return 0;
17 }
```

Le code est exécuté ainsi: on évalue la condition. Tant que `compteur` est strictement inférieur à 10, alors on affiche le nombre puis on l'incrémente. Quand finalement `compteur` vaut 10, la condition devient fausse, on passe à la suite du code.



Boucle infinie

Il est important de vérifier qu'à un moment ou un autre la condition devienne fausse et qu'on sort bien de la boucle. Si l'on n'y prend pas garde, on tombe dans le piège de la **boucle infinie**!



```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Salut, je vais compter de 0 à 9." <<
6         std::endl;
7     int compteur { 0 };
8     while (compteur < 10)
9     {
10         std::cout << compteur << std::endl;
11         // Oups, on a oublié d'incrémenter le compteur. Nous
12         // sommes bloqués dans la boucle.
13     }
14     std::cout << "Je ne serai jamais affiché." << std::endl;
15     return 0;
16 }
```

Dans ces cas, il ne reste plus qu'à terminer le programme sauvagement en le tuant (oui les informaticiens sont brutaux). Je répète donc, **faites attention aux boucles infinies**.

Comme les boucles sont basées sur les conditions, aucun problème pour en utiliser plusieurs en même temps.

```
1 #include <iostream>
2
3 int main()
4 {
5     char entree { '?' };
6     int compteur { 0 };
7
8     // On boucle tant qu'on a pas rentré la lettre 'e' ou que le
9     // compteur n'a pas atteint 5.
10    while (compteur < 5 && entree != 'e')
11    {
12        std::cout << "Le compteur vaut " << compteur << std::endl;
13        std::cout << "Rentre un caractère : ";
14        std::cin >> entree;
15
16        ++compteur;
17    }
18
19    std::cout << "Fin du programme." << std::endl;
20    return 0;
21 }
```


II.6.2. Exercices

II.6.2.1. Une laverie

Nous allons faire un **programme d'aide dans une laverie automatique**. Dans celle-ci, deux types de machines sont employées: des machines économes capables de laver 5kg de linge au maximum et d'autres, plus gourmandes, capables de laver 10kg.

- Si le linge fait moins de 5 kg, il faut le mettre dans une machine de 5kg.
- S'il fait entre 5kg et 10kg, il faut le mettre dans une machine de 10kg.
- Enfin, s'il fait plus, il faut répartir le linge entre machines de 10kg et, dès qu'il reste 5kg ou moins, mettre le linge restant dans une machine 5kg.

👁 Correction laverie

II.6.2.2. PGCD

Allez, on va faire un peu de mathématiques! Le **PGCD** de deux nombres est un entier qui est le plus grand diviseur que ces deux nombres ont en commun. Ainsi, le **PGCD** de 427 et 84 est 7, car $427 = 7 \times 61$ et $84 = 7 \times 12$. Pour le calculer, voici l'algorithme.

- On stocke le modulo du premier nombre, appelé **a**, par l'autre nombre, appelé **b**, dans une variable **r**.
- Tant que **r** est différent de zéro, on effectue les opérations suivantes.
 - On affecte la valeur de **b** à **a**.
 - On affecte la valeur de **r** à **b**.
 - On affecte à **r** le résultat du modulo de **a** par **b**.
- Quand **r** est nul, alors **b** représente le **PGCD** des deux nombres donnés en entrée.

👁 Correction **PGCD**

II.6.2.3. Somme de nombres de 1 à n

Disons que je vous donne un entier N au hasard. Pouvez-vous me créer un programme capable de me donner la somme de tous les entiers de 1 à n , c'est-à-dire la somme de $1 + 2 + 3 + \dots + (n - 2) + (n - 1) + n$?

👁 Correction somme de 1 à n

II.6.3. do while— Répéter ... tant que

La deuxième boucle offerte par C++ est très proche de la précédente. En fait, c'est la même, à deux différences près. Une boucle **do while** s'exécute **toujours une fois au minimum**, même si la condition est fausse. La condition est en effet vérifiée **après** l'exécution du code.

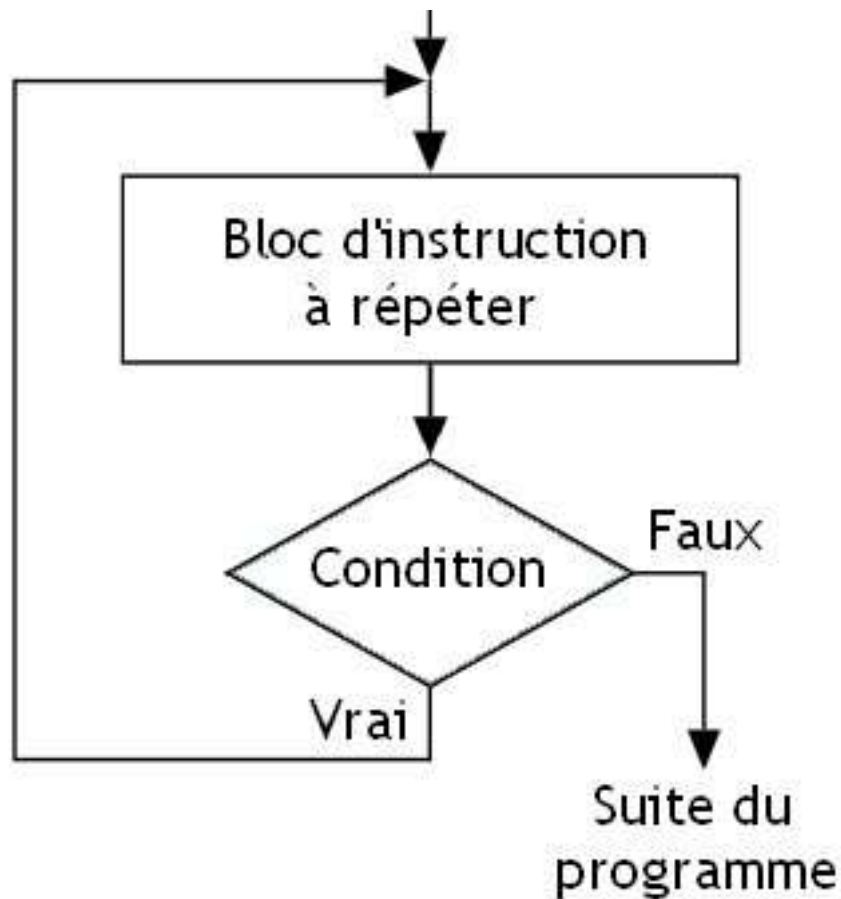


FIGURE II.6.2. – Do while

La deuxième, c'est qu'il faut un point-virgule après le **while**. Voyez par vous-mêmes le code suivant.

```
1 #include <iostream>
2
3 int main()
4 {
5     do
6     {
7         std::cout << "On passe quand même ici." << std::endl;
8     } while (false);
9
10    return 0;
11 }
```

II. Le début du voyage

Hormis ces deux subtiles différences, `do while` se comporte exactement comme `while` et permet elle aussi de boucler. Nous allons voir dans la section suivante comment les deux nous aiderons à résoudre le problème de sécurisation des entrées, soulevé au chapitre précédent.

II.6.4. [T.P] Gérer les erreurs d'entrée— Partie II

Dans la première partie, nous avions un mécanisme qui ne nous protégeait qu'une seule fois des erreurs. Maintenant, nous sommes en mesure de **nous protéger tant que l'entrée n'est pas correcte**. Améliorez donc le code que vous avez écrit pour qu'il soit capable de protéger le code suivant.

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Quel jour es-tu né ? ";
6      int jour { 0 };
7      std::cin >> jour;
8
9      std::cout << "Quel mois es-tu né ? ";
10     int mois { 0 };
11     std::cin >> mois;
12
13     std::cout << "Tu es né le " << jour << "/" << mois << "." <<
        std::endl;
14     return 0;
15 }
```

👁 Correction T.P partie II

Petit à petit, grâce à nos connaissances qui s'étoffent, nous améliorons la protection des entrées. Mais c'est dommage d'avoir le même code dupliqué à plusieurs endroits. C'est sale, ça nous fait perdre du temps si l'on doit modifier un morceau de code dans chaque endroit; et je ne parle pas des erreurs liés à des morceaux de codes différents des autres. Nous apprendrons dans le chapitre suivant comment s'en sortir.

II.6.5. `for`— Une boucle condensée

Les boucles `while` et `do while` sont très utiles et ajoutent une corde à notre arc. Mais ne trouvez-vous pas ça dommage d'avoir nos variables liées à l'itération (comme `compteur`) séparées de la boucle elle-même?

Actuellement, notre code est simple, mais imaginons un instant un code bien plus complexe. Il serait mieux de **séparer les opérations liées à la boucle de nos autres variables**

II. Le début du voyage

importantes. Cela rendra le code plus clair et plus compréhensible. C'est exactement ce que permet `for` («pour» en anglais).

```
1  for (/* initialisation */; /* condition */; /* itération */)
2  {
3      // Code de la boucle.
4  }
5
6  // Est équivalent à la boucle ci-dessous.
7
8  // Notez la création d'une portée artificielle.
9  {
10     // Initialisation.
11     while (/* condition */)
12     {
13         // Code de la boucle.
14         /* itération */
15     }
16 }
```

Hormis cette syntaxe différente, `for` s'utilise comme ses consœurs. Nous sommes libres de l'initialisation, de la condition et de l'itération.

```
1  #include <iostream>
2
3  int main()
4  {
5      // Compte de 10 à 1. On décrémente ici.
6      for (int compteur {10}; compteur > 0; --compteur)
7      {
8          std::cout << compteur << std::endl;
9      }
10
11     std::cout << "Fin du programme." << std::endl;
12     return 0;
13 }
```

Nous pouvons même effectuer plusieurs opérations en même temps, tant que les variables déclarées sont du même type. Cette possibilité est cependant à prendre avec des gants parce qu'elle peut rendre le code très peu lisible.

```
1  #include <iostream>
2
3  int main()
4  {
```

II. Le début du voyage

```
5 // Compte de 10 à 1 et de 1 à 10 en même temps.
6 for (int compteur {10}, compteur_inverse {1}; compteur > 0 &&
    compteur_inverse <= 10; --compteur, ++compteur_inverse)
7 {
8     std::cout << compteur << " et " << compteur_inverse <<
        std::endl;
9 }
10
11 std::cout << "Fin du programme." << std::endl;
12 return 0;
13 }
```



À quel moment dois-je utiliser `for` ou `while`?

Le choix est vôtre, C++ vous laisse entièrement libre. Après, on utilise plus souvent `for` dans le cas où l'on veut **itérer** (compter de 1 à 100 par exemple), alors que `while` **boucle pour réaliser un traitement autant de fois que nécessaire** (par exemple, tant que `std::cin.fail` renvoie `true`).



Variante

Dans le chapitre suivant, nous verrons qu'il existe une autre boucle `for`.

II.6.6. Boucles imbriquées

Comme pour les conditions, il est tout à fait possible d'**imbriquer plusieurs boucles** l'une dans l'autre. De même, on peut tout à fait mélanger les types de boucles, comme le prouve l'exemple suivant, qui affiche les tables de multiplications de 1 à 10 en utilisant `while` et `for`.

```
1 #include <iostream>
2
3 int main()
4 {
5     int facteur_gauche { 1 };
6     while (facteur_gauche <= 10)
7     {
8         for (int facteur_droite { 1 }; facteur_droite <= 10;
            ++facteur_droite)
9         {
10             std::cout << facteur_gauche << "x" << facteur_droite <<
                " = " << facteur_gauche * facteur_droite <<
                std::endl;
11         }
12     }
13 }
```

```
11     }
12     // On saute une ligne pour séparer chaque table.
13     std::cout << std::endl;
14     ++facteur_gauche;
15 }
16
17 return 0;
18 }
```

II.6.7. Convention de nommage

Dernier point avant de continuer: vous verrez très souvent, tant dans ce tutoriel que dans les divers codes C++ que vous aurez l'occasion d'examiner, que les programmeurs utilisent `i` comme identifiant de variable pour parcourir une boucle (ainsi que `j` et `k` dans le cas de boucles imbriquées). C'est une abréviation pour *iterator*, parce que cette variable itère, répète le corps de la boucle. **Ce nom est tellement court et explicite qu'il est pour ainsi dire devenu une convention de nommage en C++.**

II.6.8. Contrôler plus finement l'exécution de la boucle

Les boucles sont très utiles, mais le code à l'intérieur des accolades est soit entièrement sauté, soit entièrement exécuté. Sachez que C++ permet de contrôler un peu plus précisément l'exécution avec deux mots-clefs: `break` et `continue`.

i

Précision

Simple précision: `break` et `continue` s'utilisent avec les boucles uniquement. Ils ne s'utilisent pas avec les conditions: pas de `break` au sein d'un `if`, par exemple.

II.6.8.1. `break`— Je m'arrête là

Le premier, c'est `break`, qui en anglais signifie littéralement «casser», «interrompre». Celui-ci nous autorise à **mettre fin à l'exécution de la boucle**, peu importe où nous en étions. Dans le code ci-dessous, vous ne verrez jamais afficher les nombres au-dessus de 3 parce que la boucle est terminée par `break` dès que notre variable vaut 3.

```
1 #include <iostream>
2
3 int main()
4 {
```

II. Le début du voyage

```
5   for (int i { 0 }; i < 10; ++i)
6   {
7       // Si i vaut 3, j'arrête la boucle.
8       if (i == 3)
9       {
10          break;
11      }
12
13      std::cout << i << std::endl;
14  }
15
16  return 0;
17 }
```

Dans le cas de boucles imbriquées, `break` ne stoppe l'exécution que de la boucle dans laquelle il se trouve, mais pas les boucles englobantes.

```
1  for (int i { 0 }; i < 5; ++i)
2  {
3      for (int j { 0 }; j < 3; ++j)
4      {
5          if (j == 1)
6          {
7              // Quand j vaut 1, on retourne dans le for(i).
8              break;
9          }
10
11          std::cout << "Voici la valeur de j : " << j << std::endl;
12      }
13
14      std::cout << "Voici la valeur de i : " << i << std::endl;
15  }
```

Pour illustrer son fonctionnement avec un cas plus utile, prenons un exemple mathématique. Connaissez-vous le concept du [test de primalité](#) ? Il s'agit d'un test pour déterminer si un entier est premier, c'est-à-dire divisible uniquement par 1 et par lui-même; 13 est un nombre premier mais pas 10, car divisible par 1, 2, 5, et 10. Essayez donc de faire cet exercice.

👁 Indice

👁 Correction

II.6.8.2. `continue`— Saute ton tour !

L'autre mot-clef, `continue`, permet de **sauter l'itération courante**. Toutes les instructions qui se trouvent après ce mot-clef sont donc ignorées et **la boucle continue au tour suivant**. Imaginons que nous voulions un programme permettant d'afficher les nombres impairs compris entre 0 et un nombre choisi par l'utilisateur. Essayez donc de le faire, avec les connaissances que vous avez.

👁 Indice

👁 Correction



Potentielle boucle infinie

Attention lorsque vous utilisez `continue` au sein d'une boucle `while` ou `do while`, parce que **TOUT le code situé après est sauté**, même les instructions d'incrémentation. Le code suivant est donc un exemple de boucle infinie.

```
1 int i { 0 };
2 while (i < 10)
3 {
4     if (i == 5)
5     {
6         // Ohoh...
7         continue;
8     }
9
10    ++i;
11 }
```

II.6.8.3. En résumé

- C++ nous offre trois façons de boucler: `while`, `do while` et `for`.
- `while` s'utilise plutôt pour boucler tant qu'une condition n'est pas remplie alors que `for` est utilisé pour itérer sur un nombre connu d'éléments.
- Il faut faire attention à ne pas tomber dans une boucle infinie.
- Nous disposons de deux mots-clefs, `break` et `continue`, pour affiner notre contrôle sur l'exécution de la boucle.

Contenu masqué

Contenu masqué n°13 : Correction laverie

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout <<
6          "Bienvenue à la laverie de Clem. Combien de kilos de linge as-tu ? ";
7      int kilos { 0 };
8      std::cin >> kilos;
9
10     if (kilos <= 5)
11     {
12         std::cout <<
13             "Tu as peu de linge, mets-le dans une machine de 5kg."
14             << std::endl;
15     }
16     else if (kilos <= 10)
17     {
18         std::cout <<
19             "Tu as beaucoup de linge, mets-le dans une machine de 10kg."
20             << std::endl;
21     }
22     else
23     {
24         // On a plus que 10 kilos, il va falloir diviser le linge
25         // en plusieurs machines.
26         std::cout <<
27             "Dites donc, il faut laver son linge plus souvent !" <<
28             std::endl;
29
30         int nb_machines_10_kilos { 0 };
31
32         while (kilos > 5)
33         {
34             // Pour chaque machine de 10kg utilisée, on enlève 10kg
35             // au total de linge restant.
36             kilos -= 10;
37             ++nb_machines_10_kilos;
38         }
39
40         std::cout << "Tu as besoin de " << nb_machines_10_kilos <<
41             " machine(s) de 10kg." << std::endl;
42         if (kilos >= 0)
```

II. Le début du voyage

```
33     {
34         // S'il reste entre 0 et 5 kilos, il ne faut pas
           oublier la dernière machine de 5kg.
35         std::cout <<
           "Le reste rentrera dans une machine de 5kg." <<
           std::endl;
36     }
37 }
38
39 return 0;
40 }
```

[Retourner au texte.](#)

Contenu masqué n°14 : Correction PGCD

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Rentre un nombre a : ";
6      int a { 0 };
7      std::cin >> a;
8
9      std::cout << "Rentre un nombre b : ";
10     int b { 0 };
11     std::cin >> b;
12
13     int r { a % b };
14     while (r != 0)
15     {
16         a = b;
17         b = r;
18         // r vaut le reste de la division entière de a par b.
19         r = a % b;
20     }
21
22     std::cout << "PGCD vaut = " << b << std::endl;
23     return 0;
24 }
```

[Retourner au texte.](#)

Contenu masqué n°15 : Correction somme de 1 à n

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Donne-moi un entier : ";
6     int n { 0 };
7     std::cin >> n;
8
9     int total { 0 };
10    int compteur { 1 };
11
12    while (compteur <= n)
13    {
14        total += compteur;
15        ++compteur;
16    }
17
18    std::cout << "La somme totale vaut " << total << "." <<
19        std::endl;
20    return 0;
21 }
```

Ceux qui aiment les maths ont remarqué qu'il s'agit de la [somme des premiers entiers de 1 à n](#) et qu'il existe une formule pour calculer directement sans boucler: $\frac{N \times (N+1)}{2}$. Bravo à vous.



[Retourner au texte.](#)

Contenu masqué n°16 : Correction T.P partie II

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Quel jour es-tu né ? ";
6     int jour { 0 };
7
8     // Comme std::cin >> jour renvoie false s'il y a eu une erreur,
9     // afin de rentrer dans la boucle, on prend la négation de
10    // l'expression avec !
11    while (!(std::cin >> jour))
12    {
13    }
```

II. Le début du voyage

```
11     std::cout << "Entrée invalide. Recommence." << std::endl;
12     std::cin.clear();
13     std::cin.ignore(255, '\n');
14 }
15
16 std::cout << "Quel mois es-tu né ? ";
17 int mois { 0 };
18
19 // Même version avec un do while.
20 do
21 {
22     if (std::cin.fail())
23     {
24         std::cout << "Entrée invalide. Recommence." <<
                std::endl;
25         std::cin.clear();
26         std::cin.ignore(255, '\n');
27     }
28
29 } while (!(std::cin >> mois));
30
31 std::cout << "Tu es né le " << jour << "/" << mois << "." <<
    std::endl;
32 return 0;
33 }
```

Les deux versions sont légèrement différentes dans ce qu'elles font. La première demande l'entrée, puis tant qu'elle est invalide, elle nettoie puis demande à nouveau. La deuxième vérifie s'il y a eu une quelconque erreur, la nettoie puis demande l'entrée et recommence si l'entrée est invalide.

Peu importe celle que vous avez choisi d'utiliser, l'essentiel est de nettoyer nos entrées et d'avoir des programmes robustes. Et c'est justement ce que nous faisons. 🍊 [Retourner au texte.](#)

Contenu masqué n°17 :

Indice

Nous allons devoir tester tous les nombres entre 2 et x pour savoir s'ils sont diviseurs de x . Mais si nous trouvons un nombre y différent de x et qui divise x , alors nous savons que le nombre n'est pas premier, inutile de continuer. 🍊 [Retourner au texte.](#)

Contenu masqué n°18 :

Correction

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout <<
6         "Donne-moi un nombre, je te dirai s'il est premier : ";
7     int nombre { 0 };
8     std::cin >> nombre;
9
10    bool est_premier { true };
11    for (int i { 2 }; i < nombre; ++i)
12    {
13        if (nombre % i == 0)
14        {
15            // Si i divise nombre, alors nous savons que nombre
16            // n'est pas premier.
17            est_premier = false;
18            break;
19        }
20    }
21
22    if (est_premier)
23    {
24        std::cout << nombre << " est un nombre premier !" <<
25            std::endl;
26    }
27    else
28    {
29        std::cout << nombre << " n'est pas premier." << std::endl;
30    }
31
32    return 0;
33 }
```

[Retourner au texte.](#)

Contenu masqué n°19 :

Indice

Pour rappel, un nombre est pair s'il est divisible par 2 et impair autrement.

[Retourner au](#)

[texte.](#)

Contenu masqué n°20 :

Correction

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout <<
6         "Donne-moi un nombre, je t'afficherai tous les nombres impairs jusqu'à
7     int maximum { 0 };
8     std::cin >> maximum;
9
10    for (int i { 0 }; i <= maximum; ++i)
11    {
12        if (i % 2 == 0)
13        {
14            // On saute ce nombre puisqu'il est pair.
15            continue;
16        }
17        std::cout << i << std::endl;
18    }
19
20    return 0;
21 }
```

[Retourner au texte.](#)

II.7. Au tableau !

Si je vous demande de me faire un programme calculant la moyenne d'une série de nombres, comment feriez-vous? Avec une boucle, comme nous avons appris dans le chapitre précédent. Mais maintenant, j'ai besoin de les stocker. Comment faire, puisque vous ne savez pas à l'avance combien de notes je veux rentrer?

Ce chapitre va résoudre cette problématique en vous présentant **les tableaux**.

II.7.1. Un tableau c'est quoi ?

Les tableaux, en informatique, font partie de la grande famille des **structures de données** [↗](#). Ces structures de données sont des façons particulières **d'organiser et de stocker des données**. Il y a ainsi des structures pour stocker un nombre fixe de données du même type, d'autres pour un nombre variable, d'autre encore pour stocker des données de types différents. Ces objets, conçus spécialement pour stocker des données, sont appelés **conteneurs**.

Celles qui nous intéressent maintenant sont les tableaux. Ils sont conçus pour stocker des **données de même type** et ayant **des points communs**: notes d'un élève à l'école, titres de livres, âge d'utilisateurs. Dès que vous pensez «liste de...», dès que des éléments ont des points communs **et qu'il fait sens de les regrouper**, alors l'usage d'un tableau peut s'envisager.

Un tableau n'est cependant pas la solution universelle, mais ça, nous le verrons au fur et à mesure, pour ne pas vous encombrer inutilement de détails. 🍊

II.7.2. `std::vector`— Mais quel dynamisme !

Commençons par les **tableaux dynamiques**, c'est-à-dire dont la taille ne sera connue qu'au moment de l'exécution du programme. C++ nous fournit justement un objet déjà codé pour ça et qui peut facilement s'agrandir ou se rétrécir: `std::vector`. Et le fichier d'en-tête correspondant se doit bien évidemment d'être inclus en faisant `#include <vector>`.

II.7.2.1. Déclarer un `std::vector`

Les concepteurs de `std::vector` l'ont créé de telle sorte qu'il soit utilisable avec n'importe quel type. On peut donc utiliser un `std::vector` de `int` pour stocker des âges et un `std::vector` de `double` pour stocker des moyennes. Voici la façon de faire.

II. Le début du voyage

```
1 std::vector< /* type des éléments du tableau */> identifiant {};
```

Avec maintenant trois exemples concrets.

```
1 #include <string>
2 // N'oubliez pas cette ligne.
3 #include <vector>
4
5 int main()
6 {
7     std::vector<int> const tableau_de_int {};
8     std::vector<double> const tableau_de_double {};
9     // Même avec des chaînes de caractères c'est possible.
10    std::vector<std::string> const tableau_de_string {};
11
12    return 0;
13 }
```

Tous nos `std::vector`, ou tableaux, sont vides avec ces déclarations. Mais on peut très bien leur affecter des valeurs, comme ci-dessous. Chacun de nos tableaux a un nombre d'éléments différents dès sa création.

```
1 #include <string>
2 #include <vector>
3
4 int main()
5 {
6     // On stocke 5 entiers.
7     std::vector<int> const tableau_de_int { 1, 2, 3, 4, 5 };
8     // Et ici 2 flottants.
9     std::vector<double> const tableau_de_double { 2.71828, 3.14159
10     };
11     // Maintenant 3 chaînes de caractères.
12     std::vector<std::string> const tableau_de_string {
13         "Une phrase.", "Et une autre !", "Allez, une pour la fin."
14     };
15
16    return 0;
17 }
```

Rien ne nous empêche non plus de copier un tableau dans un autre, tout comme on le fait pour des `int`.

II. Le début du voyage

```
1 #include <vector>
2
3 int main()
4 {
5     std::vector<int> const tableau_de_int { 1, 2, 3 };
6     // Maintenant, copie possède les mêmes éléments que
7     // tableau_de_int.
8     std::vector<int> copie { tableau_de_int };
9
10    return 0;
11 }
```

II.7.2.2. Manipuler les éléments d'un `std::vector`

II.8. Accès aux éléments

Pour l'instant, nos tableaux sont inutiles, nous ne faisons rien avec. Commençons donc à nous amuser un peu en récupérant un élément du tableau pour l'afficher. On utilise pour cela les crochets devant l'identifiant de notre tableau, avec entre crochets la position de l'élément à récupérer.

```
1 identificateur[/ * un entier */];
```

```
1 #include <vector>
2 #include <iostream>
3
4 int main()
5 {
6     std::vector<int> const nombres { 2, 4, 6, 8 };
7     std::cout << "Deuxième élément: " << nombres[1] << '\n';
8
9     return 0;
10 }
```

?

Pourquoi récupère-t-on le deuxième élément en écrivant `1`? Le deuxième élément, ça devrait être `2` non?

Justement non. La subtilité des tableaux en C++, c'est qu'ils **commencent à l'indice 0 et non l'indice 1**. Pourquoi ça? C'est un héritage du C que je ne vais pas détailler car il demande des connaissances que vous n'avez pas encore. Souvenez-vous simplement que **le premier élément d'un tableau est à l'indice 0** et que donc le x ème élément d'un tableau est à l'indice $x - 1$.

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> const tableau_de_int { 1, 2, 3 };
7 }
```

II. Le début du voyage

```
8      std::cout << "Le premier élément est " << tableau_de_int[0] <<
      "." << std::endl;
9      std::cout << "Le deuxième élément est " << tableau_de_int[1] <<
      "." << std::endl;
10     std::cout << "Le troisième élément est " << tableau_de_int[2]
      << "." << std::endl;
11
12     return 0;
13 }
```

?

Et si j'accède à l'élément 4, que se passe-t-il?

Potentiellement, tout et n'importe quoi. Parfois, le programme va continuer sans rien dire, d'autre fois vous allez avoir des erreurs et des plantages, comme je l'illustre ci-dessous. Parfois, le programme peut aussi péter complètement un câble et lancer un missile nucléaire sur un pays pris au hasard. C'est ce qu'on appelle un **comportement indéterminé**, en anglais «*undefined behavior*». Impossible donc de faire du code de qualité dans ces conditions. Voilà pourquoi il ne faut pas utiliser d'indice inférieur à 0 ou plus grand que la taille de notre tableau moins un.

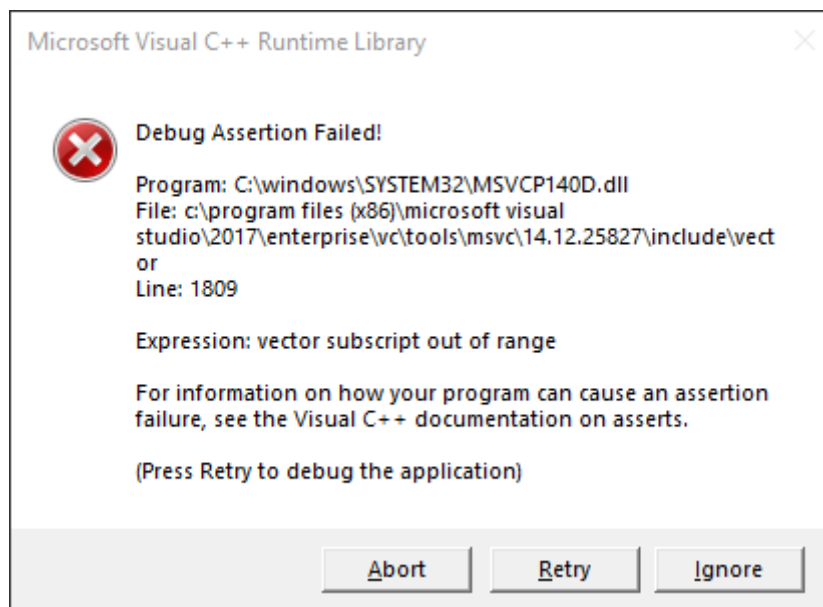


FIGURE II.8.1. – Un exemple avec Visual Studio 2017 de ce qui arrive.

×

Comportement indéterminé

Un « UB » est le **signe d'une erreur de programmation**. Vous avez mal fait quelque chose, quelque part dans le code et le programme ne peut plus être considéré comme fiable désormais. Il faut donc à tout prix les éviter et faire attention en codant!

II. Le début du voyage

II.8.0.0.1. Premier et dernier élément

Il existe d'ailleurs deux fonctions dont le but est d'accéder spécifiquement au premier ou au dernier élément d'un tableau. Elles se nomment, respectivement, `front` et `back`. Leur utilisation est très simple.

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> const tableau_de_int { 1, 2, 3, 4, 5, 6, 7, 8,
7                                             9 };
8
9     std::cout << "Le premier élément est " <<
10         tableau_de_int.front() << "." << std::endl;
11     std::cout << "Le dernier élément est " << tableau_de_int.back()
12         << "." << std::endl;
13
14     return 0;
15 }
```

II.8.0.0.2. Obtenir la taille d'un tableau

Justement, comment récupère-t-on la taille de notre tableau? En utilisant la fonction `std::size`, qui renvoie le nombre d'éléments de n'importe quel conteneur.

Le type de retour de cette fonction est un type nouveau, que nous n'avons pas rencontré jusque-là et qui s'appelle `std::size_t`. C'est un type entier conçu, entre autres, pour être capable de stocker la taille de n'importe quel tableau, aussi grand soit-il.

En vrai, un `int` est suffisant et `std::size_t` est présent car historiquement déjà là en C. De toute façon, avec `auto`, nous n'avons aucune raison de nous casser la tête là-dessus. 🍊

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> const tableau_de_int { 1, 2, 3 };
7
8     auto const taille { std::size(tableau_de_int) };
9     std::cout << "Mon tableau contient " << taille << " éléments."
10         << std::endl;
11
12     return 0;
13 }
```

```
12 }
```

II.8.0.0.3. Afficher les éléments d'un tableau

Comme nous l'avions dit dans le chapitre sur les boucles, depuis l'arrivée de C++11, il existe une variante de la boucle `for`, conçue exclusivement pour parcourir les éléments d'un conteneur. Ci-dessous, voici un pseudo-code illustrant cette possibilité.

```
1  for (/* type d'un élément du tableau ou auto */ identifiant : /*  
   tableau à parcourir */)
2  {
3      // Manipuler identifiant, en l'affichant par exemple.
4  }
```

Voici ce que cela donne avec un exemple concret. Pas besoin de récupérer la taille, la boucle est intelligente. Cela nous facilite la vie et rend notre code plus concis et plus facile à lire.

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector<int> const tableau_entiers { 1, 2, 3 };
7
8      std::cout << "Je vais afficher mon tableau en entier.\n";
9      for (auto const element : tableau_entiers)
10     {
11         std::cout << element << std::endl;
12     }
13
14     return 0;
15 }
```



`auto` ou pas?

Quant à utiliser `auto` ou préciser explicitement le type d'un élément (`int` par exemple), c'est une question personnelle. Certains ne vont jurer que par `auto`, d'autres ne le mettront que si le type est long à écrire. Vous êtes libres de choisir ce qu'il vous plaît.

II.8.0.0.4. Vérifier si un tableau est vide

Fonction bien utile, `std::empty` permet de vérifier si un tableau est vide, c'est-à-dire qu'il ne contient aucun élément et donc qu'il a une taille de zéro. Elle renvoie un booléen, donc cela signifie que cette fonction appliquée à un tableau vide renverra `true` et `false` pour un tableau avec trois éléments, par exemple.

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> const tableau_vide {};
7     std::vector<int> const tableau_rempli { 1, 2, 3 };
8
9     std::cout << std::boolalpha;
10    std::cout << "Est-ce que tableau_vide est vide ? Réponse : " <<
        std::empty(tableau_vide) << std::endl;
11    std::cout << "Est-ce que tableau_rempli est vide ? Réponse : "
        << std::empty(tableau_rempli) << std::endl;
12
13    return 0;
14 }
```

II.8.0.0.5. Ajouter des éléments

Pour l'instant, nos `std::vector` ne sont pas trop dynamiques. Mais cela va changer avec la fonction `push_back`, qui va nous donner la possibilité d'ajouter un élément à la fin de notre tableau. Celui-ci doit cependant être du même type que les autres éléments du tableau. On ne pourra ainsi pas ajouter de `int` à un tableau de `std::string`.

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4
5 int main()
6 {
7     std::vector<int> tableau_de_int { 12, 24 };
8     // On ajoute un élément...
9     tableau_de_int.push_back(36);
10    // ...mais on peut en ajouter encore d'autres.
11    tableau_de_int.push_back(48);
12    tableau_de_int.push_back(100);
13
14    // On affiche pour vérifier.
```

II. Le début du voyage

```
15     for (auto const valeur : tableau_de_int)
16     {
17         std::cout << valeur << std::endl;
18     }
19
20     std::vector<std::string> tableau_de_string { "Salut !",
21         "Voici une phrase." };
22
23     tableau_de_string.push_back("Mais je vais en ajouter une autre.");
24     // Ceci ne compilera pas.
25     //tableau_de_string.push_back(5);
26
27     for (auto const chaine : tableau_de_string)
28     {
29         std::cout << chaine << std::endl;
30     }
31
32     return 0;
33 }
```

Et pour vous prouver que la taille change bien, il suffit d'utiliser `std::size` comme vu plus haut.

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector<int> tableau_de_int { 12, 24 };
7
8      std::cout << "Taille avant l'ajout : " << tableau_de_int.size()
9          << std::endl;
10
11      tableau_de_int.push_back(36);
12      tableau_de_int.push_back(48);
13      tableau_de_int.push_back(100);
14
15      std::cout << "Taille après l'ajout : " <<
16          std::size(tableau_de_int) << std::endl;
17
18      return 0;
19  }
```

II.8.0.0.6. Supprimer des éléments

On ajoute, on ajoute, mais on a aussi envie de pouvoir retirer un voire tous les éléments. Cette possibilité nous est également offerte avec les fonctions `pop_back` et `clear`. Ces fonctions sont

II. Le début du voyage

simples, comme l'illustre l'exemple ci-dessous.

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> tableau_de_int { 12, 24 };
7
8     std::cout << "Taille avant l'ajout : " <<
9         std::size(tableau_de_int) << std::endl;
10
11     tableau_de_int.push_back(36);
12     tableau_de_int.push_back(48);
13     tableau_de_int.push_back(100);
14
15     std::cout << "Taille après l'ajout : " <<
16         std::size(tableau_de_int) << std::endl;
17
18     // Je retire le dernier élément.
19     tableau_de_int.pop_back();
20     std::cout << "Taille après avoir retiré le dernier élément : "
21         << std::size(tableau_de_int) << std::endl;
22
23     // Finalement j'en ai marre, je vide le tableau.
24     tableau_de_int.clear();
25     std::cout << "Taille après avoir vidé le tableau : " <<
26         std::size(tableau_de_int) << std::endl;
27
28     return 0;
29 }
```



Comportement indéterminé

Appeler `pop_back` sur un tableau vide est aussi un comportement indéterminé.

II.8.0.0.7. Modifier des éléments

Pour l'instant, nous ne faisons que lire les valeurs de notre tableau, mais il n'y a aucun problème pour les modifier, tant que le tableau n'est pas déclaré comme `const` bien évidemment. Et la bonne nouvelle, c'est que vous savez déjà comment faire: les crochets `[]`. En effet, s'ils sont suivis d'un `=` puis d'une valeur, alors l'élément du tableau auquel on accède se voit modifié.

II. Le début du voyage

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> tableau { 1, 2, 3 };
7
8     // Finalement, je préfère que la première valeur soit un 15.
9     tableau[0] = 15;
10
11     for (auto const valeur : tableau)
12     {
13         std::cout << valeur << std::endl;
14     }
15
16     return 0;
17 }
```

De même que pour la lecture, n'essayez pas de modifier la valeur d'un élément en dehors du tableau. Je vous le rappelle, c'est un comportement indéterminé et qui est potentiellement dangereux!

II.8.0.0.8. Remplir le tableau avec une valeur unique

Dernière fonction que je vais vous montrer, celle-ci est bien utile pour remplir d'un coup un tableau d'une même valeur. La fonction `assign` a besoin de savoir combien de valeurs vous comptez insérer et quelle est-elle.

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> valeurs {};
7     std::cout << "Taille actuelle de mon tableau : " <<
8         std::size(valeurs) << " éléments.\n";
9
10    // On ajoute dix fois la valeur 42.
11    valeurs.assign(10, 42);
12    std::cout << "Taille actuelle de mon tableau : " <<
13        std::size(valeurs) << " éléments.\n";
14
15    for (auto const valeur : valeurs)
16    {
17        std::cout << valeur << std::endl;
18    }
19 }
```

```
17
18     std::cout << '\n';
19
20     // Finalement on réduit en assignant cinq fois la valeur 3.
21     valeurs.assign(5, 3);
22     std::cout << "Taille actuelle de mon tableau : " <<
        std::size(valeurs) << " éléments.\n";
23
24     for (auto const valeur : valeurs)
25     {
26         std::cout << valeur << std::endl;
27     }
28
29     std::cout << '\n';
30
31     // Mais on peut très bien augmenter.
32     valeurs.assign(12, 78);
33     std::cout << "Taille actuelle de mon tableau : " <<
        std::size(valeurs) << " éléments.\n";
34
35     for (auto const valeur : valeurs)
36     {
37         std::cout << valeur << std::endl;
38     }
39
40     return 0;
41 }
```

II.8.1. Exercices

II.8.1.1. Calcul de moyenne

Dans cet exercice, vous allez laisser l'utilisateur entrer autant de valeurs qu'il le souhaite puis en faire la moyenne, c'est-à-dire diviser la somme des notes par le nombre de notes. L'utilisateur pourra taper autant de valeurs positives qu'il le souhaite. Cependant, un nombre négatif signifiera qu'il ne souhaite plus entrer de note.

👁 Correction calcul de moyenne

II.8.1.2. Minimum et maximum

Le but de ce programme est tout simple: il faut trouver le plus petit élément du tableau, ainsi que le plus grand. Pour se simplifier la vie, vous pouvez utiliser un tableau déjà rempli.

👁 Correction maximum et minimum

II.8.1.3. Séparer les pairs des impairs

Cette fois, on veut créer deux nouveaux tableaux qui contiendront, pour le premier les nombres pairs uniquement, pour le second les nombres impairs.

👁 Correction pairs et impairs

II.8.1.4. Compter les occurrences d'une valeur

Une opération très courante avec un tableau est de compter le nombre de fois que revient une certaine valeur. Faites donc un programme qui permet de compter le nombre de fois que revient une lettre dans un tableau de `char`.

👁 Correction occurrence d'une valeur

II.8.2. `std::array`— Le tableau statique

Abordons maintenant l'autre catégorie de tableaux qui existe en C++: les **tableaux statiques**, c'est-à-dire dont la taille est **connue à la compilation** et **qui ne peut varier**. En contrepartie, ce tableau est plus performant et plus rapide qu'un `std::vector` puisqu'il n'y a pas d'opération d'ajout ou de retrait d'éléments.

Pour l'utiliser, il faut inclure le fichier d'en-tête `<array>`.

II.8.2.1. Déclarer un `std::array`

Comme pour `std::vector`, notre nouveau conteneur attend qu'on lui indique non seulement quel type de données il doit stocker, mais également **combien**. Je rappelle que la taille doit être **définie à la compilation**. Vous ne pouvez pas déclarer un `std::array` et lui passer une taille ultérieurement. Voyez vous-mêmes l'exemple ci-dessous.

```
1 #include <array>
2 #include <iostream>
3
4 int main()
5 {
```

II. Le début du voyage

```
6      // Tableau contenant cinq entiers explicitement précisés.
7      std::array<int, 5> const tableau_de_5_entiers { 1, 2, 3, 4,
8              5 };
9      for (auto const element : tableau_de_5_entiers)
10     {
11         std::cout << element << std::endl;
12     }
13
14     // Tableau contenant deux décimaux définis puis cinq zéros.
15     std::array<double, 7> const tableau_de_7_decimaux {
16         3.14159, 2.1878 };
17     for (auto const element : tableau_de_7_decimaux)
18     {
19         std::cout << element << std::endl;
20     }
21
22     // Tableau qui a trop d'éléments et ne compilera pas.
23     //std::array<int, 2> const tableau_qui_ne_compilera_pas {
24         1, 2, 3, 4 };
25
26     // Il est tout à fait possible de copier un tableau.
27     auto copie { tableau_de_5_entiers };
28     for (auto const element : copie)
29     {
30         std::cout << element << std::endl;
31     }
32
33     return 0;
34 }
```

Avez-vous remarqué que, contrairement à `std::vector`, un tableau statique défini avec `std::array` contient des éléments par défaut? Ainsi, même si l'on ne précise pas tous les éléments, comme pour `tableau_de_7_decimaux`, nous avons la garantie que les autres seront initialisés à zéro.

II.8.2.2. Remplir un tableau

Il existe une fonction permettant de remplir le tableau, mais contrairement à `std::vector` qui peut être redimensionné et donc attend le nouveau nombre d'éléments, la fonction `fill` pour `std::array` n'attend que la valeur utilisée pour remplir le tableau.

```
1  #include <array>
2  #include <iostream>
3
4  int main()
5  {
```

II. Le début du voyage

```
6     std::array<int, 5> valeurs { 1, 2, 3, 4, 5 };
7
8     std::cout << "Avant" << std::endl;
9     for (auto const valeur : valeurs)
10    {
11        std::cout << valeur << std::endl;
12    }
13
14    // On remplit de 42.
15    valeurs.fill(42);
16
17    std::cout << std::endl << "Après" << std::endl;
18    for (auto const valeur : valeurs)
19    {
20        std::cout << valeur << std::endl;
21    }
22
23    return 0;
24 }
```

II.8.2.3. Accéder aux éléments

Comme précédemment, pour accéder à un élément du tableau, nous allons utiliser les crochets []. Et comme pour `std::vector`, faites attention à **ne pas accéder à un élément en dehors des limites du tableau**.

```
1  #include <array>
2  #include <iostream>
3
4  int main()
5  {
6      std::array<double, 7> tableau_de_7_decimaux { 3.14159, 2.1878
7          };
8      // C'est le nombre d'or.
9      tableau_de_7_decimaux[2] = 1.61803;
10
11     // Aïe aïe aïe, catastrophe !
12     //tableau_de_7_decimaux[7] = 1.0;
13
14     for (auto const element : tableau_de_7_decimaux)
15     {
16         std::cout << element << std::endl;
17     }
18
19     return 0;
20 }
```

II. Le début du voyage

Les fonctions `front` et `back` sont toujours de la partie.

```
1 #include <array>
2 #include <iostream>
3
4 int main()
5 {
6     std::array<int, 9> const tableau_de_int { 1, 2, 3, 4, 5, 6, 7,
7         8, 9 };
8
9     std::cout << "Le premier élément est " <<
10         tableau_de_int.front() << "." << std::endl;
11     std::cout << "Le dernier élément est " << tableau_de_int.back()
12         << "." << std::endl;
13
14     return 0;
15 }
```

II.8.2.4. Connaître la taille

Sans surprise, c'est la même fonction que pour `std::vector`, je parle de `std::size`. Le code qui suit ne devrait donc pas vous surprendre.

```
1 #include <array>
2 #include <iostream>
3
4 int main()
5 {
6     std::array<int, 5> const valeurs { 1, 2, 3, 4, 5 };
7     std::cout << "Et la taille de mon tableau vaut bien " <<
8         std::size(valeurs) << "." << std::endl;
9     return 0;
10 }
```

II.8.2.5. Vérifier si le tableau est vide

De même que vu plus haut, rien de dépayçant, la fonction `std::empty` renvoie `false` si le tableau contient des éléments et `true` s'il est vide. En pratique, cette fonction ne vous renverra jamais `true` car avoir un tableau vide signifie ici que la taille donnée est `0`. Autant vous dire que c'est inutile d'avoir un tableau vide.

```
1 #include <array>
2 #include <iostream>
3
4 int main()
5 {
6     std::array<int, 5> const valeurs { 1, 2, 3, 4, 5 };
7     std::cout << std::boolalpha;
8     std::cout << "Est-ce que mon tableau est vide ? " <<
9         std::empty(valeurs) << " Ah, je le savais !" << std::endl;
10    return 0;
11 }
```

II.8.3. `std::string`— Un type qui cache bien son jeu

Vous vous demandez peut-être toujours à quoi servent **vraiment** les tableaux. Vous voudriez un exemple **concret** d'un tableau qui rend de grands services. Ça tombe très bien puisque vous en utilisez un depuis le début de ce cours: `std::string`.

En effet, vous vous souvenez que C++ utilise le type `char` pour stocker un caractère. De là, stocker du texte se fait en stockant un ensemble de `char`. Le type `std::string` n'est donc rien d'autre qu'un **tableau de `char`**, conçu et optimisé spécialement pour le stockage de texte.

II.8.3.1. Connaître la taille d'une chaîne

En toute originalité, on peut récupérer la taille d'une `std::string` grâce à la fonction `std::size`.

```
1 #include <cassert>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::string const phrase { "Voici une phrase normale." };
8     std::cout << "Voici la longueur de cette phrase : " <<
9         std::size(phrase) << " caractères." << std::endl;
10    return 0;
11 }
```

II.8.3.2. Accéder à un caractère

Comme pour les deux conteneurs que nous avons découverts plus tôt dans ce chapitre, nous pouvons utiliser les crochets `[]` pour **accéder à un caractère** en particulier. Et, exactement comme pour les deux autres, il faut faire attention à bien utiliser un indice qui est inférieur à la taille de la chaîne.

```
1 #include <cassert>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::string const phrase { "Ceci me servira d'exemple." };
8     auto const indice { 5 };
9
10    std::cout << "Voici le caractère à la position " << indice <<
11        " : " << phrase[indice] << std::endl;
12    return 0;
13 }
```

On peut bien entendu modifier un caractère sans problème.

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string verbe { "Estamper" };
7     verbe[3] = 'o';
8     std::cout << "Verbe choisi : " << verbe << std::endl;
9
10    return 0;
11 }
```

II.8.3.3. Premier et dernier caractère

Les deux fonctions `front` et `back` nous rendent le même service que précédemment, en nous permettant de récupérer le premier et le dernier caractère de la chaîne.

```
1 #include <iostream>
2 #include <string>
3
```


II. Le début du voyage

```
4 int main()
5 {
6     std::string const phrase { "Bonjour tout le monde !" };
7     std::cout << "Première lettre : " << phrase.front() <<
        std::endl;
8     std::cout << "Dernière lettre : " << phrase.back() <<
        std::endl;
9
10    return 0;
11 }
```

II.8.3.4. Vérifier qu'une chaîne est vide

Toujours sans surprise pour vous, nous allons utiliser la même fonction `std::empty` que pour `std::vector` et `std::array`.

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string const texte { "Du texte." };
7
8     std::cout << std::boolalpha;
9     std::cout << "Est ce que 'texte' est vide ? " <<
        std::empty(texte) << std::endl;
10
11    std::string const rien { "" };
12    std::cout << "Est ce que 'rien' est vide ? " <<
        std::empty(rien) << std::endl;
13
14    return 0;
15 }
```

II.8.3.5. Ajouter ou supprimer un caractère à la fin

Nous bénéficions également des fonctions `push_back` et `pop_back` pour, respectivement, ajouter et supprimer un caractère à la fin de la chaîne. Faites toujours attention à ne pas retirer le dernier caractère d'une chaîne vide, c'est un comportement indéfini.

```
1 #include <iostream>
2 #include <string>
```

```
3
4 int main()
5 {
6     std::string phrase { "Une phrase !" };
7
8     phrase.pop_back();
9     phrase.pop_back();
10
11     phrase.push_back('.');
12
13     std::cout << phrase << std::endl;
14
15     return 0;
16 }
```

II.8.3.6. Supprimer tous les caractères

Nous avons une méthode portant le doux nom de `clear`, de l'anglais «débarasser, nettoyer, vider», qui va nous laisser une chaîne vide.

```
1 #include <cassert>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::string phrase { "Blablabla du texte." };
8     phrase.clear();
9     std::cout << "Rien ne va s'afficher : " << phrase << std::endl;
10
11     return 0;
12 }
```

II.8.3.7. Boucler sur une chaîne

On peut bien évidemment boucler sur tous les caractères qui composent une chaîne à l'aide de `for`.

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
```

II. Le début du voyage

```
6     std::string const phrase { "Voici un ensemble de lettres." };
7     for (char lettre : phrase)
8     {
9         std::cout << lettre << std::endl;
10    }
11
12    return 0;
13 }
```

Bien entendu, `std::string` est encore plus fourni que ça et permet de faire de nombreuses autres choses. Nous aurons l'occasion de les examiner plus tard dans ce tutoriel. Retenez simplement que **`std::string` peut être manipulé comme un tableau.**

II.8.3.8. En résumé

- Les tableaux nous permettent de stocker un ensemble de valeurs liées entre elles (notes, âges, moyennes, caractères, etc).
- Il existe plusieurs types de tableaux en C++: dynamique quand la taille est connue à l'exécution et statique quand elle est connue à la compilation.
- `std::vector` est un tableau dynamique, dont le nombre d'éléments peut augmenter ou diminuer à volonté.
- `std::array` est un tableau statique, dont la taille est fixée définitivement à la compilation, et dont on ne peut pas ajouter ni supprimer d'éléments.
- `std::string` est un tableau dynamique spécialement conçu pour la gestion du texte.

Contenu masqué

Contenu masqué n°21 : Correction calcul de moyenne

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector<double> notes {};
7
8      while (true)
9      {
10         std::cout <<
11             "Entre une note (inférieur à 0 pour arrêter) : ";
12         double note { 0.0 };
13     }
```

II. Le début du voyage

```
12
13         // N'oublions pas de protéger notre entrée.
14     while (!(std::cin >> note))
15     {
16         std::cout << "Entrée invalide. Recommence."
17             << std::endl;
18         std::cin.clear();
19         std::cin.ignore(255, '\n');
20     }
21     if (note < 0)
22     {
23         // On s'arrête si la note est inférieur à 0.
24         break;
25     }
26
27     // On ajoute la note à la liste des notes.
28     notes.push_back(note);
29 }
30
31 if (!std::empty(notes))
32 {
33     double total{};
34     for (auto const note : notes)
35     {
36         total += note;
37     }
38
39     std::cout << "Ta moyenne est de " << total / notes.size()
40         << "." << std::endl;
41 }
42 else
43 {
44     std::cout <<
45         "Tu n'as pas rentré de notes, je ne peux pas calculer ta moyenne."
46         << std::endl;
47 }
48
49 return 0;
50 }
```

Aviez-vous pensé à prendre en compte si l'utilisateur ne rentre aucune note car sa seule entrée est négative? 🍊

[Retourner au texte.](#)

Contenu masqué n°22 :

Correction maximum et minimum

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> const entiers { 5, -2, 74, 455, -12 };
7     int minimum { entiers[0] };
8     int maximum { entiers[0] };
9
10    for (auto const entier : entiers)
11    {
12        if (entier < minimum)
13        {
14            minimum = entier;
15        }
16
17        if (entier > maximum)
18        {
19            maximum = entier;
20        }
21    }
22
23    std::cout << "Le minimum est " << minimum << "." << std::endl;
24    std::cout << "Le maximum est " << maximum << "." << std::endl;
25
26    return 0;
27 }
```

[Retourner au texte.](#)

Contenu masqué n°23 : Correction pairs et impairs

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> const entiers{ 5, -2, 74, 455, -12, 10, 11 };
7     std::vector<int> pairs {};
8     std::vector<int> impairs {};
9
10    for (auto const entier : entiers)
```

II. Le début du voyage

```
11     {
12         if (entier % 2 == 0)
13         {
14             pairs.push_back(entier);
15         }
16         else
17         {
18             impairs.push_back(entier);
19         }
20     }
21
22     std::cout << "Les pairs sont : " << std::endl;
23     for (int const entier_pair : pairs)
24     {
25         std::cout << entier_pair << std::endl;
26     }
27
28     std::cout << std::endl << "Les impairs sont : " << std::endl;
29     for (int const entier_impair : impairs)
30     {
31         std::cout << entier_impair << std::endl;
32     }
33
34     return 0;
35 }
```

[Retourner au texte.](#)

Contenu masqué n°24 : Correction occurrence d'une valeur

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector<char> const lettres { 'a', 'b', 'b', 'c', 'b', 'd',
7                                       'd', 'a' };
9      char const lettre_a_compter { 'b' };
10     int total { 0 };
11
12     for (auto const lettre : lettres)
13     {
14         if (lettre == lettre_a_compter)
15         {
16             ++total;
17         }
18     }
19 }
```

II. Le début du voyage

```
15     }
16 }
17
18 std::cout << "La lettre '" << lettre_a_compter <<
    "' est présente " << total << " fois." << std::endl;
19
20 return 0;
21 }
```

[Retourner au texte.](#)

II.9. Déployons la toute puissance des conteneurs

Depuis que nous avons commencé ce cours, nous avons étudié deux conteneurs très utiles, `std::vector`, qui est un tableau dynamique, et `std::array`, un tableau statique. Nous avons appris qu'on peut accéder à un élément grâce aux crochets `[]`. Cette solution n'est pourtant **pas très générique**, puisque avec d'autres conteneurs C++, comme `std::list`, les crochets ne sont pas disponibles.

Ce chapitre va donc introduire une solution générique à ce problème qui porte le nom **d'itérateurs**, largement employés dans la bibliothèque standard, notamment avec les **algorithmes**.

II.9.1. Le socle de base : les itérateurs

Qu'est-ce qu'un itérateur? Littéralement, il s'agit de quelque chose dont on se sert **pour itérer**. Dans notre cas, c'est sur des conteneurs que nous voulons itérer. Et qui dit concept de boucle, dit accès aux éléments un par un. Nous avons vu plus tôt les crochets `[]` avec `std::array` et `std::vector`. Il s'agit ici du même concept, mais encore plus puissant, car totalement **indépendant du conteneur utilisé**!

Un itérateur peut accomplir ceci puisque il sait à quel endroit de la séquence regarder. On dit que l'itérateur est **un curseur sur l'élément**. Et en se déplaçant d'emplacement en emplacement, il peut ainsi parcourir l'ensemble d'une collection de données.

II.9.1.1. Déclaration d'un itérateur

Pour déclarer un itérateur, rien de plus simple, vous connaissez déjà la moitié de la syntaxe. Schématiquement, le code ressemblera à ça. Notez que le type de l'identificateur doit correspondre à celui du conteneur associé: on ne peut pas utiliser un itérateur de `std::vector<double>` pour parcourir un `std::vector<char>`.

```
1 // Dans le cas d'un conteneur modifiable.
2 std::/* Le conteneur utilisé. */::iterator identificateur { /*
   L'élément sur lequel pointer. */ };
3 // Dans le cas d'un conteneur const.
4 std::/* Le conteneur utilisé. */::const_iterator identificateur {
   /* L'élément sur lequel pointer. */ };
```


II. Le début du voyage

Pour raccourcir le code et se simplifier la vie, on peut bien entendu faire appel à l'inférence de type, à l'aide d'`auto`. C'est ce que je vais faire dans le reste de ce cours.

```
1 auto identificateur { /* L'élément sur lequel pointer. */ };
```

II.9.1.2. Début et fin d'un conteneur

Notez, dans la déclaration précédente, le commentaire `/* L'élément sur lequel pointer. */`. Deux fonctions, dans un premier temps, vont nous intéresser: `std::begin`, un curseur vers le **premier élément** du conteneur, et `std::end`, un curseur vers un **élément virtuel inexistant**, qui se trouve **après le dernier élément**.

?

Pourquoi ne pas pointer vers le dernier élément de la collection plutôt?

Pour une raison simple: comment faites-vous donc pour **déterminer que vous êtes à la fin de la collection**? En effet, si `std::end` pointe vers l'adresse en mémoire d'un élément valide de la collection, qu'est-ce qui vous prouve que vous êtes à la fin? C'est pour cela que `std::end` renvoie un itérateur sur un élément inexistant, qui indique la fin de la collection.

```
1 #include <vector>
2
3 int main()
4 {
5     std::vector<int> const tableau { -1, 28, 346 };
6     // Déclaration explicite d'un itérateur de std::vector
7     // d'entiers.
8     std::vector<int>::const_iterator debut_tableau {
9         std::begin(tableau) };
10    // Déclaration plus courte en utilisant auto.
11    auto fin_tableau { std::end(tableau) };
12
13    return 0;
14 }
```

II.9.1.3. Accéder à l'élément pointé

Nous savons comment pointer vers le premier élément de la collection, ainsi que signaler sa fin, mais ça ne nous avance pas vraiment. Il nous manque l'accès aux éléments. Pour cela, il faut **déréférencer** l'itérateur, grâce à l'étoile `*` devant celui-ci. L'exemple suivant affiche la valeur sur laquelle pointe l'itérateur, c'est-à-dire le premier élément du tableau, ici `T`.

II. Le début du voyage

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string tableau { "Tu texte." };
7     auto debut_tableau { std::begin(tableau) };
8
9     // On affiche le premier élément du tableau.
10    std::cout << *debut_tableau << std::endl;
11    // Et on peut même le modifier, comme ce qu'on fait avec les
12    // références, ou les crochets.
13    *debut_tableau = 'D';
14    // On vérifie.
15    std::cout << tableau << std::endl;
16
17    return 0;
18 }
```



Attention avec `std::end!`

Ne tentez pas de déréférencer `std::end!` Vous accéderiez à un espace mémoire inconnu, ce qui entraînera un comportement indéterminé, donc tout et n'importe quoi potentiellement. Je répète, ne tentez pas d'accéder à la valeur pointée par `std::end`, ni de la modifier.

II.9.1.4. Se déplacer dans une collection

II.10. Déplacement vers l'élément suivant

Maintenant que nous sommes capables d'afficher le premier élément, il serait bien de faire de même avec les autres. Nous pouvons, à partir d'un itérateur sur un élément, récupérer l'adresse de l'élément suivant en appliquant l'opérateur d'incrément `++`.

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> tableau { -1, 28, 346 };
7     auto debut_tableau { std::begin(tableau) };
8
9     std::cout << *debut_tableau << std::endl;
10
11     // On incrémente l'itérateur pour pointer sur l'élément
12     // suivant.
13     ++debut_tableau;
14
15     // On affiche le deuxième élément.
16     std::cout << *debut_tableau << std::endl;
17
18     return 0;
19 }
```

II.10.0.0.1. Utilisation concrète

Nous avons maintenant tout ce qu'il nous faut pour afficher l'ensemble d'une collection. L'exemple suivant vous montre comment parcourir un conteneur à l'aide d'une boucle `for` classique et d'itérateurs. Vous noterez l'utilisation d'un identifiant `it`, qui est l'équivalent de `i` quand nous avons vu les boucles.

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> tableau { -1, 28, 346, 84 };
```

II. Le début du voyage

```
7
8     for (auto it { std::begin(tableau) }; it != std::end(tableau);
9         ++it)
10     {
11         std::cout << *it << std::endl;
12     }
13     return 0;
14 }
```

II.10.0.0.2. Déplacement vers l'élément précédent

La plupart des conteneurs fournissent aussi l'opération inverse, c'est-à-dire récupérer l'adresse de l'élément précédent, grâce à l'opérateur de décrémentation `--`. Cet opérateur va, par exemple, nous permettre de récupérer le dernier élément d'un conteneur et l'afficher.

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector<int> tableau { -1, 28, 346, 84 };
7      auto itereur_dernier_element { std::end(tableau) };
8
9      // On récupère l'adresse de l'élément précédent.
10     --itereur_dernier_element;
11     // Et on affiche pour vérifier.
12     std::cout << *itereur_dernier_element << std::endl;
13
14     return 0;
15 }
```

II.10.0.0.3. Se déplacer de plusieurs éléments

Admettons que vous vouliez vous déplacer de plusieurs éléments, disons pour récupérer le cinquième. N'y a-t-il pas un autre moyen que d'utiliser cinq fois l'opérateur `++`? La réponse dépend du conteneur que vous utilisez. En effet, en fonction de celui-ci, les itérateurs derrières ne proposent pas exactement les mêmes fonctionnalités.

Les conteneurs comme `std::vector`, `std::array` ou `std::string`, que nous avons découvert plut tôt dans ce tutoriel, sont ce qu'on appelle des **conteneurs à accès direct** (*random access* en anglais), c'est-à-dire qu'on peut **accéder directement à n'importe quel élément**. Cela est vrai en utilisant les crochets `[]` (c'est même à leur présence qu'on reconnaît qu'un conteneur est à accès direct), et c'est également vrai pour les itérateurs. Ainsi, on peut directement écrire ceci quand on utilise un conteneur à accès direct.

II. Le début du voyage

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> tableau { -1, 28, 346, 84, -94, 31, 784 };
7     // Notez le +4 pour accéder au cinquième élément, car comme
8     // avec les crochets, on commence à 0.
9     auto it = std::begin(tableau) + 4;
10    std::cout << *it << std::endl;
11
12    return 0;
13 }
```

Il existe d'autres conteneurs, comme `std::list`, qui sont dit **bidirectionnels**, parce qu'ils autorisent le déplacement vers un élément précédent ou suivant, mais pas plusieurs d'un coup. Si vous remplacez, dans le code précédent, `std::vector` par `std::list`, vous verrez qu'il ne compilera pas.



Plusieurs types d'itérateurs

Il existe, dans la bibliothèque standard, **cinq types** différents d'itérateurs. Nous en avons déjà vu deux ici. Nous aurons l'occasion de voir les trois autres au fur et à mesure que nous progresserons dans le cours.

II.10.0.1. `const` et les itérateurs

Le concept d'itérateur réclame des explications supplémentaires concernant l'utilisation de `const`. En effet, avec les itérateurs, **on distingue la constance de l'itérateur lui-même de la constance de l'objet pointé**. Pour parler plus concrètement, appliquer `const` à un itérateur rend celui-ci constant, c'est-à-dire qu'on ne peut pas, une fois assigné, le faire pointer sur autre chose. Par contre, la valeur pointée est tout à fait modifiable.

```
1 #include <string>
2 #include <iostream>
3
4 int main()
5 {
6     std::string mot { "Donjour" };
7
8     // Itérateur constant sur un élément modifiable.
9     auto const it { std::begin(mot) };
10 }
```

II. Le début du voyage

```
11 // Ceci est impossible.
12 // it = std::end(mot);
13
14 // Ceci est parfaitement autorisé.
15 *it = 'B';
16
17 std::cout << mot << std::endl;
18 return 0;
19 }
```

Pour garantir la constance des éléments pointés, on va utiliser deux fonctions très proches de celles vues plus haut. Il s'agit de `std::cbegin` et de `std::cend`, pour *const begin* et *const end*. L'exemple suivant vous montre la différence.

```
1 #include <string>
2
3 int main()
4 {
5     std::string mot { "Donjour" };
6
7     // Itérateur constant sur un élément modifiable.
8     auto const iterateur_constant { std::begin(mot) };
9
10    // Ceci est impossible.
11    // it = std::end(mot);
12
13    // Ceci est parfaitement autorisé.
14    *iterateur_constant = 'B';
15
16    auto iterateur_non_constant { std::cbegin(mot) };
17
18    // Ceci est impossible.
19    // *iterateur_non_constant = 'D';
20
21    // Ceci est parfaitement autorisé.
22    iterateur_non_constant = std::cend(mot) - 1;
23
24    return 0;
25 }
```

On peut, bien entendu, combiner les deux formes pour obtenir un itérateur constant sur un élément constant.

II.10.0.2. Itérer depuis la fin

Il existe encore une troisième paire de fonctions, qui vont nous autoriser à parcourir une collection de la fin vers le début. Elles se nomment `std::rbegin` (*reverse begin*) et `std::rend` (*reverse*

II. Le début du voyage

`end`). Elles s'utilisent exactement comme les précédentes, leur utilisation est transparente. 🍊

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> const tableau { -1, 28, 346, 84 };
7
8     // Affiche depuis la fin, en commençant par 84.
9     for (auto it { std::rbegin(tableau) }; it !=
10         std::rend(tableau); ++it)
11     {
12         std::cout << *it << std::endl;
13     }
14     return 0;
15 }
```

Et comme pour les autres, les itérateurs peuvent être déclarés comme constant avec `std::crbegin` et `std::crend`.

?

Pourquoi avoir des itérateurs spéciaux? Pourquoi ne pas partir de `std::end` et itérer jusqu'à `std::begin`?

Pour vous répondre, je vais utiliser une illustration qu'a créé @gbdivers dans son cours C++, comme la licence m'y autorise.

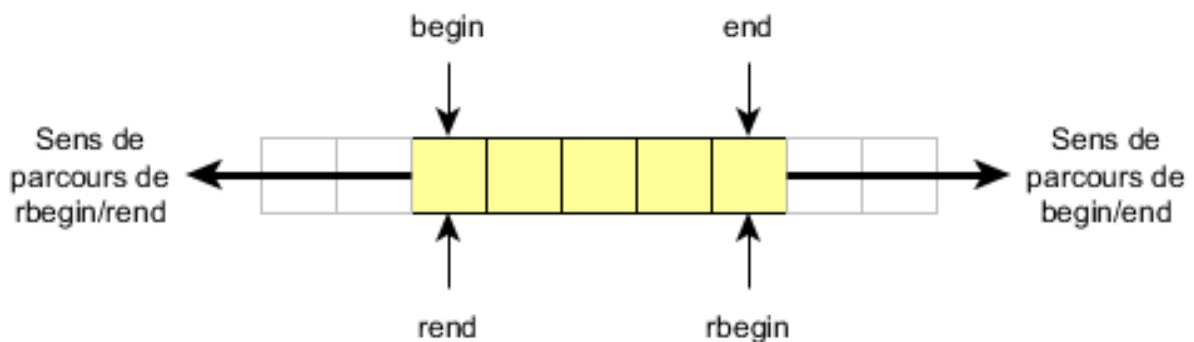


FIGURE II.10.1. – Illustration des itérateurs classiques et *reverses*.

Si l'on part de `std::end`, on va se déplacer vers la droite et jamais l'on ne retombera sur `std::begin`; le programme plantera bien avant. Voilà pourquoi il existe des itérateurs spécialement conçus pour parcourir une collection de la fin vers le début.



Alors, pourquoi ne pas partir de la fin et aller vers le début en décrémentant?

Vous pourriez... mais ça serait tellement plus pénible. Il faudrait gérer les cas spéciaux que sont la valeur de début et la valeur de fin. Ou bien réécrire tous les algorithmes pour prendre en compte la décrémentation. Ou bien... utiliser `std::rbegin` et `std::rend` de manière totalement transparente. 🍌

II.10.0.3. Utilisation conjointe avec les conteneurs

Terminons cette section en présentant une possibilité d'initialisation des conteneurs qui nous était inconnue jusque-là. Grâce aux itérateurs, nous pouvons facilement **créer des sous-ensembles à partir d'une collection d'éléments**, comme un tableau qui ne contiendrait que les éléments 2 à 5 d'un autre tableau. Cela se fait très simplement: il suffit de donner en argument, entre accolades {}, l'itérateur de début et l'itérateur de fin.

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> collection { 1, -8, 4, 654, -7785, 0, 42 };
7     // On passe l'itérateur de début et celui de fin.
8     std::vector<int> sous_ensemble { std::begin(collection) + 1,
9                                     std::begin(collection) + 5 };
10
11     // On affiche pour vérifier.
12     for (auto element : sous_ensemble)
13     {
14         std::cout << element << std::endl;
15     }
16
17     return 0;
18 }
```

Certains conteneurs, comme `std::vector`, `std::list` ou encore `std::string`, proposent également une fonction nommée `erase`, qui se base sur le principe des itérateurs pour supprimer plusieurs éléments d'une collection. On peut ainsi supprimer tous les éléments à partir du cinquième jusqu'à la fin, ou bien les trois premiers.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <string>
4 #include <vector>
```



```
5
6 int main()
7 {
8     std::vector<int> nombres { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
9     // Certains éléments.
10    nombres.erase(std::begin(nombres) + 2, std::begin(nombres) +
11                  5);
12
13    for (auto nombre : nombres)
14    {
15        std::cout << nombre << std::endl;
16    }
17
18    std::cout << std::endl;
19
20    std::string phrase { "Voici une phrase beurk !" };
21    // Jusqu'à la fin.
22    phrase.erase(std::begin(phrase) + 16, std::end(phrase));
23    std::cout << phrase << std::endl;
24
25    return 0;
26 }
```

II.10.1. Des algorithmes à gogo !

Maintenant, les itérateurs vont devenir vraiment intéressants et concrets, parce qu'ils sont pleinement employés par **les algorithmes** de la bibliothèque standard. Ces derniers, assez nombreux, sont des **fonctions déjà programmées et fortement optimisées pour manipuler des collections**, permettant ainsi de les trier, de récupérer tous les éléments satisfaisant une certaine condition, etc. Nous n'en verrons que quelques-uns dans ce chapitre, mais une fois le principe compris, vous n'aurez aucune difficulté à utiliser les autres. 🍊

i

Fichier à inclure

La plupart des algorithmes se trouvent dans le fichier d'en-tête `<algorithm>`, pensez donc à l'inclure.

II.10.1.1. `std::count` — Compter les occurrences d'une valeur

Commençons avec un algorithme simple, mais on ne peut plus utile: compter les occurrences d'une certaine valeur dans une collection. Cela se fait avec l'algorithme `std::count`. Celui-ci attend trois paramètres.

- Le premier est un itérateur pointant sur le début de la séquence.
- Le deuxième est un itérateur pointant sur la fin de la séquence.

II. Le début du voyage

— Le dernier est l'élément à compter.

La force de l'algorithme, c'est qu'il ne nous impose pas de chercher dans toute la collection. En effet, comme on manipule ici des itérateurs, rien ne nous empêche de compter les occurrences d'une valeur seulement entre le troisième et le septième élément de la collection, ou entre le cinquième et le dernier.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::string const phrase {
8         "Exemple illustrant le tutoriel C++ de Zeste de Savoir." };
9
10    // Toute la collection.
11    auto const total_phrase { std::count(std::begin(phrase),
12        std::end(phrase), 'e') };
13    std::cout << "Dans la phrase entière, il y a " << total_phrase
14        << " 'e' minuscule." << std::endl;
15
16    // Un sous-ensemble seulement de la collection.
17    auto const total_premier_mot { std::count(std::begin(phrase),
18        std::begin(phrase) + 7, 'e') };
19    std::cout << "Dans le premier mot, il y a " <<
20        total_premier_mot << " 'e' minuscule." << std::endl;
21
22    return 0;
23 }
```

Cet exemple utilise `std::string`, mais j'aurais tout aussi bien pu employer `std::vector` ou `std::array`. Par contre, `std::list` ne convient pas parce que ... oui! j'ai fait `std::begin(phrase) + 7`, ça suit dans le fond. Heureusement, une solution existe, que je vais vous montrer de suite.

II.10.1.2. `std::find`—Trouver un certain élément

Vous remarquerez que le code précédent n'est pas de très bonne qualité, puisque j'ai écrit `+7` directement, en dur. En plus du problème soulevé précédemment (ne pas pouvoir utiliser `std::list`), cela n'est pas très flexible: si le premier mot devient plus long ou plus court, je fausse les résultats. Alors qu'avec un itérateur qui pointerait sur le premier espace, on serait assuré d'avoir plus de souplesse. C'est justement ce que propose `std::find`, qui prend trois paramètres.

- Le premier est un itérateur pointant sur le début de la séquence.
- Le deuxième est un itérateur pointant sur la fin de la séquence.
- Le dernier est l'élément à rechercher.

II. Le début du voyage

Cette fonction retourne un **itérateur** sur le **premier élément** correspondant à la **recherche**, ou bien le deuxième argument si rien n'est trouvé.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::string const phrase {
8         "Exemple illustrant le tutoriel C++ de Zeste de Savoir." };
9
10    // On obtient un itérateur pointant sur le premier espace
11    // trouvé.
12    // Si l'on n'avait rien trouvé, on aurait obtenu
13    // std::end(phrase) comme valeur de retour.
14    auto const itérateur_premier_mot {
15        std::find(std::begin(phrase), std::end(phrase), ' ') };
16
17    // On peut donc s'en servir ici. Même si l'on modifie la
18    // longueur du premier mot, on est assuré que cette solution
19    // marche.
20    auto const total_premier_mot { std::count(std::begin(phrase),
21        itérateur_premier_mot, 'e') };
22    std::cout << "Dans le premier mot, il y a " <<
23        total_premier_mot << " 'e' minuscule." << std::endl;
24
25    return 0;
26 }
```

Je précise de nouveau pour être bien clair: `std::find` retourne un itérateur sur le **premier** élément trouvé, pas le deuxième ou le troisième. Mais avec une simple boucle, rien ne nous empêche de récupérer la deuxième occurrence, ou la troisième.

Cela me donne d'ailleurs une excellente idée d'exercice. En utilisant les deux fonctions que nous venons de voir, nous allons faire un programme qui compte le nombre de 'e' dans chaque mot d'une phrase. Essayez de coder par vous-mêmes ce petit exercice. Une solution est proposée ci-dessous.



Un indice

Petite astuce avant de commencer: on peut initialiser une chaîne de caractères avec un itérateur de début et de fin, comme n'importe quel autre conteneur.

II.10.1.3. `std::sort`—Trier une collection

L'algorithme `std::sort`, fonctionnant, comme les autres, avec une paire d'itérateurs, permet de trier dans l'ordre croissant une collection d'éléments, que ce soit des nombres, des caractères, etc. Bien entendu, cet algorithme ne marchera que sur des conteneurs modifiables, c'est-à-dire non `const`. Voici un exemple tout simple qui ne devrait guère vous surprendre.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <vector>
4
5 int main()
6 {
7     std::vector<double> constantes_mathematiques { 2.71828, 3.1415,
8                                                     1.0836, 1.4142, 1.6180 };
9     std::sort(std::begin(constantes_mathematiques),
10              std::end(constantes_mathematiques));
11
12     for (auto constante : constantes_mathematiques)
13     {
14         std::cout << constante << std::endl;
15     }
16
17     return 0;
18 }
```

Si vous avez essayé de changer `std::vector` en `std::list`, vous vous êtes rendu compte que le code ne compile plus. En effet, `std::sort` attend en paramètre des itérateurs à accès direct, comme ceux de `std::array`, `std::vector` et `std::string`.

À l'opposé, `std::list` n'utilise que des itérateurs bidirectionnels. Il est donc impossible de trier une `std::list` en utilisant cet algorithme.



Le cas de `std::list`

Le type `std::list` possède une fonction `sort` qui lui est propre et qui fait la même chose.



```

1 #include <iostream>
2 #include <list>
3
4 int main()
5 {
6     std::list<int> liste { 4, -8, 45, 2 };
7     liste.sort();
8
9     for (int i : liste)
10    {
11        std::cout << i << std::endl;
12    }
13    return 0;
14 }

```

II.10.1.4. `std::reverse`—Inverser l'ordre des éléments d'une collection

Parlons de `std::reverse`, qui inverse la position des éléments compris entre l'itérateur de début et l'itérateur de fin. Un exemple va vous montrer clairement ce qu'il fait.

```

1 #include <algorithm>
2 #include <iostream>
3 #include <list>
4
5 int main()
6 {
7     std::list<int> nombres { 2, 3, 1, 7, -4 };
8     std::cout << "Avant :\n";
9     for (auto nombre : nombres)
10    {
11        std::cout << nombre << std::endl;
12    }
13
14    std::reverse(std::begin(nombres), std::end(nombres));
15
16    std::cout << "\nAprès :\n";
17    for (auto nombre : nombres)
18    {
19        std::cout << nombre << std::endl;
20    }
21
22    return 0;
23 }

```

II.10.1.5. `std::remove`—Suppression d'éléments

Admettons maintenant que vous voulez supprimer toutes les lettres 'b' d'une chaîne. Avec `std::remove`, il n'y a aucun problème à ça. L'algorithme va supprimer toutes les occurrences de la lettre dans la chaîne. Et comme l'algorithme ne peut pas modifier la taille du conteneur, il nous renvoie un itérateur qui nous signale la fin de la collection. Il ne reste plus qu'à supprimer le tout grâce à `erase`.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::string paroles { "I'm blue, da ba dee da ba daa !" };
8
9     auto itérateur_fin { std::remove(std::begin(paroles),
10                                     std::end(paroles), 'b') };
11     paroles.erase(itérateur_fin, std::end(paroles));
12
13     std::cout << paroles << std::endl;
14     return 0;
15 }
```

II.10.1.6. `std::search`—Rechercher un sous-ensemble dans un ensemble

Cette fois, avec `std::search`, ça n'est pas un unique élément que nous allons chercher, mais bien un sous-ensemble dans un autre ensemble, comme, par exemple, un mot dans une phrase. Cette fois, pas moins de quatre arguments sont attendus.

- Le premier est l'itérateur de début de la collection à fouiller.
- Le deuxième est l'itérateur de fin.
- Le troisième est l'itérateur de début de la sous-collection à rechercher.
- Le quatrième est l'itérateur de fin.

Tout comme `std::find`, si rien n'est trouvé, le deuxième argument, l'itérateur de fin de la collection à parcourir, est renvoyé. Voici donc un exemple pour illustrer le fonctionnement de l'algorithme.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
```

```
7     std::string const phrase {
8         "Un exemple de phrase avec plusieurs mots." };
9     auto const debut { std::begin(phrase) };
10    auto const fin { std::end(phrase) };
11
12    // Rappel : std::boolalpha permet d'afficher 'true' ou 'false'
13    // et non '1' ou '0'.
14    std::cout << std::boolalpha;
15
16    std::string mot { "mot" };
17    std::cout << (std::search(debut, fin, std::begin(mot),
18        std::end(mot)) != fin) << std::endl;
19
20    mot = "exemple";
21    std::cout << (std::search(debut, fin, std::begin(mot),
22        std::end(mot)) != fin) << std::endl;
23
24    return 0;
25 }
```

II.10.1.7. `std::equal`—Vérifier l'égalité de deux ensembles

Comment testeriez-vous que deux ensembles sont équivalents, autrement dit qu'ils possèdent les mêmes éléments? La réponse vient du titre: avec l'algorithme `std::equal`. Celui-ci prend plusieurs paramètres.

- L'itérateur de début du premier ensemble.
- L'itérateur de fin du premier ensemble.
- L'itérateur de début du deuxième ensemble.
- L'itérateur de fin du deuxième ensemble.

Si les deux ensembles n'ont pas la même taille, alors l'égalité est forcément impossible. Sinon, l'algorithme va comparer les deux ensembles élément par élément.

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4
5  int main()
6  {
7      std::vector<int> const premier { 1, 2, 3, 4 };
8      std::vector<int> const second { 1, 2, 3, 4, 5 };
9      std::vector<int> const troisieme { 1, 2, 3, 4 };
10
11      std::cout << std::boolalpha;
12      std::cout << std::equal(std::begin(premier), std::end(premier),
13          std::begin(second), std::end(second)) << std::endl;
```

```
13     std::cout << std::equal(std::begin(premier), std::end(premier),  
14                             std::begin(troisieme), std::end(troisieme)) << std::endl;  
15 }
```

II.10.1.8. Et tant d'autres

Il y a tellement d'algorithmes différents que je ne vais pas tous vous les présenter. Vous en avez déjà vu plusieurs, et tous fonctionnent sur le même principe, celui **des itérateurs**. Utiliser les autres ne devrait donc pas vous poser de problème. 🍊

II.10.2. Personnalisation à la demande

Jusqu'ici, nos algorithmes sont figés. Par exemple, `std::sort` ne trie que dans l'ordre croissant. Mais les concepteurs de la bibliothèque standard ont pensé à ça et ont écrit leurs algorithmes de telle manière qu'on peut personnaliser leur comportement. Mine de rien, cette possibilité les rend encore plus intéressants! Voyons ça ensemble.

II.10.2.1. Le prédicat, à la base de la personnalisation des algorithmes

Un prédicat est **une expression qui prend, ou non, des arguments et renvoie un booléen**. Le résultat sera donc `true` ou `false`, si la condition est vérifiée ou non. Littéralement, un prédicat répond à des questions ressemblant à celles ci-dessous.

- Est-ce que l'élément est impair?
- Est-ce que les deux éléments `a` et `b` sont égaux?
- Est-ce que les deux éléments sont triés par ordre croissant?

La très grande majorité des algorithmes utilisent les prédicats, même ceux vu plus haut, sans que vous le sachiez. Par exemple, `std::sort` utilise, par défaut, le prédicat `<`, qui compare si un élément `a` est strictement inférieur à un autre élément `b`, ce qui explique pourquoi l'algorithme trie une collection dans l'ordre croissant. Mais il est tout à fait possible de lui indiquer de faire autre chose.

La bibliothèque standard contient déjà quelques prédicats, regroupés dans le fichier d'en-tête `<functional>`. Nous apprendrons, plus tard dans ce tutoriel, à écrire nos propres prédicats.

II.10.2.2. Trier une liste dans l'ordre décroissant

Faisons connaissance avec `std::greater`. Il s'agit d'un **foncteur**, c'est-à-dire **un objet qui agit comme une fonction**. Les détails internes ne nous intéressent pas pour l'instant, nous y reviendrons plus tard, quand nous aurons les connaissances requises.

II. Le début du voyage

Ce foncteur compare deux éléments et retourne `true` si le premier est plus grand que le deuxième, sinon `false`. Voici quelques exemples. Notez qu'on doit préciser, entre chevrons `<>`, le type des objets comparés.

```
1 #include <functional>
2 #include <iostream>
3
4 int main()
5 {
6     std::cout << std::boolalpha;
7     // On peut l'utiliser directement.
8     std::cout << std::greater<int>{}(2, 3) << std::endl;
9     std::cout << std::greater<int>{}(6, 3) << std::endl;
10    std::cout << std::greater<double>{}(0.08, 0.02) << std::endl;
11
12    // Ou bien à travers une variable, qu'on appelle comme une
13    // fonction.
14    std::greater<int> foncteur {};
15    std::cout << foncteur(-78, 9) << std::endl;
16    std::cout << foncteur(78, 8) << std::endl;
17
18    return 0;
19 }
```

Afin de personnaliser le comportement de `std::sort`, notre fonction de tri vue plus haut, il suffit de lui passer ce foncteur. Observez par vous-mêmes.

```
1 #include <algorithm>
2 #include <functional>
3 #include <iostream>
4 #include <vector>
5
6 int main()
7 {
8     std::vector<int> tableau { -8, 5, 45, -945 };
9     std::sort(std::begin(tableau), std::end(tableau),
10              std::greater<int> {});
11
12     for (auto x : tableau)
13     {
14         std::cout << x << std::endl;
15     }
16
17     return 0;
18 }
```

II.10.2.3. Somme et produit

Dans le fichier d'en-tête `<numeric>`, il existe une fonction répondant au doux nom de `std::accumulate`, qui permet de faire la somme d'un ensemble de valeurs, à partir d'une valeur de départ. Ça, c'est l'opération par défaut, mais on peut très bien lui donner un autre prédicat, comme `std::multiplies`, afin de calculer le produit des éléments d'un ensemble.

```
1 #include <functional>
2 #include <iostream>
3 #include <numeric>
4 #include <vector>
5
6 int main()
7 {
8     std::vector<int> const nombres { 1, 2, 4, 5, 10 };
9
10    // On choisit 0 comme élément de départ pour calculer la somme,
11    // car 0 est l'élément neutre de l'addition.
12    auto somme { std::accumulate(std::cbegin(nombres),
13    std::cend(nombres), 0) };
14    std::cout << somme << std::endl;
15
16    // On choisit 1 comme élément de départ pour calculer le
17    // produit, car 1 est l'élément neutre de la multiplication.
18    auto produit { std::accumulate(std::cbegin(nombres),
19    std::cend(nombres), 1, std::multiplies<int> {}) };
20    std::cout << produit << std::endl;
21
22    return 0;
23 }
```

II.10.2.4. Prédicats pour caractères

Il existe, dans le fichier d'en-tête `<cctype>`, des prédicats vérifiant des conditions **uniquement sur un caractère**. Ils permettent de vérifier si un caractère est une lettre, un chiffre, un signe de ponctuation, etc. Voyez par vous-mêmes.

```
1 #include <cctype>
2 #include <iostream>
3
4 int main()
5 {
6     char const lettre { 'A' };
7 }
```

II. Le début du voyage

```
8      std::cout << "Est-ce que " << lettre << " est une minuscule ? "
      << islower(lettre) << std::endl;
9      std::cout << "Est-ce que " << lettre << " est une majuscule ? "
      << isupper(lettre) << std::endl;
10     std::cout << "Est-ce que " << lettre << " est un chiffre ? " <<
      isdigit(lettre) << std::endl;
11
12     char const chiffre { '7' };
13
14     std::cout << "Est-ce que " << chiffre << " est un chiffre ? "
      << isdigit(chiffre) << std::endl;
15     std::cout << "Est-ce que " << chiffre <<
      " est un signe de ponctuation ? " << ispunct(chiffre) <<
      std::endl;
16
17     return 0;
18 }
```



Particularité héritée

Ces fonctions sont **héritées du C**, c'est pour cela qu'elles ne sont pas précédées de l'habituel `std::`. C'est également la raison pour laquelle elles renvoient des entiers et non des booléens, comme on aurait pu s'y attendre.

On peut faire des choses intéressantes avec ça. Par exemple, en utilisant l'algorithme `std::all_of`, qui vérifie que tous les éléments d'un ensemble respectent une propriété, on peut vérifier si tous les caractères d'une chaîne sont des chiffres, pour valider qu'il s'agit d'un numéro de téléphone.

```
1  #include <algorithm>
2  #include <cctype>
3  #include <iostream>
4  #include <string>
5
6  int main()
7  {
8      std::string const numero { "0185017204" };
9      if (std::all_of(std::cbegin(numero), std::cend(numero),
10                     isdigit))
11      {
12          std::cout << "C'est un numéro de téléphone correct." <<
              std::endl;
13      }
14      else
15      {
16          std::cout << "Entrée invalide." << std::endl;
17      }
```

```
18     return 0;  
19 }
```



Nos propres prédicats

Plus loin dans le cours, nous apprendrons à écrire nos propres prédicats, ce qui nous donnera encore plus de pouvoir sur nos algorithmes. Mais patience. 🍊

II.10.3. Exercices

II.10.3.1. Palindrome

Vous devez tester si une chaîne de caractères est un palindrome, c'est-à-dire un mot pouvant être lu indifféremment de gauche à droite ou de droite à gauche. Un exemple: «kayak.»

👁 Correction palindrome

II.10.3.2. `string_trim`—Suppression des espaces

Allez, un exercice plus dur, pour vous forger le caractère. Le but est de reproduire une fonction qui existe dans de nombreux autres langages, comme C# ou Python, et qui permet de supprimer les espaces, au début et à la fin d'une chaîne de caractères. Cela signifie non seulement les espaces ' ' mais aussi les tabulations '\t', les retours à la ligne '\n', etc. Pour vous aider, vous aller avoir besoin de deux choses.

- Le prédicat `isspace`, pour savoir si un caractère est un type d'espace ou non.
- L'algorithme `std::find_if_not`, qui permet de trouver, dans un ensemble délimité par deux itérateurs, le premier caractère qui ne respecte pas une condition.

Pour vous faciliter la vie, découper l'exercice en deux. Faites d'abord la fonction permettant de supprimer tous les espaces à gauche, puis pour ceux à droite. Bon courage. 🍊

👁 Correction `string_trim`

II.10.3.3. Couper une chaîne

Une autre opération courante, et qui est fournie nativement dans d'autres langages comme Python ou C#, consiste à découper une chaîne de caractères selon un caractère donné. Ainsi, si je coupe la chaîne "Salut, ceci est une phrase." en fonction des espaces, j'obtiens en résultat ["Salut,", "ceci", "est", "une", "phrase."].

Le but est donc que vous codiez un algorithme qui permet de faire ça. Vous allez notamment avoir besoin de la fonction `std::distance`, qui retourne la distance entre deux itérateurs, sous forme d'un nombre entier.

🕒 Correction découpage de chaînes

II.10.3.4. En résumé

- Les itérateurs sont des abstractions représentant un pointeur sur un élément d'un conteneur. Ils rendent ainsi l'usage du conteneur plus transparent.
- Deux fonctions, `std::begin` et `std::end`, sont extrêmement utilisées avec les itérateurs, car retournant un pointeur sur, respectivement, le début et la fin d'une collection.
- Les itérateurs demandent de la rigueur pour ne pas manipuler des éléments qui sont hors de la collection, ce qui entraînerait des comportements indéterminés.
- La bibliothèque standard contient des algorithmes, déjà programmés et très optimisés, qu'on peut appliquer à des collections.
- Nous pouvons les personnaliser grâce à des prédicats fournis dans la bibliothèque standard.

Contenu masqué

Contenu masqué n°25

```
1 #include <algorithm>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::string const phrase {
8         "Exemple illustrant le tutoriel C++ de Zeste de Savoir, mais un peu plu
9         };
10
11     // Pour garder en mémoire le début de chaque mot.
12     auto itérateur_precedent { std::begin(phrase) };

```

II. Le début du voyage

```
11     auto itérateur_espace { std::find(std::begin(phrase),
12                                     std::end(phrase), ' ') };
13
14     // Tant qu'on est pas à la fin de la phrase.
15     while (itérateur_espace != std::end(phrase))
16     {
17         std::string const mot { itérateur_precedent,
18                                 itérateur_espace };
19         auto const total_e { std::count(std::begin(mot),
20                                         std::end(mot), 'e') };
21
22         std::cout << "Dans le mot '" << mot << "', il y a " <<
23             total_e << " fois la lettre 'e'." << std::endl;
24
25         // On incrémente pour ne pas garder l'espace au début, car
26         // itérateur_espace pointe déjà sur un espace.
27         ++itérateur_espace;
28         // On met à jour notre itérateur de sauvegarde.
29         itérateur_precedent = itérateur_espace;
30         // On cherche la première occurrence dans le nouveau
31         // sous-ensemble.
32         itérateur_espace = std::find(itérateur_espace,
33                                     std::end(phrase), ' ');
34     }
35
36     // Une dernière fois pour analyser les caractères restants.
37     std::string const dernier_mot { itérateur_precedent,
38                                     std::end(phrase) };
39     std::cout << "Dans le dernier mot '" << dernier_mot <<
40         "' , il y a " << std::count(itérateur_precedent,
41                                     std::end(phrase), 'e') << " fois la lettre 'e'." <<
42         std::endl;
43
44     return 0;
45 }
```

[Retourner au texte.](#)

Contenu masqué n°26 : Correction palindrome

```
1 #include <algorithm>
2 #include <iostream>
3 #include <string>
4
5 int main()
```

II. Le début du voyage

```
6 {
7     std::string const s1 { "azerty" };
8     std::string const s2 { "radar" };
9
10    std::cout << std::boolalpha;
11    std::cout << std::equal(std::begin(s1), std::end(s1),
12                           std::rbegin(s1), std::rend(s1)) << std::endl;
12    std::cout << std::equal(std::begin(s2), std::end(s2),
13                           std::rbegin(s2), std::rend(s2)) << std::endl;
14
15    return 0;
16 }
```

L'astuce consiste à regarder si les deux ensembles de lettres sont les mêmes, en partant de la gauche vers la droite pour le premier (itérateur classique) et l'opposé pour le deuxième (*reverse iterator*).

[Retourner au texte.](#)

Contenu masqué n°27 :

Correction `string_trim`

```
1 #include <algorithm>
2 #include <cctype>
3 #include <iostream>
4 #include <string>
5
6 int main()
7 {
8     std::string test { "\n\tHello !\n\t" };
9     std::cout << "Avant modification : " << test << std::endl;
10
11    // On utilise l'itérateur fourni par std::find_if_not, qui
12    // pointe sur le premier élément qui n'est pas un espace.
13    auto premier_non_espace { std::find_if_not(std::begin(test),
14        std::end(test), isspace) };
15    test.erase(std::begin(test), premier_non_espace);
16    // On affiche pour tester.
17    std::cout << "Suppression au début : " << test << std::endl;
18
19    // En inversant, tous les espaces de fin se retrouvent au
20    // début.
21    std::reverse(std::begin(test), std::end(test));
22    // On reprend le même algorithme.
23    premier_non_espace = std::find_if_not(std::begin(test),
24        std::end(test), isspace);
25    test.erase(std::begin(test), premier_non_espace);
26    // On revient à l'état initial.
```

II. Le début du voyage

```
23     std::reverse(std::begin(test), std::end(test));
24     // On affiche pour tester.
25     std::cout << "Suppression à la fin : " << test << std::endl;
26
27     return 0;
28 }
```

Il est vrai que nous répétons deux fois les instructions pour supprimer les espaces à gauche. Laissez ça en l'état, nous verrons très bientôt une solution pour corriger ce fâcheux problème. 🍊

[Retourner au texte.](#)

Contenu masqué n°28 : Correction découpage de chaînes

```
1  #include <algorithm>
2  #include <iostream>
3  #include <string>
4  #include <vector>
5
6  int main()
7  {
8      std::string texte { "Voici une phrase que je vais couper." };
9      char const delimitateur { ' ' };
10     std::vector<std::string> parties {};
11
12     auto debut = std::begin(texte);
13     auto fin = std::end(texte);
14     auto itérateur = std::find(debut, fin, delimitateur);
15
16     while (itérateur != fin)
17     {
18         // Grâce à std::distance, nous obtenons la taille du mot.
19         std::string mot { debut, debut + std::distance(debut,
20                                                         itérateur) };
21
22         parties.push_back(mot);
23         // +1 pour sauter le délimiteur.
24         debut = itérateur + 1;
25         // Et on recommence.
26         itérateur = std::find(debut, fin, delimitateur);
27     }
28
29     // Ne pas oublier le dernier mot.
30     std::string mot { debut, debut + std::distance(debut,
31                                                         itérateur) };
32     parties.push_back(mot);
33 }
```


II. Le début du voyage

```
31
32 // On affiche pour vérifier.
33 for (std::string mot : parties)
34 {
35     std::cout << mot << std::endl;
36 }
37
38 return 0;
39 }
```

[Retourner au texte.](#)

II.11. Des flux dans tous les sens

Depuis le début de ce tutoriel, toutes les données que nous manipulons ont un point commun, qui est **leur volatilité**. Elles sont en effet stockées dans la mémoire vive de l'ordinateur, et celle-ci se vide dès qu'on l'éteint. Impossible, dans ces conditions, de sauvegarder des informations, comme une sauvegarde de jeu vidéo ou un résumé sur l'utilisateur.

Le but de ce chapitre est donc d'apprendre à **lire et écrire dans des fichiers** et, de manière plus générale, **dans des flux**.



Fichier à inclure

Pour manipuler des fichiers, il faut inclure le fichier d'en-tête `<fstream>`.

II.11.1. Avant-propos

Votre ordinateur contient énormément de fichiers, de tous les genres. Vos musiques préférées sont des fichiers, tout comme vos photos de vacances, votre CV Word ou LibreOffice, les fichiers de configuration de vos jeux vidéos, etc. Mêmes les programmes que vous utilisez sont des fichiers!

- Chaque fichier a **un nom**.
- Chaque fichier a **une extension**.

II.11.1.1. Prendrez-vous une extension ?

Il existe tout un tas d'extension de fichiers, pour chaque type qui existe. En voici quelques exemples.

- Fichiers audios: `.mp3`, `.oga`, etc.
- Fichiers vidéos: `.mp4`, `.avi`, etc.
- Images: `.jpg`, `.png`, `.gif`, etc.
- Exécutables: `.exe`, `.msi`, `.out`, etc.
- Fichiers textes: `.txt`, `.ini`, `.log`, etc.

Celles-ci sont surtout utiles pour nous, humains, afin de savoir quel type de fichier nous allons manipuler. Mais elles n'influencent en rien le contenu du fichier.

Certains fichiers, comme les fichiers textes, sont très simples à ouvrir et donne accès directement à l'information. D'autres, comme les `.png` ou les `.docx`, doivent être lus en respectant une certaine façon de faire, une norme, afin que l'information fasse sens. Dans le cadre de ce cours, **nous ne verrons que les documents textes simples**.

II.11.1.2. Vois sur ton chemin...

Afin de faciliter leur localisation et leur gestion, les fichiers sont classés et organisés sur leur support suivant un **système de fichiers**. C'est lui qui permet à l'utilisateur de répartir ses fichiers dans une arborescence de dossiers et de localiser ces derniers à partir d'un **chemin d'accès**.

Un chemin d'accès permet d'indiquer où est situé un fichier dans ledit système de fichier. Il contient forcément le nom du fichier concerné, mais également, si nécessaire, un ou plusieurs noms de dossiers, qu'il est nécessaire de traverser pour accéder au fichier depuis **la racine**.

La racine d'un système de fichier est le point de départ de l'arborescence des fichiers et dossiers. Sous GNU/Linux, il s'agit du répertoire / tandis que sous Windows, chaque lecteur est une racine (comme **C:** par exemple). Si un chemin d'accès commence par la racine, alors celui-ci est dit **absolu** car il permet d'atteindre le fichier depuis n'importe quelle position dans l'arborescence. Si le chemin d'accès ne commence pas par la racine, il est dit **relatif** et ne permet de parvenir au fichier que depuis un point précis dans la hiérarchie de fichiers.

Prenons un exemple. Nous avons un fichier `code_nucléaire.txt` qui se trouve dans un dossier `top_secret`, lui-même dans `documents`, qui se trouve à la racine. Le chemin absolu ressemblera à ça.

- Windows: `C:\documents\top_secret\code_nucléaire.txt`
- GNU/Linux: `/documents/top_secret/code_nucléaire.txt`

Par contre, si nous sommes déjà dans le dossier `documents`, alors nous pouvons nous contenter d'utiliser un chemin relatif. Ce chemin ne marchera cependant plus si nous sortons du dossier `documents`.

- Windows: `top_secret\code_nucléaire.txt`
- GNU/Linux: `top_secret/code_nucléaire.txt`

Le point `.` représente le dossier courant, c'est-à-dire `documents` dans notre exemple.

II.11.1.3. Un mot sur Windows

Depuis les années 1990, Windows supporte les chemins utilisant les slashes `/`. Si vous ouvrez l'explorateur de fichier et que vous écrivez `C:/Windows/System32`, ça marchera sans problème. Vous avez donc deux possibilités.

- Utilisez les slashes `/` si vous voulez que votre programme puisse fonctionner sur tous les systèmes.
- Utilisez les antislashes `\` si votre programme n'est destiné qu'à Windows.

Si vous décidez d'utiliser ces derniers, n'oubliez pas de les doubler en écrivant `\\`, car `\` seul est un caractère d'échappement. Ou bien utilisez les chaînes brutes.

II.11.1.4. Pas d'interprétation, on est des brutes

Utiliser les séquences d'échappement peut alourdir le code, en plus d'être casse-pieds à écrire. C++ offre une solution alternative qui se nomme **chaînes brutes** (de l'anglais *raw strings*). Par défaut, une chaîne brute commence par `R"` (et termine par `"`). Tout ce qui se trouve entre ces deux séquences n'est pas interprété, mais tout simplement **ignoré**.

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout <<
6         R"(Elle a dit "Va dans le dossier C:\Program Files" et regarde.)"
7         << std::endl;
8     return 0;
9 }
```

Et si jamais notre chaîne contient le délimiteur de fin, on peut personnaliser les délimiteurs pour qu'ils soient différents. Pour ça, on ajoute les caractères qu'on souhaite entre les guillemets et les parenthèses.

```
1 #include <iostream>
2
3 int main()
4 {
5     // Oups, ça ne marche pas...
6     //std::cout << R"(Il a écrit "f(x)" au tableau.)" << std::endl;
7
8     // Heureusement, je peux choisir moi-même mes délimiteurs.
9     // Ici, tout ce qui est entre "&( et )&" n'est pas interprété.
10    std::cout << R"&(Il a écrit "f(x)" au tableau.)&" << std::endl;
11    return 0;
12 }
```

II.11.2. `std::ofstream`— Écrire dans un fichier

Commençons par apprendre à écrire dans un fichier. Cela nous est utile si nous voulons sauvegarder un score, les paramètres de configuration d'une application ou n'importe quel résultat de notre programme. Et pour ce faire, nous allons utiliser le type `std::ofstream`, abréviation de «*output file stream*», ce qui signifie «flux fichier de sortie».

II.11.2.1. Ouvrir le fichier

Il attend en argument le nom du fichier à ouvrir. Par défaut, si le fichier n'existe pas, il est créé.

```
1 #include <fstream>
2
3 int main()
4 {
5     // Je veux ouvrir un fichier nommé 'sortie.txt', qui se trouve
        dans le dossier du projet.
6     // S'il existe, il sera ouvert. Sinon, il sera d'abord créé
        puis ouvert.
7     std::ofstream fichier { "sortie.txt" };
8     return 0;
9 }
```

II.11.2.2. Écriture

Pour écrire dedans, c'est très simple, il suffit de réutiliser les chevrons, comme avec `std::cout`. Voyez par vous-mêmes.

```
1 #include <fstream>
2
3 int main()
4 {
5     std::ofstream fichier { "sortie.txt" };
6     // On écrit un 3, un espace et un 4.
7     fichier << 3 << " " << 4;
8
9     return 0;
10 }
```

II. Le début du voyage

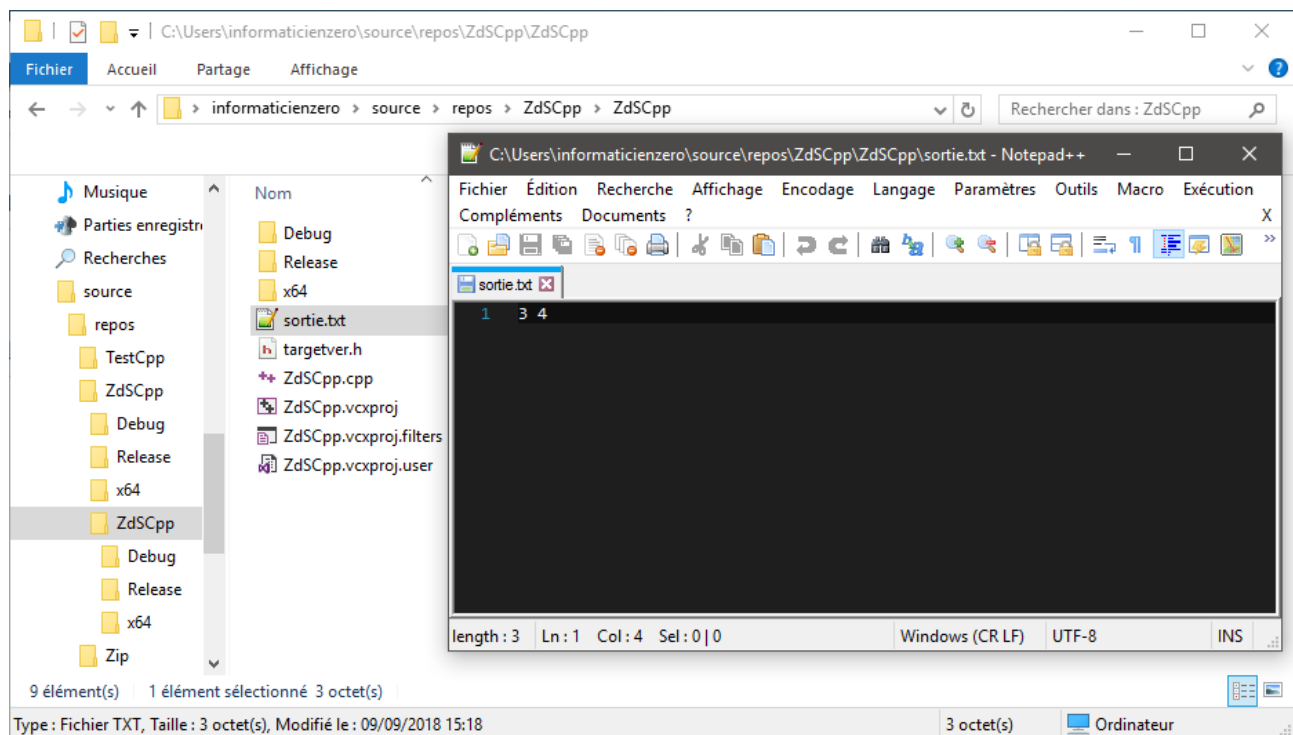


FIGURE II.11.1. – On voit que les deux valeurs, ainsi que l'espace, ont bien été écrites.

Et, exactement comme pour `std::cout`, nous pouvons écrire des littéraux, mais aussi des variables et des expressions.

```
1 #include <fstream>
2 #include <string>
3
4 int main()
5 {
6     std::ofstream fichier { "sortie.txt" };
7     fichier << 3 << " " << 4;
8
9     int x { 0 };
10    // On va à la ligne puis on écrit une équation.
11    fichier << '\n' << x << " + 2 = " << x + 2;
12
13    // Pas de problème avec le texte non plus.
14    std::string texte { "Voici une phrase." };
15    fichier << '\n' << texte;
16
17    return 0;
18 }
```



J'ai essayé d'écrire une phrase avec des accents et j'ai obtenu un texte avec des caractères bizarres. Que s'est-il passé?

II. Le début du voyage

Normalement, les Linuxiens sont épargnés par ce problème et seuls ceux utilisant Windows sont concernés (pas de chance). En fait, c'est une histoire d'**encodage**.

Sans rentrer dans les détails, car [un cours entier](#) existe déjà sur le sujet, il existe différents encodages qui permettent de stocker plus ou moins de caractères. Le premier inventé (années 1960), l'**ASCII**, ne permet de stocker que 127 caractères, dont les 26 lettres de l'alphabet anglais, mais pas nos accents français, le **ñ** espagnol, le **ß** allemand, etc. D'autres sont apparus et permettent de stocker plus de caractères, dont **UTF-8**.

Par défaut en C++, quand on manipule une chaîne de caractères, celle-ci est interprétée comme étant de l'**ASCII**. Comme nos accents français ne sont pas représentables, on obtient du grand n'importe quoi en sortie. La solution consiste à utiliser UTF-8, ce qui se fait en précisant **u8** devant nos chaînes de caractères.

```
1 #include <fstream>
2 #include <iostream>
3 #include <string>
4 #include <vector>
5
6 int main()
7 {
8     std::vector<std::string> const phrases
9     {
10         u8"Voici un mot important.\n",
11         u8"Russe : резиновые перчатки",
12         u8"Polonais : gumowe rękawiczki",
13         u8"Roumain : mănuși de cauciuc"
14     };
15
16     std::ofstream fichier { "sortie.txt" };
17     for (auto const & phrase : phrases)
18     {
19         fichier << phrase << std::endl;
20     }
21
22     return 0;
23 }
```

Enfin, il faut enregistrer le code source en UTF-8. Pour se faire, suivez les instructions ci-dessous.

- Sous Visual Studio, cliquez sur **Fichier** -> **Enregistrer main.cpp sous...**, puis cliquez sur la petite flèche à côté du bouton **Enregistrer** et sélectionnez l'option **Enregistrer avec encodage...**. Ensuite, pour le champs «Encodage», choisissez la valeur «**Unicode (UTF-8 avec signature) - Page de codes 65001**» et validez.

Il ne vous reste plus qu'à lancer et voir le résultat.

II. Le début du voyage

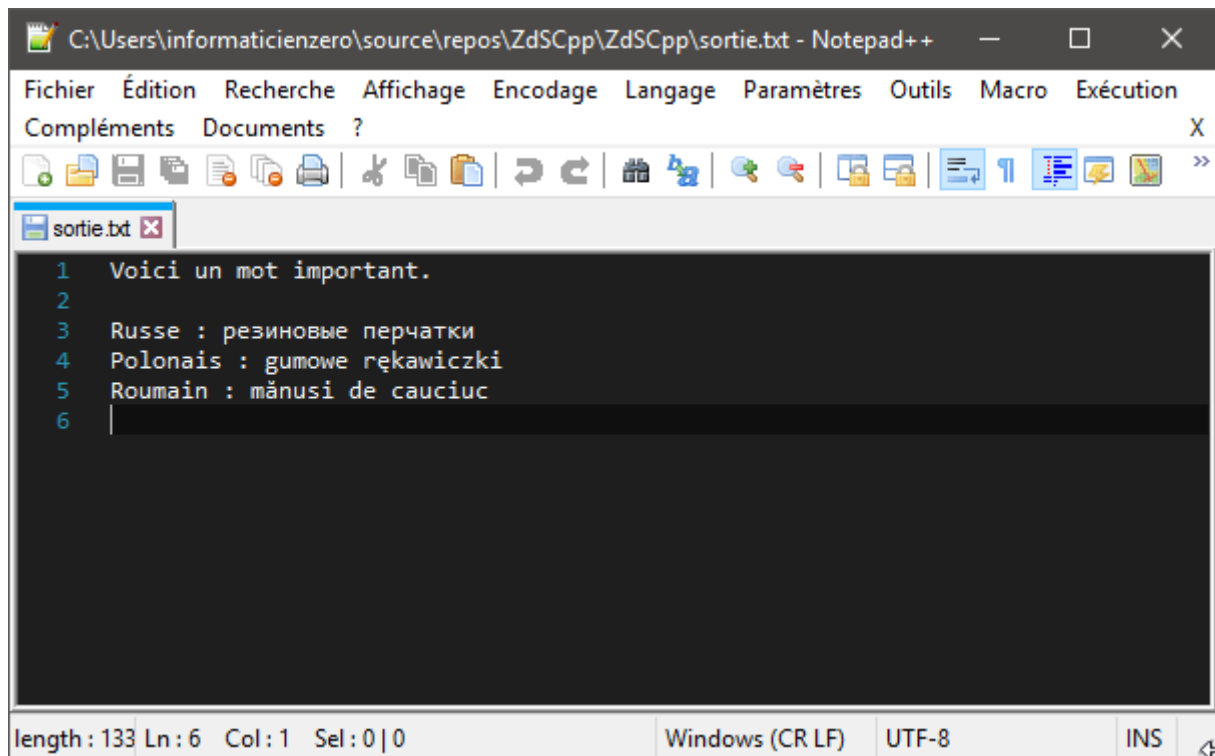


FIGURE II.11.2. – Hop, plus de soucis, ça marche maintenant.

II.11.2.3. Ouvrir sans effacer

Essayer de modifier le fichier vous-mêmes, avec votre éditeur de texte préféré, et ajouter au tout début une phrase quelconque. Lancez le programme, rouvrez le fichier et horreur! votre texte a disparu. 🤖

En fait, par défaut, `std::ofstream` tronque le contenu du fichier. Autrement dit, s'il y avait des données écrites précédemment, elles sont perdues dès que le programme rouvrira le fichier. Mais il est possible de régler ce problème très simplement.

Lorsque nous initialisons notre objet `std::ofstream`, nous pouvons lui donner un deuxième argument qui est le mode d'ouverture. Dans notre cas, ce mode est `std::ios::app`. Ici, `ios` veut dire *Input Output Stream*, c'est-à-dire *flux d'entrée et de sortie*; `app` est le diminutif de *append*, qui veut dire ajouter en anglais.

Notre contenu n'est plus effacé et les données supplémentaires sont *ajoutées* à la suite.

```
1 #include <fstream>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::ofstream fichier { "sortie.txt", std::ios::app };
8     fichier << 3 << " " << 4;
```



```
9
10     int x { 0 };
11     fichier << '\n' << x << " + 2 = " << x + 2;
12
13     std::string texte { u8"Voici une phrase." };
14     fichier << '\n' << texte;
15
16     return 0;
17 }
```

II.11.3. std::ifstream— Lire dans un fichier

Abordons maintenant la lecture, opération elle aussi ô combien importante. Cela nous permet de charger des paramètres, d'analyser des fichiers, etc. Pour ce faire, nous allons utiliser le type `std::ifstream`, abréviation de «*input file stream*», ce qui signifie «flux fichier d'entrée».

II.11.3.1. Ouvrir le fichier

L'ouverture se fait de la même façon que pour `std::ofstream`, c'est-à-dire en donnant le nom du fichier à ouvrir. Sauf que cette fois, il faut que **le fichier existe déjà**. C'est logique: vous ne pouvez pas lire un fichier qui n'existe pas.

Créez donc avec moi un fichier appelé `entrée.txt`.

```
1 #include <fstream>
2
3 int main()
4 {
5     std::ifstream fichier { "entrée.txt" };
6     return 0;
7 }
```

II.11.3.2. Lecture

Écrivez avec moi, dans le fichier `entrée.txt`, un entier, puis un retour à la ligne, puis un mot.

```
1 42
2 Bonjour
```

II. Le début du voyage

La lecture se fait très simplement. On utilise les chevrons comme pour `std::cin`. Voyez par vous-mêmes.

```
1 #include <fstream>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::ifstream fichier { "entrée.txt" };
8
9     int entier { 0 };
10    fichier >> entier;
11    std::cout << "Mon entier vaut : " << entier << std::endl;
12
13    std::string mot { "" };
14    fichier >> mot;
15    std::cout << "Mon mot vaut : " << mot << std::endl;
16
17    return 0;
18 }
```

?

J'avais dans mon fichier une phrase entière, et seul le premier mot est apparu dans la console. Que s'est-il passé?

En fait, c'est un comportement tout à fait normal, qu'on retrouve également avec `std::cin`. Par défaut, les espaces et les retours à la ligne sont considérés comme étant des **délimiteurs**, des **séparateurs**. Ainsi, là où vous voyez une phrase complète, le programme va voir un ensemble de mots séparés par des espaces.

Les concepteurs de C++ ne nous laissent néanmoins pas sans solution. Il existe une fonction, `std::getline`, qui permet de lire une ligne entière. Elle attend le flux d'entrée à lire, ainsi qu'une `std::string` pour écrire le résultat.

```
1 #include <fstream>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::ifstream fichier { "entrée.txt" };
8
9     int entier { 0 };
10    fichier >> entier;
11    std::cout << "Mon entier vaut : " << entier << std::endl;
```

II. Le début du voyage

```
12
13     std::string phrase { "" };
14     std::getline(fichier, phrase);
15     std::cout << "Ma phrase vaut : " << phrase << std::endl;
16
17     return 0;
18 }
```



Ça ne marche toujours pas! Pourquoi je ne vois plus rien maintenant?

Vicieux comme je suis, je vous ai encore fait tomber dans un piège. 🦊

Le problème vient de l'utilisation, juste avant `std::getline`, des chevrons. Ceux-ci lisent la valeur entière, pas de soucis, mais **le caractère de retour à la ligne est laissé**. Si l'on refait encore cette opération avec les chevrons, alors ce caractère est «mangé» et ne nous gêne pas. Mais ce n'est pas le cas avec `std::getline`.

Pour régler ce problème, nous avons plusieurs solutions.

- Nous pouvons utiliser `fichier.ignore(255, '\n')` après l'utilisation des chevrons, comme nous l'avons appris dans les T.P de gestion des erreurs d'entrée. Ainsi, nous sommes sûrs de supprimer le retour à la ligne qui nous embête.
- Nous pouvons utiliser le **modificateur de flux** (c'est-à-dire qui modifie son comportement) `std::ws` (de l'anglais «*white spaces*», espaces blancs). Il supprime tous les espaces et retours à la ligne qu'il trouve et s'utilise comme suit.

```
1 // On peut l'utiliser directement, en même temps que la lecture.
2 int entier { 0 };
3 fichier >> entier >> std::ws;
4 std::cout << "Mon entier vaut : " << entier << std::endl;
5
6
7 // On peut en faire une instruction à part.
8 fichier >> std::ws;
9
10
11 // Ou même directement avec std::getline !
12 std::getline(fichier >> std::ws, phrase);
```

Maintenant, en complétant le code précédent, nous n'avons plus aucun problème.

```
1 #include <fstream>
2 #include <iostream>
3 #include <string>
4
```

II. Le début du voyage

```
5 int main()
6 {
7     std::ifstream fichier { "entrée.txt" };
8
9     int entier { 0 };
10    fichier >> entier;
11    std::cout << "Mon entier vaut : " << entier << std::endl;
12
13    std::string phrase { "" };
14    // On est sûr de ne pas oublier en l'utilisant directement dans
15    // std::getline.
16    std::getline(fichier >> std::ws, phrase);
17    std::cout << "Ma phrase vaut : " << phrase << std::endl;
18
19    return 0;
20 }
```

II.11.3.3. Tout lire

Vous êtes un lecteur vorace et vous voulez lire tout le fichier? Aucun problème. Tant les chevrons (mot par mot) que `std::getline` (ligne par ligne) peuvent être utilisés dans une boucle, pour lire jusqu'à la fin du fichier.

```
1 #include <fstream>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::ifstream fichier { "entrée.txt" };
8     std::string ligne { "" };
9
10    while (std::getline(fichier, ligne))
11    {
12        std::cout << "Ligne lue : " << ligne << std::endl;
13    }
14
15    return 0;
16 }
```

Remarquez la syntaxe de la boucle: `while (std::getline(fichier, ligne))`. Cela devrait vous rappeler notre utilisation de `std::cin` dans les boucles (et plus généralement dans les conditions). Ce parallèle est dû au fait que la fonction `std::getline` renvoie un flux; plus précisément, elle renvoie le flux qu'on lui donne en paramètre (ici, le fichier ouvert), dont l'état a donc été modifié. C'est ce flux qui est alors converti en booléen pour nous dire si tout s'est bien

II. Le début du voyage

passé. En particulier, lorsqu'on a atteint la fin du fichier, le flux renvoyé par `std::getline` est évalué à `false`, ce qui permet de sortir de la boucle en fin de lecture de fichier.

II.11.4. Exercice

II.11.4.1. Statistiques sur des fichiers

Quand vous écrivez avec un logiciel de traitement de texte, celui-ci vous donne en direct certaines informations, telles que le nombre de caractères, de mots, de paragraphes, etc. Faisons donc un petit programme qui va ouvrir un fichier et obtenir les informations suivantes.

- Le nombre de lignes.
- Le nombre de caractères (sans les espaces).
- Le nombre de mots.

🕒 Correction statistiques sur des fichiers

II.11.5. Encore plus de flux !

Nous savons lire et écrire dans des fichiers, ce qui est déjà bien. Mais allons un peu plus loin, voulez-vous? Nous allons découvrir que la manipulation de fichiers et celle des entrées / sorties ont beaucoup plus de points communs que ce que vous auriez pu penser. En effet, dans les deux cas, on parle de **flux**.

II.11.5.1. Un flux c'est quoi ?

Il existe de très nombreux composants électroniques. Que ce soient des processeurs, des cartes-mères, des claviers, des écrans, des téléphones, une multitude s'offre à nous. Et tous viennent avec leur propre façon de communiquer. Écrire un programme qui devrait savoir communiquer avec tous ces périphériques est impossible. C'est pour cela qu'on utilise **des abstractions**.

Par exemple, votre **système d'exploitation** (Windows, GNU/Linux, Android, etc) est une abstraction. Notre programme ne communique plus avec un nombre potentiellement très grand de matériels et composants, mais uniquement avec le système d'exploitation, qui se charge lui-même de **faire le lien avec le matériel** (la mémoire vive, le processeur, le disque dur, etc). Cela nous simplifie grandement la vie.

La **bibliothèque standard** est aussi une abstraction. Elle nous fournit un lot unique de fonctionnalités précises, qui marcheront **peu importe le système d'exploitation** utilisé. Ainsi, en utilisant la bibliothèque standard, notre programme pourra être compilé pour fonctionner tant avec Windows, que GNU/Linux, qu'Android, etc.

Les flux sont eux aussi une abstraction. Grâce à eux, on peut envoyer ou recevoir des «flux de données» (d'où le nom), d'une taille potentiellement infinie, à des composants. Dans la

II. Le début du voyage

bibliothèque standard, nous avons ainsi de quoi recevoir des informations du clavier (`std::cin`), d'en envoyer à l'écran (`std::cout`) mais aussi avec des fichiers (`std::ifstream` et `std::ofstream`).

L'avantage, c'est qu'ils offrent une façon **transparente** et **unifiée** d'envoyer et de recevoir des données. Vous l'avez vu vous-mêmes plus haut, il n'y a aucune différence entre écrire sur la sortie standard ou un fichier de notre choix. On pourrait même coder nos propres flux de façon à lire et écrire sur le réseau, qui s'utiliseraient de la même façon que les autres!

II.11.5.2. Un *buffer* dites-vous ?

Les flux sont, selon le terme anglophone largement employé, *bufferisé*. Le problème est qu'accéder à des composants externes, des périphériques, comme la carte graphique, un appareil externe ou autre, peut être **long**. Ainsi, dans le code suivant, on accéderait potentiellement trois fois à un composant externe.

```
1 flux << 1 << 'A' << 3.1415;
```

Pour éviter ça, tout ce qu'on envoie ou reçoit d'un flux n'est pas directement traité mais mis en mémoire dans un tampon (en anglais «*buffer*»). Pour que les données soient concrètement et physiquement envoyées à l'écran, ou écrites sur un fichier ou autre, il faut **vider le tampon** (en anglais «*flush*»). C'est notamment ce que fait `std::endl`, en plus d'afficher un retour à la ligne. C'est également ce qui se passe quand le fichier est fermé, quand notre objet arrive à la fin de sa portée et est détruit.

Tenez, faisons un test. Compilez le code suivant et, avant de taper une phrase quelconque en guise d'entrée, ouvrez le fichier concerné et regardez son contenu. Vous le trouverez vide, car les données sont toujours dans le *buffer*.

```
1 #include <fstream>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::ofstream fichier { "sortie.txt" };
8     // Notez que je ne vide pas le tampon.
9     fichier << "Hey, salut toi !\n";
10    fichier << 42 << " " << 2.718;
11
12    // Une entrée pour vous laisser le temps d'ouvrir le fichier
13    // sortie.txt.
14    std::string phrase { "" };
15    std::cout << "Tape une phrase quelconque : ";
16    std::cin >> phrase;
```

II. Le début du voyage

```
17 // Je vide explicitement le tampon. Maintenant les données sont
    écrites.
18 fichier << std::flush;
19 return 0;
20 }
```



Retour à la ligne

Pour éviter ce phénomène parfois coûteux, certains codes remplacent `std::endl` par `'\n'`, pour éviter de *flusher* trop souvent. Dans le cadre de ce cours, ces questions de performances ne nous intéressent pas, donc nous continuerons à utiliser en alternance `std::endl` et `'\n'`.

II.11.5.3. Les modificateurs de flux

Nous avons déjà vu un modificateur de flux au début de ce cours: `std::boolalpha`. Il permet d'afficher `true` ou `false` au lieu de `1` ou `0` quand on manipule des booléens. Il est tout à fait possible de l'utiliser avec d'autres types de flux. L'exemple suivant vous le démontre.

```
1 #include <fstream>
2
3 int main()
4 {
5     std::ofstream fichier { "sortie.txt" };
6     // On va écrire true.
7     fichier << std::boolalpha << true << std::endl;
8     // On revient comme avant et on va écrire 0.
9     fichier << std::noboolalpha << false << std::endl;
10
11     return 0;
12 }
```



Modificateurs par paire

Le modificateur `std::noboolalpha` fait l'opération inverse de `std::boolalpha`. La plupart des modificateurs sont ainsi disponibles «en paire», l'un activant le comportement désiré, l'autre le désactivant.

Voici ce que vous allez obtenir dans le fichier de sortie.

II. Le début du voyage

```
1 true
2 0
```

Examinons un autre exemple. Vous savez que, si vous affichez dans la console un nombre positif, vous le verrez tel quel. Mais on peut aussi demander à afficher le signe `+`, tout comme le signe `-` est affiché pour les nombres négatifs.

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << std::showpos << 42 << " " << 89 << " " << -48 <<
6     std::endl;
7     return 0;
8 }
```

```
1 +42 +89 -48
```

II.11.5.4. Même les chaînes y passent !

Maintenant, admettons que vous vouliez avoir une chaîne de caractères représentant un nombre réel. Mais, contraintes supplémentaires, vous voulez que le signe soit toujours affiché et que le nombre soit écrit en utilisant la [notation scientifique](#) [↗](#). Comment faire ?

Il y a bien `std::scientific` et `std::showpos`, mais ils n'existent que pour les flux. Justement, la solution va être de **traiter les chaînes de caractères comme des flux**. Cela se fait en utilisant `std::ostringstream` comme flux de sortie et `std::istringstream` comme flux d'entrée. Les deux sont disponibles dans le fichier `<sstream>`.

```
1 #include <iostream>
2 #include <sstream>
3 #include <string>
4
5 int main()
6 {
7     double const reel { 10245.789 };
8
9     std::ostringstream flux_chaine;
10    // Notez comment l'utilisation est identique à ce que vous
11    // connaissez.
```


II. Le début du voyage

```
11     flux_chaine << std::scientific << std::showpos << reel <<
12         std::endl;
13     // On récupère une chaîne de caractères en appelant str().
14     std::string resultat { flux_chaine.str() };
15
16     std::cout << "Affichage par défaut : " << reel << std::endl;
17     std::cout << "Affichage modifié : " << resultat << std::endl;
18
19     return 0;
20 }
```

```
1 Affichage par défaut : 10245.8
2 Affichage modifié : +1.024579e+04
```

Maintenant, abordons `std::istringstream`, qui permet de traiter une chaîne de caractères comme un flux, avec un exemple concret. En informatique, beaucoup de valeurs sont écrites suivant le système [hexadécimal](#). Que ce soit l'identifiant d'une carte réseau ou le codage des couleurs sur un site web, il est courant de l'utiliser.

i

L'hexadécimal

Le système hexadécimal est un système de numération qui utilise **16 chiffres** au lieu des 10 que nous connaissons dans le système décimal. Les chiffres vont de 0 à 9, comme pour le décimal, puis on utilise les lettres A, B, C, D, E et F. Ainsi, des nombres comme 2A et FD sont **des nombres écrits en hexadécimal**, qui valent respectivement 42 et 253 dans notre système décimal.

Si vous souhaitez en savoir plus, lisez l'article [Wikipédia](#), ainsi que d'autres ressources. Pour ma part, je ne vais pas rentrer plus dans les détails que ça.

Par exemple, l'orange agrume utilisé sur Zeste de Savoir est écrit **f8ad32** dans son code source. Imaginons donc que nous récupérions cette valeur. C'est une chaîne de caractères et nous voulons en déterminer le niveau de rouge, de vert et de bleu.

Il existe un modificateur de flux appelé `std::hex` qui permet de dire au flux concerné que les nombres qu'il va extraire utilisent le système hexadécimal et non décimal. En traitant une chaîne de caractères comme un flux, on résout alors notre problème.

```
1 #include <iostream>
2 #include <sstream>
3
4 int main()
5 {
6     std::istringstream flux_entree { "f8 ad 32" };
7     int rouge { 0 };
8     int vert { 0 };
```

II. Le début du voyage

```
9      int bleu { 0 };
10
11      flux_entree >> std::hex >> rouge >> vert >> bleu;
12      std::cout << "Niveau de rouge : " << rouge << std::endl;
13      std::cout << "Niveau de vert : " << vert << std::endl;
14      std::cout << "Niveau de bleu : " << bleu << std::endl;
15
16      return 0;
17 }
```

```
1 Niveau de rouge : 248
2 Niveau de vert : 173
3 Niveau de bleu : 50
```



Zeste d'histoire

On utilisait beaucoup `std::ostringstream` et `std::istringstream` avant C++11 pour convertir des nombres en chaînes de caractères et vice-versa. D'autres méthodes existent maintenant, mais nous en parlerons plus tard.

II.11.5.5. En résumé

- Pour ouvrir un fichier, il faut préciser son chemin, qui peut être relatif ou absolu.
- On écrit dans un fichier en utilisant `std::ofstream`.
- On lit dans un fichier en utilisant `std::ifstream`.
- Les flux sont une abstraction permettant d'envoyer ou recevoir un flux de données à des composants quelconques.
- Les flux sont bufferisés.
- Il existe des flux pour communiquer avec des fichiers (`std::ifstream` et `std::ofstream`), avec des chaînes de caractères (`std::istringstream` et `std::ostringstream`), avec l'entrée et la sortie standards (`std::cin` et `std::cout`).
- On utilise des modificateurs de flux pour en modifier le comportement.

Voilà, nous avons vu les bases et vous commencez déjà à faire des choses intéressantes. Mais vous ne connaissez pas tout, loin de là. Par exemple, nous ne savons pas comment découper notre programme. En effet, nous écrivons tout dans le `main`. Heureusement, la prochaine partie est là pour nous.

Contenu masqué


Contenu masqué n°29 :

Correction statistiques sur des fichiers

```
1 #include <fstream>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::ifstream fichier { "fichier_1.txt" };
8     std::string ligne { "" };
9
10    int total_lignes { 0 };
11    int total_caracteres { 0 };
12    int total_mots { 0 };
13
14    bool espace_caractere_precedent { false };
15    while (std::getline(fichier, ligne))
16    {
17        ++total_lignes;
18
19        for (char c : ligne)
20        {
21            if (isspace(c))
22            {
23                // Si le précédent n'était pas un espace, alors
24                // c'était une partie de mot.
25                if (!espace_caractere_precedent)
26                {
27                    ++total_mots;
28                }
29
30                espace_caractere_precedent = true;
31            }
32            else
33            {
34                ++total_caracteres;
35                espace_caractere_precedent = false;
36            }
37        }
38
39        // Ne pas oublier de regarder pour l'éventuel dernier mot.
40        if (!espace_caractere_precedent)
41        {
42            ++total_mots;
43        }
44    }
45}
```

II. Le début du voyage

```
42         espace_caractere_precedent = true;
43     }
44 }
45
46 std::cout << "Total de caractères : " << total_caracteres <<
    std::endl;
47 std::cout << "Total de lignes : " << total_lignes << std::endl;
48 std::cout << "Total de mots : " << total_mots << std::endl;
49
50 return 0;
51 }
```

Félicitations, vous avez déjà codé une ébauche de [wc](#) , un programme UNIX existant. 🍊
[Retourner au texte.](#)

Troisième partie

On passe la deuxième !

III. On passe la deuxième!

Depuis les débuts de l'informatique et de la programmation, découper son code en sous-morceaux est un principe de base. Celui-ci permet de mieux diviser le travail, résoudre les problèmes plus facilement, aide les développeurs à mieux s'y retrouver. En bref, **plein d'avantages**.

Cette partie va non seulement nous permettre d'apprendre toutes les notions de découpages en C++, mais également nous introduire à tout un tas de notions nouvelles qui vont grandement élargir notre champs de possibilités.

III.1. Découpons du code— Les fonctions

Nos programmes deviennent de plus en plus complets, nous sommes en mesure de faire de plus en plus de choses. Mais, chaque fois que nous avons besoin de réutiliser un morceau de code, comme dans le cadre de la protection des entrées, nous nous rendons comptes que nous devons **dupliquer** notre code. Autant dire que c'est **contraignant**, ça **alourdit le code** et c'est une **source d'erreurs potentielles**.

Le moment est donc venu pour vous de découvrir **les fonctions**, à la base de nombreux codes et la solution à notre problème.

III.1.1. Les éléments de base d'une fonction

III.1.1.1. Une fonction, c'est quoi ?

Une fonction est un **ensemble d'instructions** pour réaliser une tâche précise. Cet ensemble d'instructions est isolé dans une partie spécifique qui porte un nom, appelé **identificateur**. Enfin, une fonction peut ou non prendre des informations en entrée, comme elle peut ou non fournir des informations en sortie.

Depuis le début de ce cours, nous manipulons des fonctions incluses dans la bibliothèque standard. Revenons un peu sur différents exemples qui vont illustrer la définition théorique vue ci-dessus.

III.1.1.1.1. Cas le plus simple

Nous avons vu des fonctions simples qui ne renvoyaient ni ne prenaient de valeur. Ce genre de fonction n'a pas besoin qu'on lui fournisse des données en entrées, et elle n'en fournit pas en retour.

```
1 std::string chaine { "Salut toi" };
2 // On supprime le dernier caractère.
3 chaine.pop_back();
4 // On supprime la chaîne.
5 chaine.clear();
6
7 std::list<int> liste { 4, -9, 45, 3 };
8 // On trie la liste.
9 liste.sort();
```

III. On passe la deuxième!

III.1.1.1.2. Fonction renvoyant une valeur

Certaines fonctions nous fournissent des informations que nous pouvons, au choix, récupérer ou ignorer. Le chapitre sur les tableaux nous en fournit plusieurs exemples.

```
1 std::vector<int> const tableau_de_int { 1, 2, 3, 4, 5, 6, 7, 8, 9
    };
2 // Récupérer le premier élément.
3 int premier { tableau_de_int.front() };
4 // Récupérer le dernier élément.
5 int dernier { tableau_de_int.back() };
```

Nous ne sommes pas obligés de stocker la valeur qu'une fonction renvoie. Par exemple, `std::getline` peut s'utiliser tel quel, sans se soucier de ce que la fonction a renvoyé.

III.1.1.1.3. Fonction attendant une valeur

D'autres fois, certaines fonctions ont besoin d'informations extérieures pour travailler. C'est le cas notamment des algorithmes que nous avons vus, mais aussi d'autres fonctions.

```
1 std::string texte { "Du texte." };
2 // Ajouter une lettre à la fin.
3 texte.push_back('!');
4
5 std::vector<char> tableau {};
6 // On remplit le tableau avec cinq fois la valeur 'C'.
7 tableau.assign(5, 'C');
```



Point de vocabulaire

Dans le cas où une fonction ne retourne aucune valeur, on parle dans la littérature informatique de **procédure**. Cette appellation est valable, peu importe si la fonction attend ou non des données.

III.1.1.1.4. Fusion!

Enfin, dernière possibilité, il y a les fonctions qui attendent des informations en entrée et qui en fournissent en sortie. Là encore, nous en avons vu dans le cours.

```
1 std::vector<int> tableau { 1, 2, 3, 4 };
2 // Renvoie un itérateur sur le premier élément.
3 auto it = std::begin(tableau);
```


III. On passe la deuxième!

```
4 // On compte le nombre d'éléments valant deux.
5 std::count(std::begin(tableau), std::end(tableau), 2);
6
7 std::string texte { "" };
8 // std::getline attend qu'on lui donne le flux à lire et la chaîne
  qui servira à stocker le résultat.
9 // Elle renvoie ce flux en guise de retour.
10 std::getline(std::cin, texte);
11 // Attend un conteneur en entrée et renvoie sa taille en sortie.
12 auto taille { std::size(texte) };
```

III.1.1.2. Une fonction bien connue

Nous avons un exemple de fonction sous nos yeux depuis le premier code que vous avez écrit: la fonction `main`. Celle-ci est le point d'entrée de tous les programmes que nous codons. Je la remets ci-dessous.

```
1 int main()
2 {
3     // Instructions diverses.
4     return 0;
5 }
```

- D'abord, `main` est le nom de la fonction, son **identificateur**.
- Ensuite, le `int` situé juste avant définit le **type de retour**. Dans notre cas, la fonction renvoie une valeur entière, qui est le `0` que nous voyons dans `return 0`; et qui indique au système d'exploitation que tout s'est bien passé.
- Après, nous voyons une paire de parenthèses vides, ce qui signifie que la fonction n'attend **aucune information en entrée**.
- Enfin, entre les accolades `{}`, nous avons le **corps de la fonction**, les **instructions** qui la composent.

III.1.1.3. Les composants d'une fonction

III.2. Schéma d'une fonction

En écrivant en pseudo-code, voici à quoi ressemble une fonction en C++.

```
1 type_de_retour identificateur(paramètres)
2 {
3     instructions
4 }
```

III.2.0.0.1. Votre identité, s'il vous plaît

Le point le plus important est de donner un nom à notre fonction. Les règles sont les mêmes que pour nommer nos variables. Je les réécris ci-dessous.

- L'identificateur doit **commencer par une lettre**. Il ne peut pas commencer par un chiffre, c'est interdit. Il ne doit pas commencer non plus par *underscore* `_`, car les fonctions commençant par ce caractère obéissent à des règles spéciales et sont souvent réservées à l'usage du compilateur.
- Les espaces et les signes de ponctuation sont **interdits** (`'`, `?`, etc).
- **On ne peut pas utiliser un mot-clef** du langage comme identificateur. Ainsi, il est interdit de déclarer une fonction s'appelant `int` ou `return`, par exemple.

Je vous rappelle également l'importance de donner un nom clair à vos fonctions, qui définit clairement ce qu'elles font. Évitez donc les noms trop vagues, les abréviations obscures, etc.

III.2.0.0.2. Mais de quel type es-tu ?

Comme vous le savez, une fonction peut ou non renvoyer une valeur de retour, comme nous l'avons vu à l'instant. Si elle doit renvoyer une valeur quelconque, alors il suffit d'écrire le type de cette valeur.

À l'inverse, si votre fonction ne doit rien renvoyer, alors **on utilise le mot-clef `void`**, qui signifie «vide» et qui, dans ce contexte, indique que la fonction renvoie du vide, c'est-à-dire rien.

Pour renvoyer une valeur, on utilise le mot-clef `return`, exactement comme dans la fonction `main`. Il peut y en avoir **plusieurs** dans une fonction, par exemple deux `return`, un si un `if` est vrai, l'autre s'il est faux. Par contre, lorsqu'une instruction `return` est exécutée, **on sort de la fonction en cours** et tout le code restant n'est pas exécuté. On dit qu'un retour de fonction **coupe le flot d'exécution de la fonction**. Oui, comme `break` dans une boucle. Vous suivez, c'est très bien. 🍊

III. On passe la deuxième!

III.2.0.0.3. Paramétrage en cours...

Maintenant, parlons des paramètres. Grâce aux paramètres, une fonction montre au reste du monde **ce qu'elle attend pour travailler**. Par exemple, `std::sort` attend qu'on lui donne un itérateur de début et un itérateur de fin. Quand nous ajoutons un élément à un tableau à l'aide de `push_back`, il faut indiquer à cette dernière la valeur que nous ajoutons.

Comment les déclarer? C'est très simple, il faut que chaque paramètre ait **un type** et **un identificateur**. S'il y en a plusieurs, ils seront **séparés par des virgules**.

III.2.0.0.4. Exemples

Voici un exemple de fonctions que j'ai créé en reprenant les codes que nous avons écrits il y a plusieurs chapitres.

```
1  int pgcd(int a, int b)
2  {
3      int r { a % b };
4      while (r != 0)
5      {
6          a = b;
7          b = r;
8          r = a % b;
9      }
10
11     // On peut tout à fait renvoyer la valeur d'un paramètre.
12     return b;
13 }
14
15 int somme(int n)
16 {
17     // On peut renvoyer le résultat d'un calcul ou d'une expression
18     // directement.
19     return (n * (n + 1)) / 2;
20 }
21
22 bool ou_exclusif(bool a, bool b)
23 {
24     return (a && !b) || (b && !a);
25 }
26
27 int main()
28 {
29     int const a { 845 };
30     int const b { 314 };
31
32     std::cout << "Le PGCD de " << a << " et " << b << " vaut " <<
33         pgcd(a, b) << "." << std::endl;
```

III. On passe la deuxième!

```
32     std::cout << "La somme de tous les entiers de 1 à 25 vaut " <<
      somme(25) << "." << std::endl;
33
34     std::cout << std::boolalpha;
35     std::cout << "XOR(true, false) vaut " << ou_exclusif(true,
      false) << "." << std::endl;
36
37     return 0;
38 }
```

Notez qu'on peut très bien donner en argument à une fonction une variable qui porte le même identificateur qu'un des paramètres. Il n'y a aucun problème à ça, car nous sommes dans **deux portées différentes**: `main` et la fonction en question.

Conséquence du point précédent, il n'est **pas possible d'utiliser un paramètre de fonction en dehors de celle-ci**. Le code suivant produit donc une erreur.

```
1 void fonction(int parametre)
2 {
3     // Aucun problème.
4     parametre = 5;
5 }
6
7 int main()
8 {
9     // La variable parametre n'existe pas !
10    parametre = 410;
11    return 0;
12 }
```



Argument et paramètre

Chez beaucoup de personnes, la distinction entre paramètre et argument n'est pas claire.

- Un paramètre, aussi appelé paramètre formel, c'est **ce qu'attend une fonction pour travailler** et qui est inscrit dans sa déclaration.
- Un argument, aussi appelé paramètre réel, c'est **la valeur transmise à la fonction quand on l'utilise**.



```
1 void fonction(int entier)
2 {
3     // entier est le paramètre de la fonction.
4 }
5
6 int main()
7 {
8     // 4 est l'argument de la fonction.
9     fonction(4);
10
11     int const valeur { 5 };
12     // valeur est l'argument de la fonction.
13     fonction(valeur);
14     return 0;
15 }
```

En pratique, beaucoup utilisent les deux de façon interchangeable et vous serez parfaitement compris si vous faites de même. Je tenais néanmoins à vous expliquer cela rigoureusement.

III.2.1. Exercices

Il est temps de pratiquer toutes ces nouvelles notions que vous avez acquises, afin de mieux les comprendre et bien les retenir.

III.2.1.1. Afficher un rectangle

Commençons par une fonction simple qui **affichera un rectangle** composé d'étoiles *. Voici un exemple en lui donnant une longueur de 4 et une largeur de 3.

```
1 ***
2 ***
3 ***
4 ***
```

👁 Correction affichage de rectangles

III. On passe la deuxième!

III.2.1.2. Distributeur d'argent

Le but est de **donner une quantité de coupures correspondant à une somme donnée**, en utilisant les plus grosses en premiers. Par exemple, 285€ avec des coupures de 500€, 200€, 100€, 50€, 20€, 10€, 5€, 2€ et 1€ donne un total d'un billet de 200€, un billet de 50€, un billet de 20€, un billet de 10€ et enfin, un billet de 5€.

Par contre, 346€ avec uniquement 20€, 10€, 5€, 2€ et 1€ donne dix-sept billets de 20€, un billet de 5€ et une pièce de 1€. Il faut donc prendre en compte les coupures disponibles dans le résultat.

👁 Correction distributeur d'argent

Bonus: faire une fonction qui affiche de façon lisible le total. Par exemple, pour 285€, on peut imaginer la sortie suivante.

1	1 x 200 euros
2	1 x 50 euros
3	1 x 20 euros
4	1 x 10 euros
5	1 x 5 euros

👁 Correction bonus

III.2.1.3. Parenthésage

Imaginons une expression de plusieurs dizaines de caractères, avec des parenthèses ouvrantes (et fermantes). Il serait fastidieux de regarder manuellement si une expression est correctement parenthésée, c'est-à-dire qu'elle a le **même nombre de symboles ouvrants et fermants** et qu'ils sont bien **dans l'ordre**.

Ainsi, l'expression `((()))` est correcte, alors que `((())` est incorrecte parce qu'il manque une parenthèse fermante. Quant à `) (`, elle est également incorrecte parce que le symbole fermant est avant le symbole ouvrant.

👁 Correction parenthésage

III.2.2. Quelles sont vos références ?

III.2.2.1. La problématique

J'ai écrit dans la première section que les paramètres d'une fonction étaient dans une portée différente, ce qui autorise de faire ceci.

```
1 void fonction(int a)
2 {
3     a = 5;
4 }
5
6 int main()
7 {
8     int a { 8 };
9     // Aucun problème.
10    fonction(a);
11
12    return 0;
13 }
```

Il faut cependant savoir que ceci implique **une copie**. Pour chaque paramètre, le compilateur va réserver un espace en mémoire de la taille nécessaire et copier la variable qu'on lui passe. Nous allons donc manier des variables totalement différentes. C'est ce qui s'appelle le **passage par copie**.

```
1 #include <iostream>
2 #include <vector>
3
4 void fonction(int a, double b, std::vector<int> c)
5 {
6     a = 5;
7     b = 2.18781;
8     c.assign(7, 0);
9
10    // Ici, nous manipulons donc des copies.
11    std::cout << a << std::endl;
12    std::cout << b << std::endl;
13    for (auto element : c)
14    {
15        std::cout << element << " ";
16    }
17    std::cout << std::endl << std::endl;
18 }
19
20 int main()
```

III. On passe la deuxième!

```
21 {
22     int const entier { 42 };
23     double const reel { 3.14159 };
24     std::vector<int> const tableau { 1, 782, 4, 5, 71 };
25
26     // Chaque variable va être copiée.
27     fonction(entier, reel, tableau);
28
29     // Affichons pour prouver que les variables originales n'ont
30     pas changé.
31     std::cout << entier << std::endl;
32     std::cout << reel << std::endl;
33     for (auto element : tableau)
34     {
35         std::cout << element << " ";
36     }
37
38     std::cout << std::endl;
39     return 0;
40 }
```

Le côté négatif, c'est qu'on doit forcément copier l'argument, même si on ne le modifie pas. Dans le cas d'un entier ou d'un caractère, c'est idéal, mais dans le cas d'un tableau de plusieurs milliers d'éléments, on perd du temps inutilement. C'est comme si je demandais à consulter un certain original dans des archives et qu'on me faisait des photocopies à la place. S'il s'agit d'un post-it, pas grave, mais s'il s'agit d'un livre de 400 pages, bonjour le gâchis!

III.2.2.2. Les références

La solution consiste à ne pas passer une copie de la variable originale mais un **alias**, une **référence** vers celle-ci et qui soit manipulable comme si j'avais l'original entre les mains. Cela revient à me donner le document original et éventuellement, selon le contexte, le droit de le modifier.

Une référence en C++ est simple à déclarer mais doit réunir plusieurs conditions.

- On utilise l'esperluette & pour ça.
- Elle doit être **de même type que la variable cible**. Une référence sur un **double** ne peut pas se voir associée à une variable de type **int**.
- Une référence **cible une seule et unique variable**. On ne peut pas créer de référence qui ne cible rien, ni changer la cible d'une référence une fois qu'on l'a créée.
- Une référence peut être **définie comme constante**, ce qui implique qu'on ne pourra pas modifier sa valeur.

```
1 #include <iostream>
2
3 int main()
```


III. On passe la deuxième!

```
4 {
5     int entier { 40 };
6     // On initialise notre référence pour qu'elle référence entier.
7     int & reference_entier { entier };
8
9     std::cout << "Entier vaut : " << entier << std::endl;
10    std::cout << "Référence vaut : " << reference_entier <<
        std::endl;
11
12    reference_entier += 2;
13    std::cout << "Entier vaut : " << entier << std::endl;
14    std::cout << "Référence vaut : " << reference_entier <<
        std::endl;
15
16    int const entier_constant { 0 };
17    // On peut également déclarer des références constantes sur des
        entiers constants.
18    int const & reference_const_entier_constant { entier_constant
        };
19    // Ce qui interdit ce qui suit, car le type référencé est un
        int const.
20    // reference_const_entier_constant = 1;
21
22    // On ne peut pas déclarer de référence non constante sur un
        entier constant.
23    //int & reference_entier_constant { entier_constant };
24
25    // On peut par contre tout à fait déclarer une référence
        constante sur un objet non constant.
26    // On ne peut donc pas modifier la valeur de l'objet en
        utilisant la référence, mais l'entier
27    // original reste tout à fait modifiable.
28    int const & reference_const_entier { entier };
29    std::cout << "Entier vaut : " << entier << std::endl;
30    std::cout << "Référence constante vaut : " <<
        reference_const_entier << std::endl;
31
32    entier = 8;
33    std::cout << "Entier vaut : " << entier << std::endl;
34    std::cout << "Référence constante vaut : " <<
        reference_const_entier << std::endl;
35
36    return 0;
37 }
```



Tableaux et références

Vous avez déjà utilisé des références sans le savoir. En effet, quand, dans le chapitre précédent, nous accédons à un élément d'un tableau, c'est en fait une référence qui est renvoyée, ce qui autorise à modifier un tableau comme nous l'avions vu.

```
1 // On change la valeur du premier élément.
2 tableau[0] = 5;
```

C++ est un langage qui cache plein de concepts sous des airs innocents. 🍊

III.2.2.3. Paramètres de fonctions

On peut tout à fait déclarer les paramètres d'une fonction comme étant des références. Ainsi, celles-ci ne recevront plus une copie mais bel et bien des références sur les objets. On parle donc de **passage par référence**. On peut, bien entendu, préciser si la référence est constante ou non. Dans le premier cas, toute modification du paramètre sera interdite, alors que le deuxième cas autorise à le modifier et conserver ces changements.

```
1 #include <iostream>
2 #include <string>
3
4 void fonction_reference_const(std::string const & texte)
5 {
6     std::cout <<
7         "J'ai reçu une référence comme paramètre. Pas de copie, youpi !"
8         << std::endl;
9     // Interdit.
10    // texte = "Changement";
11 }
12
13 void fonction_reference(std::string & modifiable)
14 {
15     std::cout <<
16         "Je peux modifier la chaîne de caractères et les changements seront co
17         << std::endl;
18     modifiable = "Exemple";
19 }
20
21 int main()
22 {
23     std::string modifiable { "Du texte modifiable." };
24     std::string const non_modifiable {
25         "Du texte qui ne sera pas modifiable." };
26 }
```

III. On passe la deuxième!

```
22     std::cout << "Avant : " << modifiable << std::endl;
23     fonction_reference(modifiable);
24     std::cout << "Après : " << modifiable << std::endl;
25
26     fonction_reference_const(non_modifiable);
27     // Possible.
28     fonction_reference_const(modifiable);
29     // Impossible, parce que la fonction attend une référence sur
        un std::string et nous lui donnons une référence sur un
        std::string const.
30     //fonction_reference(non_modifiable);
31
32     return 0;
33 }
```



Quand dois-je utiliser une référence constante ou une référence simple?

Il suffit de se poser la question: «est-ce que ce paramètre est destiné à être modifié ou non?» Si c'est le cas, alors on utilisera une référence simple. Dans le cas contraire, il faut utiliser une référence constante. Ainsi, l'utilisateur de la fonction aura des garanties que son objet ne sera pas modifié dans son dos.

III.2.2.3.1. Le cas des littéraux

Analysez le code suivant, sans le lancer, et dites-moi ce que vous pensez qu'il va faire. Si vous me dites qu'il va simplement afficher deux messages... dommage, vous avez perdu. 🍊

```
1  #include <iostream>
2
3  void test(int & a)
4  {
5      std::cout << "Je suis dans la fonction test.\n";
6  }
7
8  void test_const(int const & a)
9  {
10     std::cout << "Je suis dans la fonction test_const.\n";
11 }
12
13 int main()
14 {
15     test(42);
16     test_const(42);
17     return 0;
18 }
```

III. On passe la deuxième!

En fait, ce code ne va même pas compiler, à cause de la ligne 15. Il y a une très bonne raison à ça: nous passons un littéral, qui est **invariable** et **n'existe nulle part en mémoire**, or nous tentons de le donner en argument à une fonction qui attend une référence, un alias, sur une variable modifiable. Mais un littéral n'est pas modifiable et tenter de le faire **n'a aucun sens**. Le compilateur refuse donc et ce code est invalide.

Par contre, supprimez-la et le code compilera. Pourtant, la seule différence entre les deux fonctions, c'est la présence d'un **const**. *Eh bien c'est lui qui rend le code valide*. En effet, le compilateur va **créer un objet temporaire anonyme**, qui n'existera que le temps que la fonction prendra pour travailler, et va lui assigner le littéral comme valeur. On peut réécrire le code précédent comme ceci pour mieux comprendre.

```
1 #include <iostream>
2 #include <string>
3
4 void test_const(int const & a)
5 {
6     std::cout << "Je suis dans la fonction test_const.\n";
7 }
8
9 void test_string_const(std::string const & texte)
10 {
11     std::cout << "Je suis dans la fonction test_string_const.\n";
12 }
13
14 int main()
15 {
16     // test_const(42);
17     {
18         // Nouvelle portée.
19         int const variable_temporaire { 42 };
20         int const & reference_temporaire { variable_temporaire };
21         test_const(reference_temporaire);
22         // Nos variables disparaissent.
23     }
24
25     // Exemple avec une chaîne de caractères.
26     // test_string_const("Salut toi !");
27     {
28         std::string const texte_temporaire { "Salut toi !" };
29         std::string const & reference_temporaire { texte_temporaire
30             };
31         test_string_const(reference_temporaire);
32     }
33     return 0;
34 }
```

III. On passe la deuxième!

III.2.2.3.2. Le cas des types standard

Utiliser des références pour `std::vector` ou `std::string` d'accord, mais qu'en est-il des types simples comme `int`, `char` et `double`? Y a-t-il un intérêt à utiliser des références constantes? Doit-on bannir toutes les copies de nos codes sous prétexte d'efficacité? Non.

L'usage en C++, c'est que les types natifs sont **toujours passés par copie**. Pourquoi cela? Parce qu'ils sont très petits et le coût de la création d'une référence sur des types aussi simples est souvent plus élevé que celui d'une bête copie. En effet, le compilateur peut **optimiser la copie** et la rendre extrêmement rapide, bien plus qu'avec les références.

Donc ne tombez pas dans le piège de **l'optimisation prématurée**, qui consiste à vouloir améliorer les performances de son programme en faisant des changements qui n'apporteront aucune amélioration. Laissez plutôt ce travail au compilateur. 🍊

III.2.2.4. Valeur de retour

Les références ont une contrainte particulière: **elles doivent toujours être valides**. Dans le cas d'un paramètre de fonction, il n'y a aucun danger. En effet, l'argument transmis à la fonction existera toujours quand celle-ci se terminera, comme le montre le code suivant.

```
1 void fonction(int & reference)
2 {
3     // On modifie une référence sur une variable qui existe, pas de
4     // soucis.
5     reference = 42;
6 }
7 int main()
8 {
9     int variable { 24 };
10    // On transmet variable...
11    fonction(variable);
12    // ... qui existe toujours après la fin de la fonction.
13
14    return 0;
15 }
```

Cependant, dans le cas où l'on souhaite qu'une fonction renvoie une référence, il faut faire très attention parce qu'on peut **renvoyer une référence sur un objet qui n'existe plus!** Ceci va donc entraîner un **comportement indéterminé**.

```
1 int& fonction()
2 {
3     int variable { 42 };
```

III. On passe la deuxième!

```
4      // Oho, variable sera détruite quand la fonction sera terminée
5      !
6      return variable;
7  }
8  int main()
9  {
10     // Grosse erreur, notre référence est invalide !
11     int& reference { fonction() };
12
13     return 0;
14 }
```



Référence locale

Ne renvoyez JAMAIS de référence sur une variable locale.

III.2.2.5. Un mot sur la déduction de type

Certains reprochent son côté «rigide» à ce mot-clé. En effet, `auto` seul dégage la référence. À l'inverse, `auto &` sera toujours une référence. Et il en est de même pour `const`: `auto const` sera toujours constant et `auto const &` sera toujours une référence sur un objet constant.

```
1  int main()
2  {
3      int variable { 2 };
4      int & reference { variable };
5
6      // Sera toujours une variable et jamais une référence.
7      auto toujours_variable_1 { variable };
8      auto toujours_variable_2 { reference };
9
10     // Sera toujours une référence et jamais une variable.
11     auto & toujours_reference_1 { variable };
12     auto & toujours_reference_2 { reference };
13
14     // On pourrait rajouter des exemples avec const, mais vous
15     // comprenez le principe.
16     return 0;
17 }
```

Il existe cependant un mot-clé qui s'adapte toujours au type en paramètre parce qu'il conserve la présence (ou non) de `const` et de références. Il s'appelle `decltype` et s'utilise de deux façons. La première, c'est avec une **expression explicite**, comme un calcul, une autre variable ou le

III. On passe la deuxième!

retour d'une fonction, ce qui donne `decltype(expression) identifiant { valeur };` et est illustré ci-dessous.

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> tableau { -8, 45, 35, 9 };
7
8     // Sera toujours une variable.
9     auto premier_element_variable { tableau[0] };
10    // Est en fait une référence sur cet élément, ce qui autorise à
11    // le modifier.
12    decltype(tableau[1]) deuxieme_element_reference { tableau[1] };
13
14    // Modifions et regardons.
15    premier_element_variable = 89;
16    deuxieme_element_reference = 0;
17
18    for (auto const & element : tableau)
19    {
20        std::cout << element << std::endl;
21    }
22
23    return 0;
24 }
```

La deuxième façon de l'utiliser, c'est de combiner `decltype` et `auto`, donnant `decltype(auto) identifiant { valeur };` afin d'avoir le type exact de `valeur` et de ne pas se répéter. La ligne 11 de notre code précédent peut ainsi être raccourcie.

```
1 std::vector<int> tableau { -8, 45, 35, 9 };
2 // Sera de type int& si tableau est modifiable, de type int const &
3 // si tableau est déclaré comme const.
4 decltype(auto) deuxieme_element_reference { tableau[1] };
```

Ces deux mots-clefs ne sont pas deux concurrents s'opposant pour faire le même travail, bien au contraire. Ils sont une preuve de la liberté que C++ laisse au développeur. Nous verrons plus tard dans ce cours d'autres usages, qui rendront ces mots-clefs encore plus concrets pour vous. En attendant, reprenez ceci pour vous aider à choisir.

- Choisissez `auto` lorsque vous souhaitez explicitement une valeur (par copie).
- Choisissez `auto &` (`const` ou non) lorsque vous souhaitez explicitement une référence.
- Choisissez `decltype` lorsque vous souhaitez le type exact.

III.2.3. Nos fonctions sont surchargées !

Dans le chapitre sur les fichiers, je vous ai appris qu'il y a, en plus de `std::ostringstream` et `std::istringstream`, d'autres méthodes pour faire la conversion nombre <-> chaîne de caractères.

Voici, ci-dessous, comment cette fonction est déclarée dans la bibliothèque standard.

```
1 std::string to_string(int value);
2 std::string to_string(double value);
```

?

Attends une seconde! Comment c'est possible qu'on ait deux fonctions avec le même nom?



La réponse est simple: ce qui différencie deux fonctions, ça n'est pas seulement leur identificateur mais également leurs paramètres. C'est ce qu'on appelle la **signature d'une fonction**. Dans notre cas, le compilateur voit deux signatures différentes: `to_string(int)` et `to_string(double)`, sans que ça ne pose le moindre problème.

Cette possibilité s'appelle **surcharge**, de l'anglais «*overloading*». Elle rend le code plus clair car on laisse au compilateur le soin d'appeler la bonne surcharge.

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     int const entier { -5 };
7     double const flottant { 3.14156 };
8
9     std::string affichage { "Voici des nombres : " };
10    affichage += std::to_string(entier) + ", ";
11    affichage += std::to_string(flottant);
12
13    std::cout << affichage << std::endl;
14
15    return 0;
16 }
```

Par contre, le **type de retour n'entre pas en compte dans la signature** d'une fonction. Ainsi, deux fonctions avec le même identifiant, les mêmes paramètres et des types de retour différents seront rejetées par le compilateur car il ne saura pas laquelle utiliser. Et c'est logique, car comment, en lisant le code, sauriez-vous ce que la fonction va vous renvoyer?

III. On passe la deuxième!

C'est pour ça que les fonctions convertissant des chaînes de caractères en nombres ont des noms bien distincts: `std::stoi` pour convertir une chaîne en `int` et `std::stod` pour `double`.

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     int const entier { std::stoi("-8") };
7     double const flottant { std::stod("2.71878") };
8
9     std::cout << entier + flottant << std::endl;
10    return 0;
11 }
```

III.2.4. [T.P] Gérer les erreurs d'entrée— Partie III

Avec ce que nous avons vu, nous sommes maintenant capable de faire des fonctions qui vont nous épargner d'avoir à dupliquer plusieurs fois le même code chaque fois que nous souhaiterons sécuriser nos entrées. Nous pourrions ainsi sécuriser le code suivant sans duplication.

```
1 #include <iostream>
2
3 int main()
4 {
5     int jour { 0 };
6     std::cout << "Quel jour es-tu né ? ";
7     // Entrée.
8
9     int mois { 0 };
10    std::cout << "Quel mois ? ";
11    // Entrée.
12
13    int annee { 0 };
14    std::cout << "Quelle année ? ";
15    // Entrée.
16
17    double taille { 0.0 };
18    std::cout << "Quelle taille ? ";
19    // Entree.
20
21    std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
        annee << " et tu mesures " << taille << "m." << std::endl;
22    return 0;
23 }
```

III.2.5. Dessine-moi une fonction

Nous allons nous attarder sur un petit problème qui peut apparaître lorsque l'on crée des fonctions: les appels croisés.

III.2.5.1. Le problème

Introduisons ce problème par un exemple: essayons d'implémenter deux fonctions `est_pair` et `est_impair`, dont l'objectif est, comme leurs noms l'indiquent, de dire si un entier est pair ou impair. Une manière élégante de procéder est de faire en sorte que chacune appelle l'autre.

```
1 bool est_pair(int i)
2 {
3     if (i == 0)
4     {
5         return true;
6     }
7     else if (i < 0)
8     {
9         return est_pair(-i);
10    }
11    else
12    {
13        return est_impair(i - 1);
14    }
15 }
16
17 bool est_impair(int i)
18 {
19     if (i < 0)
20     {
21         return est_impair(-i);
22     }
23     else
24     {
25         return est_pair(i - 1);
26     }
27 }
```

L'idée est de s'approcher pas à pas du cas 0 (dont on connaît la parité) en appelant la fonction avec l'entier `i-1`. C'est ce qu'on appelle de la programmation récursive, rappelant l'idée d'une récurrence en mathématiques.

III. On passe la deuxième!

Bref, ce qui est important ici, c'est que ce code ne compile pas. En effet, comme le compilateur lit le code de haut en bas, il ne connaît pas la fonction `est_impair` lorsqu'on l'utilise dans `est_pair`. Et ça ne marche pas non plus en échangeant les deux fonctions, puisque `est_impair` appelle aussi `est_pair`.

Heureusement, nos fonctions ne sont pas à jeter puisque C++ est bien fait, et qu'il a donc une solution à ce problème: les **prototypes**.

III.2.5.2. La solution

Un prototype est une expression qui décrit toutes les informations qui définissent une fonction, à part son implémentation.

- Son identificateur.
- Son type de retour.
- Ses paramètres.

On peut résumer en disant que c'est tout simplement la première ligne de la définition d'une fonction, à laquelle on ajoute un point-virgule.

```
1 type_de_retour identifiant(paramètres);
```

Ces informations sont suffisantes pour que le compilateur sache que la fonction existe et qu'il sache comment elle s'utilise. Cela nous permet alors d'écrire, pour notre exemple précédent:

```
1 bool est_impair(int i);
2
3 bool est_pair(int i)
4 {
5     if (i == 0)
6     {
7         return true;
8     }
9     else if (i < 0)
10    {
11        return est_pair(-i);
12    }
13    else
14    {
15        return est_impair(i - 1);
16    }
17 }
18
19 bool est_impair(int i)
20 {
21     if (i < 0)
22     {
```

III. On passe la deuxième!

```
23         return est_impair(-i);
24     }
25     else
26     {
27         return est_pair(i - 1);
28     }
29 }
```

Et maintenant, ça compile!

III.2.5.3. Utilisons les prototypes

Les prototypes ne servent pas qu'à résoudre les problèmes d'appels croisés. En effet, ils peuvent être utilisés pour améliorer la lisibilité du code. Tout comme le compilateur, **nous n'avons pas besoin de l'implémentation pour savoir comment s'utilise une fonction**. En plus, le nom nous suffit souvent, lorsqu'il est bien choisi, pour savoir **à quoi sert la fonction**.

Constat: **nous ne voulons pas voir l'implémentation**. Derrière cette phrase un tantinet abusive, je veux dire que l'essentiel de l'information est réunie dans le prototype.

On peut donc tirer parti de cela pour rendre nos programmes plus lisibles. Par exemple, on peut mettre tous les prototypes avant la fonction `main` et reléguer les implémentations à la fin du fichier, juste après cette dernière. Ainsi, lorsque l'on veut utiliser une fonction, le prototype n'est plus noyé dans les implémentations et on peut en un coup d'œil récupérer toutes les informations qui nous intéressent.

Cette idée de **faire valoir les interfaces plutôt que les implémentations** est très importante en programmation et l'on aura l'occasion de la recroiser plusieurs fois.

III.2.5.4. En résumé

- Une fonction permet de factoriser un morceau de code utilisé régulièrement. Nous en utilisons beaucoup depuis le début de ce cours.
- Une fonction a un identifiant et peut prendre ou non plusieurs paramètres, ainsi que renvoyer ou non une valeur. Dans le cas où elle ne renvoie rien, elle est déclarée avec le mot-clef `void`.
- Les références sont des alias de variables qui permettent, à travers un autre nom, de les manipuler directement. On s'en sert, notamment, dans les paramètres des fonctions pour éviter des copies inutiles ou pour conserver les modifications faites sur un objet.
- Il faut faire attention à ce qu'une référence soit toujours valide. Cela nous interdit donc de renvoyer une référence sur une variable locale.
- Le mot-clef `auto` ne conserve pas les références, mais le nouveau mot-clef que nous avons introduit, `decltype`, si.
- La signature d'une fonction est ce qui la rend unique et est composée de son identifiant et de ses paramètres. On peut donc avoir plusieurs fonctions avec le même identifiant si les paramètres sont différents, ce qu'on appelle la surcharge.

III. On passe la deuxième!

- Le prototype d'une fonction permet, entre autres, de résoudre le problème des appels croisés.
- Le prototype contenant l'identificateur, le type de retour et les paramètres d'une fonction, il donne suffisamment d'informations pour ne pas avoir besoin de lire l'implémentation.

Contenu masqué

Contenu masqué n°30 :

Correction affichage de rectangles

```
1 #include <iostream>
2
3 void rectangle(int longueur, int largeur)
4 {
5     for (auto i { 0 }; i < longueur; ++i)
6     {
7         for (auto j { 0 }; j < largeur; ++j)
8         {
9             std::cout << "*";
10        }
11
12        std::cout << std::endl;
13    }
14 }
15
16 int main()
17 {
18     rectangle(2, 5);
19     rectangle(4, 3);
20     return 0;
21 }
```

[Retourner au texte.](#)

Contenu masqué n°31 :

Correction distributeur d'argent

```
1 #include <iostream>
2 #include <vector>
3
4 std::vector<int> distributeur(int total, std::vector<int>
    coupures_disponibles)
```

III. On passe la deuxième!

```
5 {
6     std::vector<int> resultat {};
7
8     for (auto coupure : coupures_disponibles)
9     {
10         resultat.push_back(total / coupure);
11         total %= coupure;
12     }
13
14     return resultat;
15 }
16
17 int main()
18 {
19     std::vector<int> const coupures_disponibles { 500, 200, 100,
20         50, 20, 10, 5, 2, 1 };
21     auto const coupures { distributeur(285, coupures_disponibles)
22         };
23
24     for (auto coupure : coupures)
25     {
26         std::cout << coupure << std::endl;
27     }
28
29     return 0;
30 }
```

[Retourner au texte.](#)

Contenu masqué n°32 : Correction bonus

```
1 #include <iostream>
2 #include <vector>
3
4 std::vector<int> distributeur(int total, std::vector<int>
5     coupures_disponibles)
6 {
7     std::vector<int> resultat {};
8
9     for (auto coupure : coupures_disponibles)
10    {
11        resultat.push_back(total / coupure);
12        total %= coupure;
13    }
```

III. On passe la deuxième!

```
14     return resultat;
15 }
16
17 void affichage_distributeur(std::vector<int> total,
18     std::vector<int> coupures_disponibles)
19 {
20     int i { 0 };
21     for (auto coupure : total)
22     {
23         // Si 0, ça veut dire qu'il n'y a aucune coupure de ce type
24         // dans le total, donc nous n'afficherons rien.
25         if (coupure != 0)
26         {
27             std::cout << coupure << " x " <<
28                 coupures_disponibles[i] << " euros" << std::endl;
29         }
30         ++i;
31     }
32 }
33
34 int main()
35 {
36     std::vector<int> const coupures_disponibles { 500, 200, 100,
37         50, 20, 10, 5, 2, 1 };
38     auto const coupures_285 { distributeur(285,
39         coupures_disponibles) };
40     affichage_distributeur(coupures_285, coupures_disponibles);
41
42     // Moins de coupures disponibles.
43     std::vector<int> const autre_coupures_disponibles { 200, 50,
44         10, 1 };
45     auto const coupures_45874 { distributeur(45874,
46         autre_coupures_disponibles) };
47     affichage_distributeur(coupures_45874,
48         autre_coupures_disponibles);
49
50     return 0;
51 }
```

[Retourner au texte.](#)

Contenu masqué n°33 :

Correction parenthésage

```
1 #include <iostream>
2
3 bool parentheses(std::string expression)
4 {
5     int ouvrantes { 0 };
6     int fermantes { 0 };
7
8     for (auto caractere : expression)
9     {
10         if (caractere == '(')
11         {
12             ++ouvrantes;
13         }
14         else
15         {
16             ++fermantes;
17         }
18
19         if (fermantes > ouvrantes)
20         {
21             // Pas la peine de continuer, l'expression est
22             // invalide.
23             return false;
24         }
25     }
26
27     return ouvrantes == fermantes;
28 }
29
30 int main()
31 {
32     std::cout << std::boolalpha;
33     std::cout << parentheses("((()))") << std::endl;
34     std::cout << parentheses("(()())") << std::endl;
35     std::cout << parentheses(")(") << std::endl;
36
37     return 0;
38 }
```

[Retourner au texte.](#)

Contenu masqué n°34 :

Correction T.P Partie III

```
1 #include <iostream>
2 #include <string>
3
4 void entree_securisee(int & variable)
5 {
6     while (!(std::cin >> variable))
7     {
8         std::cout << "Entrée invalide. Recommence." << std::endl;
9         std::cin.clear();
10        std::cin.ignore(255, '\n');
11    }
12 }
13
14 void entree_securisee(double & variable)
15 {
16     while (!(std::cin >> variable))
17     {
18         std::cout << "Entrée invalide. Recommence." << std::endl;
19         std::cin.clear();
20         std::cin.ignore(255, '\n');
21     }
22 }
23
24 int main()
25 {
26     int jour { 0 };
27     std::cout << "Quel jour es-tu né ? ";
28     entree_securisee(jour);
29
30     int mois { 0 };
31     std::cout << "Quel mois ? ";
32     entree_securisee(mois);
33
34     int annee { 0 };
35     std::cout << "Quelle année ? ";
36     entree_securisee(annee);
37
38     double taille { 0.0 };
39     std::cout << "Quelle taille ? ";
40     entree_securisee(taille);
41
42     std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
        annee << " et tu mesures " << taille << "m." << std::endl;
43     return 0;
44 }
```

On remarque quand même que si le code n'est plus répété à chaque entrée, **nos fonctions**

III. On passe la deuxième!

sont quasiment identiques, la seule différence résidant dans le paramètre qu'elles attendent. Finalement, on duplique toujours le code. Mais comme vous vous en doutez, il y a une solution pour ça, que nous verrons en temps voulu. [Retourner au texte.](#)

III.3. Erreur, erreur, erreur...

Depuis le début de ce cours, nous avons été épargnés par les erreurs. Nous avons certes appris à sécuriser un peu les entrées, mais c'est tout. Ça ne sera pas toujours le cas. Il serait de bon ton que nos programmes soient **résistants aux erreurs**, afin de les rendre fiables et robustes.

Le but de ce chapitre va être d'introduire la **gestion des erreurs** ainsi que la **réflexion à mener** lors de l'écriture de son code.

III.3.1. L'utilisateur est un idiot

Derrière ce titre un poil provoquant se cache une vérité informatique universelle: peu importe que ce soit consciemment ou non, il arrivera **forcément** que l'utilisateur rentre une information incomplète, fausse, ou totalement différente de ce qu'on attendait. Que ce soit parce qu'il veut s'amuser avec votre programme ou bien parce qu'il a mal compris une consigne, a oublié un caractère ou s'est trompé de ligne, votre programme doit être **capable de gérer ça**.

Vérifier les entrées veut bien sûr dire s'assurer qu'elles soient du bon type, mais aussi contrôler que la donnée est cohérente et valide. Ainsi, vous ne pouvez pas dire que vous êtes né le 38 avril, car un mois a 31 jours au maximum. Si la donnée n'est pas correcte, même si elle est du bon type, il faut demander à l'utilisateur de la ressaisir. Cela se fait relativement facilement.

```
1  #include <iostream>
2
3  int main()
4  {
5      int jour { 0 };
6      std::cout << "Donne un jour entre 1 et 31 : ";
7
8      while (!(std::cin >> jour) || jour < 0 || jour > 31)
9      {
10         std::cout << "Entrée invalide. Recommence." << std::endl;
11         std::cin.clear();
12         std::cin.ignore(255, '\n');
13     }
14
15     std::cout << "Jour donné : " << jour << std::endl;
16     return 0;
17 }
```

III. On passe la deuxième!

Tant que l'entrée met `std::cin` dans un état invalide ou qu'elle n'est pas dans la plage de valeurs autorisées, on demande à l'utilisateur de saisir une nouvelle donnée. Vous reconnaîtrez que ce code est tiré de notre fonction `entree_securisee`. Je l'ai extraite pour pouvoir ajouter des conditions supplémentaires. Nous verrons dans le prochain chapitre une façon d'améliorer ça.

Pour l'instant, reprenez qu'il faut **vérifier qu'une donnée est cohérente** en plus de s'assurer qu'elle est du bon type.

III.3.2. À une condition... ou plusieurs

Écrire une fonction peut être plus ou moins simple, en fonction de ce qu'elle fait, mais beaucoup de développeurs oublient une étape essentielle consistant à **réfléchir au(x) contrat(s) que définit une fonction**. Un contrat est un accord entre deux parties, ici celui qui utilise la fonction et celui qui la code.

- La fonction mathématique `std::sqrt`, qui calcule la racine carrée d'un nombre, attend qu'on lui donne un **réel nul ou positif**. Si ce contrat est respecté, elle s'engage à retourner un **réel supérieur ou égal à zéro** et que **celui-ci au carré soit égal à la valeur donnée en argument**. Le contrat inhérent à la fonction `std::sqrt` se résume donc ainsi: «donne-moi un entier positif, et je te renverrai sa racine carrée». Ça paraît évident, mais c'est important de le garder en tête.
- Quand on récupère un élément d'un tableau avec `[]`, il faut fournir un **index valide**, compris entre 0 et la taille du tableau moins un. Si on fait ça, on a la garantie d'obtenir un **élément valide**.
- Si l'on a une chaîne de caractères ayant **au moins un caractère**, on respecte le contrat de `pop_back` et on peut **retirer en toute sécurité le dernier caractère**.

On voit, dans chacun de ces exemples, que la fonction attend qu'on respecte une ou plusieurs conditions, que l'on nomme les **préconditions**. Si celles-ci sont respectées, la fonction s'engage en retour à respecter sa part du marché, qu'on appelle les **postconditions**. Les passionnés de mathématiques feront un parallèle avec les **domaines de définitions** des fonctions.

Le but de réfléchir à ces conditions est de produire du code plus robuste, plus fiable et plus facilement testable, pour toujours tendre vers une plus grande qualité de notre code. En effet, en prenant quelques minutes pour penser à ce qu'on veut que la fonction fasse ou ne fasse pas, on sait plus précisément quand **elle fonctionne** et quand **il y a un bug**.

III.3.2.1. Contrats assurés par le compilateur

III.4. Le typage

Un exemple simple de contrat, que nous avons déjà vu sans même le savoir, s'exprime avec le typage. Si une fonction s'attend à recevoir un entier, on ne peut pas passer un `std::vector`. Le compilateur s'assure qu'on passe en arguments des types compatibles avec ce qui est attendu.

III.4.0.0.1. Avec `const`

Un autre type de contrat qu'on a déjà vu implique l'utilisation de `const`. Ainsi, une fonction qui attend une référence sur une chaîne de caractères constante garantie que celle-ci restera intacte, **sans aucune modification**, à la fin de la fonction. Cette postcondition est vérifiée par le compilateur, puisqu'il est impossible de modifier un objet déclaré comme étant `const`.

III.4.0.1. Vérifier nous-mêmes

Le compilateur fait une partie du travail certes, mais la plus grosse part nous revient. Pour vérifier nos contrats, nous avons dans notre besace plusieurs outils que nous allons voir.

III.4.1. Le développeur est un idiot

Vous souvenez vous du chapitre sur les tableaux? Quand nous avons vu l'accès aux éléments avec les crochets, je vous ai dit de vérifier que l'indice que vous utilisez est bien valide, sous peine de comportements indéterminés. Si vous ne le faites pas, alors c'est de votre faute si le programme plante, car vous n'avez pas respectés une précondition. C'est ce qu'on appelle une **erreur de programmation**, ou *bug*.

Dans le cas où ce genre de problème arrive, rien ne sert de continuer, mieux vaut arrêter le programme et corriger le code. Il existe justement un outil qui va nous y aider: **les assertions**.



Fichier à inclure

Pour utiliser les assertions, il faut inclure le fichier d'en-tête `<cassert>`.

Une assertion fonctionne très simplement. Il s'agit d'évaluer une condition quelconque. Si la condition est vraie, le programme continue normalement. Par contre, si elle se révèle être fausse, **le programme s'arrête brutalement**. Voyez par vous-mêmes ce qu'il se passe avec le code suivant.

III. On passe la deuxième!

```
1 #include <cassert>
2
3 int main()
4 {
5     // Va parfaitement fonctionner et passer à la suite.
6     assert(1 == 1);
7     // Va faire planter le programme.
8     assert(1 == 2);
9     return 0;
10 }
```

```
1 Assertion failed: 1 == 2, file d:\documents\visual studio
  2017\projects\zdcpp\zdcpp\main.cpp, line 8
```

Le programme plante et nous indique quel fichier, quelle ligne et quelle condition exactement lui ont posé problème. On peut même ajouter une chaîne de caractères pour rendre le message d'erreur plus clair. Cela est possible car, pour le compilateur, une chaîne de caractères est toujours évaluée comme étant `true`.

```
1 #include <cassert>
2
3 int main()
4 {
5     // Va parfaitement fonctionner et passer à la suite.
6     assert(1 == 1 && "1 doit toujours être égal à 1.");
7     // Va faire planter le programme.
8     assert(1 == 2 && "Oulà, 1 n'est pas égal à 2.");
9     return 0;
10 }
```

```
1 [Visual Studio]
2 Assertion failed: 1 == 2 && "Oulà, 1 n'est pas égal à 2."
3
4 -----
5
6 [GCC]
7 prog.exe: prog.cc:8: int main(): Assertion `1 == 2 && "Oulà, 1
  n'est pas égal à 2."' failed.
8
9 Aborted
10 -----
11
```

III. On passe la deuxième!

```
12 [Clang]
13 prog.exe: prog.cc:8: int main(): Assertion `1 == 2 && "Oulà, 1
    n'est pas égal à 2.'" failed.
14
15 Aborted
```

Pour donner un exemple concret, on va utiliser `assert` pour vérifier les préconditions de l'accès à un élément d'un tableau. Ce faisant, nous éviterons les comportements indéterminés qui surviennent si nous violons le contrat de la fonction.

```
1 #include <cassert>
2 #include <iostream>
3 #include <vector>
4
5 int main()
6 {
7     std::vector<int> const tableau { -4, 8, 452, -9 };
8     int const index { 2 };
9
10    assert(index >= 0 && "L'index ne doit pas être négatif.");
11    assert(index < std::size(tableau) &&
12           "L'index ne doit pas être plus grand que la taille du tableau.");
13    std::cout << "Voici l'élément " << index << " : " <<
14              tableau[index] << std::endl;
15
16    return 0;
17 }
```

Si, lors d'un moment d'inattention par exemple, nous avons utilisé un indice trop grand, alors le plantage provoqué par `assert` nous le rappellera et nous poussera ainsi à corriger le code. La documentation est, pour cela, une aide très précieuse, car elle indique comment réagit une fonction en cas d'arguments invalides. Dans la suite du cours, nous apprendrons à utiliser ces informations.



Mais je ne veux pas que mon programme plante. Pourquoi utiliser `assert`?

En effet, plus un programme est résistant aux erreurs et mieux c'est. Cependant, `assert` s'utilise non pas pour les erreurs de l'utilisateur mais bel et bien **pour celles du programmeur**. Quand le développeur est distrait et écrit du code qui résulte en comportements indéterminés (comme un dépassement d'indice pour un tableau), tout peut arriver et justement, **ce n'est pas ce qu'on attend du programme**. Cela n'a donc pas de sens de vouloir continuer l'exécution.

Par contre, les erreurs de l'utilisateur, par exemple rentrer une chaîne de caractères à la place d'un entier, ne sont absolument pas à traiter avec `assert`. On ne veut pas que notre programme plante brutalement parce que l'utilisateur s'est trompé d'un caractère.

III.4.1.1. Préconditions et assertions

Pour nos propres fonctions, le principe sera le même. Nous allons utiliser les assertions pour vérifier que les paramètres reçus respectent bien les contrats que nous avons préalablement définis.

Prenons un exemple bête avec une fonction chargée de renvoyer le résultat de la division de deux réels.

```
1  #include <cassert>
2  #include <iostream>
3
4  double division(double numérateur, double dénominateur)
5  {
6      assert(dénominateur != 0.0 &&
7             "Le dénominateur ne peut pas valoir zéro.");
8      return numérateur / dénominateur;
9  }
10
11 int main()
12 {
13     std::cout << "Numérateur : ";
14     double numérateur { 0.0 };
15     std::cin >> numérateur;
16
17     double dénominateur { 0.0 };
18     // La responsabilité de vérifier que le contrat est respecté
19     // appartient au code appelant.
20     do
21     {
22         std::cout << "Dénominateur : ";
23         std::cin >> dénominateur;
24     } while (dénominateur == 0.0);
25
26     std::cout << "Résultat : " << division(numérateur,
27                                             dénominateur) << std::endl;
28     return 0;
29 }
```

La fonction `division` pose ici une précondition, qui est que le dénominateur soit non nul. Si elle est respectée, alors elle s'engage, en post-condition, à retourner le résultat de la division du numérateur par le dénominateur.

Au contraire, si le contrat est violé, le programme va s'interrompre sur le champ, car une division par zéro est mathématiquement impossible. Si le programme continuait, tout et n'importe quoi pourrait arriver.

III.4.2. Les tests unitaires à notre aide

III.4.2.1. Pourquoi tester ?

C'est une bonne question. Pourquoi ne pas se contenter de lancer le programme et de voir soi-même avec quelques valeurs si tout va bien? Parce que **cette méthode est mauvaise**. Déjà, nous n'allons tester que quelques valeurs, en général des valeurs qui sont correctes, sans forcément penser aux valeurs problématiques. De plus, ces tests **sont consommateurs en temps**, puisqu'il faut les faire à la main.

À l'opposé, en écrivant des tests nous-mêmes, nous bénéficions de **plusieurs avantages**.

- Nos tests sont **reproductibles**, on peut les lancer à l'infini et vérifier autant que l'on veut. Ils sont déjà écrits, pas de perte de temps donc.
- Conséquence du point précédent, on peut **les lancer au fur et à mesure** que l'on développe pour **vérifier l'absence de régressions**, c'est-à-dire des problèmes qui apparaissent suite à des modifications, comme une fonction qui ne renvoie plus un résultat correct alors qu'auparavant, c'était le cas.
- Comme on ne perd pas de temps manuellement, on peut écrire **plus de tests**, qui permettent donc de détecter plus facilement et rapidement d'éventuels problèmes. Ceux-ci étant détectés plus tôt, ils sont **plus faciles à corriger** et le code voit sa qualité progresser.

III.4.2.2. Un test unitaire, qu'est-ce que c'est ?

Un test unitaire, abrégé TU est un morceau de code dont le seul but est de **tester un autre morceau de code**, afin de vérifier si celui-ci **respecte ses contrats**. Il est dit **unitaire** car il teste ce morceau de code **en toute indépendance** du reste du programme. Pour cela, les morceaux de code qui seront testés unitairement doivent être relativement courts et facilement isolable. Voici des exemples d'identifiants explicites qui vont vous aider à voir ce à quoi peut ressembler un test unitaire.

- `test_sinus_0_retourne_0`
- `test_cosinus_pi_retourne_moins1`
- `test_parenthesage_expression_simple_retourne_true`

Notez que les noms des fonctions sont **un peu long mais très clairs**. Cela est important, car quand certains tests échouent, s'ils ont un nom clair, qui définit sans ambiguïté ce que chaque test vérifie, il est plus facile de corriger ce qui ne va pas. Ainsi, si la fonction `test_cosinus_zero` échoue, vous savez où regarder. À l'inverse, un nom trop vague, trop générique, comme `test_cosinus`, ne nous donnerait aucune indication sur ce qui n'est pas correct.

Et si ce genre de test est nommé unitaire, c'est qu'il en existe d'autres types, que nous ne verrons pas dans le cadre de ce cours, mais dont vous entendrez parler au fur et à mesure que vous continuerez en programmation.

- Les **tests d'intégration**, qui vérifient que différents modules s'intègrent bien entre eux. On peut tester, par exemple, que le module de paiement s'intègre bien au reste de l'application déjà développé.

III. On passe la deuxième!

- Les **tests fonctionnels**, qui vérifient que le produit correspond bien à ce qu'on a demandé. Des tests fonctionnels sur une calculatrice vérifieront que celle-ci offre bien les fonctionnalités demandées (addition, logarithme, etc) par rapport à ce qui avait été défini et prévu (si l'on avait demandé le calcul de cosinus, il faut que la calculatrice l'implémente).
- Les **tests d'UI** (interface utilisateur, de l'anglais *user interface*), qui vérifient, dans le cas d'une application graphique, que les boutons sont bien au bon endroit, les raccourcis, les menus et sous-menus, etc.
- Et encore de nombreux autres que je passerai sous silence.

III.4.2.3. Écrire des tests unitaires

Il existe de nombreuses bibliothèques pour écrire des tests unitaires en C++, certaines basiques, d'autres vraiment complètes, en bref, pour tous les goûts. Dans notre cas, nous n'avons pas besoin d'une solution aussi poussée. Nous allons écrire nos tests très simplement: quelques fonctions que nous lancerons dans le `main` pour vérifier que tout va bien.

Et pour illustrer cette section, nous allons tester un type que nous connaissons bien, `std::vector`. Bien entendu, il a déjà été testé bien plus complètement et profondément par les concepteurs de la bibliothèque standard, mais ce n'est pas grave.

Par exemple, pour tester que la fonction `push_back` fonctionne correctement, on va vérifier qu'elle augmente bien le nombre d'éléments du tableau de un, et que l'élément ajouté se trouve bien à la fin du tableau. On obtient un code qui ressemble à ça.

```
1 #include <cassert>
2 #include <iostream>
3 #include <vector>
4
5 void test_push_back_taille_augmentee()
6 {
7     std::vector<int> tableau { 1, 2, 3 };
8     assert(std::size(tableau) == 3 &&
9         "La taille doit être de 3 avant l'insertion.");
10
11     tableau.push_back(-78);
12     assert(std::size(tableau) == 4 &&
13         "La taille doit être de 4 après l'insertion.");
14 }
15
16 void test_push_back_element_bonne_position()
17 {
18     std::vector<int> tableau { 1, 2, 3, 4 };
19     int const element { 1 };
20
21     tableau.push_back(element);
```

III. On passe la deuxième!

```
21     assert(tableau[std::size(tableau) - 1] == element &&
22            "Le dernier élément doit être 1.");
23 }
24 int main()
25 {
26     test_push_back_taille_augmentee();
27     test_push_back_element_bonne_position();
28
29     return 0;
30 }
```

Si on lance ce code, on voit que rien ne s'affiche et c'est normal. Comme le `std::vector` fournit dans la bibliothèque standard est de qualité, on est assuré qu'il se comportera bien comme il doit. Cependant, si nous avions dû écrire nous-mêmes `std::vector`, alors ces tests nous auraient assurés de la qualité de notre code.

Vous pouvez vous entraîner en écrivant les tests unitaires des différents exemples et exercices des chapitres passés.

III.4.2.4. Les tests unitaires et les postconditions

Rappelez-vous, nous avons défini les postconditions comme étant les contrats que doit respecter une fonction si ses préconditions le sont. Nous pouvons, grâce aux tests unitaires, vérifier que la fonction fait bien son travail en lui fournissant des paramètres corrects et en vérifiant que les **résultats sont valides et correspondent à ce qui est attendu**.

III.4.3. [T.P] Gérer les erreurs d'entrée— Partie IV

Un autre genre de mauvaise pratique consiste à écrire des littéraux en dur dans le code. Un exemple, que j'ai moi-même introduit dans ce cours (honte à moi!), se trouve dans les précédents T.P sur la gestion des entrées.

```
1 std::cin.ignore(255, '\n');
```

On décide d'ignorer jusqu'à 255 caractères, mais que se passe-t-il si l'utilisateur décide d'écrire un roman en guise d'entrée? Nous ne supprimons pas tous les caractères et notre fonction échoue à remplir sa mission: c'est **un bug**, une **erreur de notre part**. Afin de le corriger, il faudrait vider le bon nombre de caractères. Comment faire?

Dans le fichier d'en-tête `<limits>`, il existe un type `std::numeric_limits` qui permet, pour chaque type, comme `int`, `double` et autres, d'en obtenir quelques propriétés, telles la valeur maximale que peut stocker ce type, ou bien la valeur minimale. Comme ces valeurs peuvent varier en fonction de la plateforme et du compilateur, utiliser `std::numeric_limits` à la

III. On passe la deuxième!

place de constantes écrites en dur est **un gage d'une meilleure robustesse et donc de qualité**.

```
1 std::numeric_limits</* type en question */>::/*fonction à appeler
   */
```

```
1 #include <iostream>
2 #include <limits>
3
4 int main()
5 {
6     std::cout << "Valeur minimale d'un int : " <<
7         std::numeric_limits<int>::min() << std::endl;
8     std::cout << "Valeur maximale d'un int : " <<
9         std::numeric_limits<int>::max() << std::endl;
10
11     std::cout << "Valeur minimale d'un double : " <<
12         std::numeric_limits<double>::min() << std::endl;
13     std::cout << "Valeur maximale d'un double : " <<
14         std::numeric_limits<double>::max() << std::endl;
15
16     return 0;
17 }
```

Pour résoudre notre problème soulevé en introduction à cette section, il faut juste savoir **combien de caractères peut stocker au maximum std::cin**. Un `int`, par exemple, pourrait être trop petit et ne pas suffire, ou être trop grand et donc nous créer des problèmes. Un type spécifique, `std::streamsize`, est justement là pour ça.

En se basant sur ces informations, vous avez toutes les clés pour rendre notre fonction de gestion des entrées plus robuste. 🍊

👁 Correction T.P Partie IV

III.4.4. L'exception à la règle

III.4.4.1. La problématique

Au fur et à mesure que vous coderez des programmes de plus en plus complexes, vous verrez qu'il existe plein de cas d'**erreurs extérieures**. Parfois, un fichier sera manquant ou supprimé pendant la lecture. D'autres fois, il n'y aura plus assez de mémoire pour stocker un gros tableau. Ce sont des situations qui n'arrivent que **rarement**, mais que le programme doit **être capable de gérer**.

III. On passe la deuxième!

Prenons un exemple très simple, avec l'ouverture et la lecture d'un fichier. Ce programme récupère un tableau de chaînes de caractères, lues depuis un fichier quelconque.

```
1 #include <fstream>
2 #include <iostream>
3 #include <string>
4 #include <vector>
5
6 std::vector<std::string> lire_fichier(std::string const &
   nom_fichier)
7 {
8     std::vector<std::string> lignes {};
9     std::string ligne { "" };
10    std::ifstream fichier { nom_fichier };
11
12    while (std::getline(fichier, ligne))
13    {
14        lignes.push_back(ligne);
15    }
16
17    return lignes;
18 }
19
20 int main()
21 {
22     std::string nom_fichier { "" };
23     std::cout << "Donnez un nom de fichier : ";
24     std::cin >> nom_fichier;
25
26     auto lignes = lire_fichier(nom_fichier);
27     std::cout << "Voici le contenu du fichier :" << std::endl;
28     for (auto const & ligne : lignes)
29     {
30         std::cout << ligne << std::endl;
31     }
32
33     return 0;
34 }
```

Si nous ne pouvons pas ouvrir le fichier parce que celui-ci n'existe pas, par exemple, alors il faut trouver un moyen de le **signaler à l'utilisateur**.

On peut modifier la fonction pour qu'elle renvoie un booléen, et le tableau sera un argument transmis par référence. C'est la façon de faire en C, mais elle comporte un gros défaut: on peut très bien **ignorer la valeur de retour** de la fonction et par là, ne pas détecter l'erreur potentielle. De plus, elle est peu élégante car elle surcharge le code de paramètres et de valeurs de retour inutiles - au sens où elles ne servent qu'à transmettre les erreurs et ne font pas partie du fonctionnement «théorique» du programme - ainsi que de code de vérification d'erreur lourd. Cette solution n'est donc pas bonne, puisqu'elle nuit à la robustesse et à l'expressivité du code.

III. On passe la deuxième!

On peut aussi afficher un message et demander de saisir un nouveau nom. C'est une solution parmi d'autres, mais elle implique de **rajouter des responsabilités** à notre fonction, puisque en plus de lire un fichier, elle doit maintenant gérer des entrées. De plus, si nous voulons changer nos entrées / sorties de la console vers un fichier, ou même un programme graphique, nous sommes **coincés**.

La solution consiste à utiliser un nouveau concept que je vais introduire: **les exceptions**.

III.4.4.2. C'est quoi une exception ?

C'est un mécanisme qui permet à un morceau de code, comme une fonction, de signaler au morceau de code qui l'a appelé que quelque chose d'exceptionnel - c'est-à-dire indépendant du développeur - s'est passé. On dit du premier morceau qu'il **lance / lève l'exception** (en anglais «*throw*») et que le second **la rattrape** (en anglais «*catch*»). Cela permet de gérer l'erreur dans le morceau de code appelant.

Contrairement aux assertions qui vont signaler la présence d'erreurs internes, de *bugs* à corriger, les exceptions s'utilisent pour les **erreurs externes**, erreurs qui peuvent arriver **même si votre programme est parfaitement codé**. Ne les voyez donc pas comme deux méthodes opposées, mais, bien au contraire, **complémentaires**.

Les exceptions sont notamment utilisées dans le cadre de la programmation par contrat: lorsqu'une fonction dont les préconditions sont vérifiées est dans l'incapacité de vérifier une de ses postconditions à cause d'un événement exceptionnel (un fichier inexistant par exemple), on lance une exception.

III.4.4.3. Lancement des exceptions dans 3, 2, 1...

Tout d'abord et comme pour beaucoup de choses en C++, il faut inclure un fichier d'en-tête, qui se nomme ici `<stdexcept>`. Ensuite, le reste est simple. On utilise le mot-clef `throw` suivi du type de l'exception qu'on veut lancer. Car, de même qu'il existe des types pour stocker des caractères, des entiers, des réels et autres, il existe **différents types d'exception** pour signaler une erreur de taille, une erreur de mémoire, une erreur d'argument, etc.

Celle que nous allons utiliser s'appelle `std::runtime_error` et représente n'importe quelle **erreur à l'exécution**. Elle attend une chaîne de caractères, le message d'erreur.

```
1 #include <fstream>
2 #include <iostream>
3 #include <stdexcept>
4 #include <string>
5 #include <vector>
6
7 std::vector<std::string> lire_fichier(std::string const &
    nom_fichier)
8 {
9     std::vector<std::string> lignes {};
```

III. On passe la deuxième!

```
10     std::string ligne { "" };
11
12     std::ifstream fichier { nom_fichier };
13     if (!fichier)
14     {
15         // Si le fichier ne s'ouvre pas, alors on lance une
16         // exception pour le signaler.
17         throw std::runtime_error("Fichier impossible à ouvrir.");
18     }
19
20     while (std::getline(fichier, ligne))
21     {
22         lignes.push_back(ligne);
23     }
24
25     return lignes;
26 }
27
28 int main()
29 {
30     std::string nom_fichier { "" };
31     std::cout << "Donnez un nom de fichier : ";
32     std::cin >> nom_fichier;
33
34     auto lignes = lire_fichier(nom_fichier);
35     std::cout << "Voici le contenu du fichier : " << std::endl;
36     for (auto const & ligne : lignes)
37     {
38         std::cout << ligne << std::endl;
39     }
40
41     return 0;
42 }
```



Tester l'ouverture d'un flux

Pour tester si notre fichier a bien été ouvert, il suffit d'un simple `if`.

Lancez le programme et regardez-le ... planter lamentablement. Bouahaha, je suis cruel. 🐱

Que s'est-il passé? Lors du déroulement du programme, dès qu'on tombe sur une ligne `throw`, alors l'exécution de la fonction où l'on était **s'arrête immédiatement** (on dit que l'exception «suspend l'exécution») et on revient à la fonction appelante. Si celle-ci n'est pas en mesure de gérer l'exception, alors on revient à la précédente encore. Quand enfin on arrive à la fonction `main` et que celle-ci ne sait pas comment gérer l'exception elle non plus, la norme C++ prévoit de tuer tout simplement le programme.

III.4.4.4. Attends que je t'attrape !

Il nous faut maintenant apprendre à rattraper (on dit aussi «catcher») l'exception, afin de pouvoir la traiter. Pour cela, on utilise la syntaxe `try catch` (`try` étant l'anglais pour «essaye»).

Concrètement, si un code est situé dans un bloc `try` et que ce code lève une exception, l'exécution du programme s'arrête à cette ligne et on part directement dans le bloc `catch` pour traiter la-dite exception. Quant à ce dernier, il faut lui préciser, en plus du mot-clef `catch`, le type d'exception à rattraper, par référence constante. Une fois une exception attrapée, on peut afficher le message qu'elle contient en appelant la fonction `what`.

```
1 #include <fstream>
2 #include <iostream>
3 #include <stdexcept>
4 #include <string>
5 #include <vector>
6
7 std::vector<std::string> lire_fichier(std::string const &
   nom_fichier)
8 {
9     std::vector<std::string> lignes {};
10    std::string ligne { "" };
11
12    std::ifstream fichier { nom_fichier };
13    if (!fichier)
14    {
15        throw std::runtime_error("Fichier impossible à ouvrir.");
16    }
17
18    while (std::getline(fichier, ligne))
19    {
20        lignes.push_back(ligne);
21    }
22
23    return lignes;
24 }
25
26 int main()
27 {
28     std::string nom_fichier { "" };
29     std::cout << "Donnez un nom de fichier : ";
30     std::cin >> nom_fichier;
31
32     try
33     {
34         // Dans le try, on est assuré que toute exception levée
35         // pourra être traitée dans le bloc catch situé après.
36
37         auto lignes = lire_fichier(nom_fichier);
```


III. On passe la deuxième!

```
38     std::cout << "Voici le contenu du fichier :" << std::endl;
39     for (auto const & ligne : lignes)
40     {
41         std::cout << ligne << std::endl;
42     }
43 }
44 // Notez qu'une exception s'attrape par référence constante.
45 catch (std::runtime_error const & exception)
46 {
47     // On affiche la cause de l'exception.
48     std::cout << "Erreur : " << exception.what() << std::endl;
49 }
50
51 return 0;
52 }
```



L'exécution s'arrête lors d'une exception

Je l'ai écrit plus haut, mais je le remets bien lisiblement ici: dès qu'un appel à une fonction, dans le `try`, lève une exception, **le code suivant n'est pas exécuté et on file directement dans le `catch`**. C'est pour ça que le message "Voici le contenu du fichier :" n'est pas affiché.

Voilà, notre programme est maintenant capable de rattraper et de gérer cette situation exceptionnelle, bien qu'on se contente ici de simplement afficher la cause de l'erreur. On peut facilement imaginer un programme qui redemande de saisir un nom de fichier valide tant qu'on rencontre une erreur. Essayez donc de le faire.

© Contenu masqué n°36

III.4.4.5. Attrapez-les tous!

Le `catch` qui suit le `try`, dans notre exemple, permet d'attraper des exceptions de type `std::runtime_error`. Mais certaines fonctions peuvent lancer plusieurs types d'exceptions. Il suffit cependant d'enchaîner les `catch` pour attraper autant d'exceptions différentes qu'on le souhaite.

Prenons l'exemple d'une fonction vue précédemment, permettant de convertir une chaîne de caractère en entier, `std::stoi`. Cette fonction peut lever deux types d'exception.

- Soit `std::invalid_argument` si la chaîne de caractère est invalide et ne peut pas être convertie.
- Soit `std::out_of_range` si le nombre à convertir est trop grand.

III. On passe la deuxième!

[illegible]

Ordre des catch

L'ordre dans lequel vous mettez les `catch` n'a, ici, pas d'importance. En effet, si le `catch` est du même type que l'exception lancée, on rentre dedans, sinon on le saute.

Lancez le programme, modifiez la valeur donnée en argument à `std::stoi` et voyez par vous-mêmes comment on passe dans le `catch` attrapant l'exception lancée.

III.4.4.6. Un type pour les gouverner tous et dans le catch les lier

Il arrive, dans certains cas, qu'on ne veuille pas traiter différemment les exceptions qu'on attrape. Cela signifie t-il qu'on doit écrire autant de `catch` qu'il n'y a d'exceptions potentiellement lançables? Non, on peut utiliser une **exception générique**.

Il existe un type, du nom de `std::exception`, qui décrit une exception générique, une erreur quelconque. Nous reviendrons plus tard sur les propriétés qui rendent ce résultat possible, mais, en attendant, retenez qu'on peut attraper n'importe quelle exception avec.

III. On passe la deuxième!

```
1 #include <iostream>
2 #include <stdexcept>
3 #include <string>
4
5 int main()
6 {
7     try
8     {
9         // Essayez avec un nombre très très grand ou bien avec des
           lettres pour observer le comportement de std::stoi.
10        int entier {
            std::stoi("1000000000000000000000000000000000000000000000000000000000000000");
        };
11        std::cout << "Entier : " << entier << std::endl;
12    }
13    catch (std::exception const& exception)
14    {
15        std::cout << "Une erreur est survenue : " <<
            exception.what() << std::endl;
16    }
17
18    return 0;
19 }
```

Son utilisation n'est absolument pas contraire à l'utilisation de type d'exception précis. On peut ainsi décider d'attraper certaines exceptions bien précises pour faire un traitement particulier, et attraper toutes les autres grâce à `std::exception` et suivre le même processus pour toutes celles-ci. Il n'y a qu'une chose à garder en tête: **le `catch` générique doit toujours se trouver en dernier**.

```

1 #include <iostream>
2 #include <stdexcept>
3 #include <string>
4
5 int main()
6 {
7     try
8     {
9         int entier { std::stoi("+a") };
10        std::cout << "Entier : " << entier << std::endl;
11    }
12    catch (std::invalid_argument const & exception)
13    {
14        // Message particulier affiché seulement dans le cas d'une
15        // exception std::invalid_argument.
16        std::cout << "Argument invalide : " << exception.what() <<
17        std::endl;
18    }
19 }

```

III. On passe la deuxième!

```
16     }
17     catch (std::exception const & exception)
18     {
19         // Message générique pour tous les autres types d'exception
20         // possibles.
21         std::cout << "Erreur : " << exception.what() << std::endl;
22     }
23     return 0;
24 }
```



Les dangers du `catch` générique

- Je répète, le `catch` générique doit **toujours être en dernier**. En effet, les exceptions sont filtrées dans l'ordre de déclaration des `catch`. Comme toutes les exceptions sont des versions plus spécifiques de `std::exception`, elles seront toutes attrapées par le `catch` générique et jamais par les éventuels autres `catch`.
- Ne prenez pas la mauvaise habitude de toujours tout attraper. En effet, vous pourriez attraper par inadvertance des exceptions que vous ne saurez pas gérer, car pas assez d'informations sur le problème, ou pas moyen à cet endroit du code de corriger le-dit problème. N'attrapez donc que celles que vous êtes en mesure de gérer.

III.4.5. [T.P] Gérer les erreurs d'entrée— Partie V

Puisque nous en sommes à améliorer notre T.P, que diriez-vous d'**afficher un meilleur message d'erreur**? En effet, peu importe qu'on soit dans le cas d'une entrée de type invalide (une chaîne de caractères à la place d'un nombre) ou dans le cas d'une entrée incorrecte (8 alors qu'on demande un numéro entre 0 et 5), on affiche toujours le même message. Que diriez-vous de le rendre plus précis? C'est une chose importante que de signaler clairement à l'utilisateur l'erreur qui s'est produite.

Pour nous aider, il existe deux fonctions. Celles-ci seront appliquées ici à `std::cin` mais sont disponibles également pour un flux de fichier ou de chaîne.

- `std::cin.eof()` renvoie `true` si le flux d'entrée est fermé, notamment dans le cas où on arrive à la fin d'un fichier. Cela peut aussi arriver quand vous tapez `Ctrl`+`D` dans le terminal sous GNU/Linux ou `Ctrl`+`Z` sous Windows.
- `std::cin.fail()` renvoie `true` si une erreur quelconque d'entrée apparaît. On va s'en servir notamment dans le cas où une entrée n'est pas du bon type.



Indice

N'oubliez pas que s'il y a fermeture de flux, il faudra lancer une exception puisque la lambda sera incapable de vérifier la postcondition «affecter une valeur correcte fournie

III. On passe la deuxième!



par le flux à la variable».

Avec ces éléments, vous êtes tous bons pour améliorer notre code. Bon courage.

👁 Correction

On remarque qu'on a toujours le dilemme entre fonction ou gestion des erreurs plus personnalisée. Bientôt, nous trouverons une solution à ce problème. 🍏



Gestion d'erreur

Ceci n'est qu'un petit T.P illustratif. Notez bien que dans un programme réel, on ne laisserait pas l'exception faire planter le programme: on la gèrerait de manière agréable pour l'utilisateur. Ici, typiquement, on pourrait afficher un message avant de fermer le programme, pour expliquer à l'utilisateur pourquoi on ne peut plus continuer.

III.4.5.1. En résumé

- Les entrées de l'utilisateur peuvent poser problème, c'est pour ça que nous les vérifions avec des conditions.
- Créer une fonction demande une réflexion sur ses préconditions et postconditions.
- Les assertions sont là pour les erreurs de programmation, les fautes du programmeur. On s'en sert pour les préconditions, notamment.
- Les tests unitaires présentent de nombreux avantages, notamment celui de vérifier les postconditions de nos fonctions, et donc leur bon fonctionnement.
- Les valeurs écrites en dur sont une mauvaise pratique et peuvent même créer des problèmes.
- Les exceptions nous permettent de signaler des erreurs externes, des situations sur lesquelles le programme n'a pas prise.
- Une exception se lève avec le mot-clé `throw`.
- Une exception se rattrape dans un bloc `catch`, qui suit immédiatement un bloc `try`.
- On peut avoir autant de bloc `catch` qu'on veut, un pour chaque type d'exception que l'on veut attraper.
- On peut attraper toutes les exceptions de manière générique en *catchant* `std::exception`.

Contenu masqué

Contenu masqué n°35 :

Correction T.P Partie IV

```
1 #include <iostream>
2 #include <limits>
3
4 void entree_securisee(int & variable)
5 {
6     while (!(std::cin >> variable))
7     {
8         std::cout << "Entrée invalide. Recommence." << std::endl;
9         std::cin.clear();
10
11         std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
12             '\n');
13     }
14 }
15
16 void entree_securisee(double & variable)
17 {
18     while (!(std::cin >> variable))
19     {
20         std::cout << "Entrée invalide. Recommence." << std::endl;
21         std::cin.clear();
22
23         std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
24             '\n');
25     }
26 }
27
28 int main()
29 {
30     int annee { 0 };
31     std::cout <<
32         "Donne-moi une année (essaye de taper un long texte, je ne broncherai p
33     entree_securisee(annee);
34
35     return 0;
36 }
```

Notez que le code est toujours dupliqué. Nous verrons progressivement comment améliorer ce problème dans les chapitres qui suivent.

[Retourner au texte.](#)

Contenu masqué n°36

```
1 #include <fstream>
2 #include <iostream>
3 #include <stdexcept>
4 #include <string>
5 #include <vector>
6
7 std::vector<std::string> lire_fichier(std::string const &
    nom_fichier)
8 {
9     std::vector<std::string> lignes {};
10    std::string ligne { "" };
11
12    std::ifstream fichier { nom_fichier };
13    // Pour tester si un fichier est ouvert et valide, un simple if
        suffit.
14    if (!fichier)
15    {
16        throw std::runtime_error("Fichier impossible à ouvrir.");
17    }
18
19    while (std::getline(fichier, ligne))
20    {
21        lignes.push_back(ligne);
22    }
23
24    return lignes;
25 }
26
27 std::string demander_nom_fichier()
28 {
29     std::string nom_fichier { "" };
30     std::cout << "Donnez un nom de fichier : ";
31     std::cin >> nom_fichier;
32     return nom_fichier;
33 }
34
35 int main()
36 {
37     std::string nom_fichier { demander_nom_fichier() };
38     bool fichier_valide { true };
39
40     do
41     {
42         try
43         {
44             auto lignes = lire_fichier(nom_fichier);
```

III. On passe la deuxième!

```
45         std::cout << "Voici le contenu du fichier :" <<
           std::endl;
46         for (auto const & ligne : lignes)
47         {
48             std::cout << ligne << std::endl;
49         }
50
51         // Si on atteint cette ligne, c'est qu'aucune exception
           n'a été levée et donc que le fichier est OK.
52         fichier_valide = true;
53     }
54     catch (std::runtime_error const & exception)
55     {
56         std::cout << "Erreur : " << exception.what() <<
           std::endl;
57         // Si on atteint cette ligne, c'est qu'une exception a
           été levée car le nom n'est pas bon.
58         fichier_valide = false;
59         // Donc on redemande le nom.
60         nom_fichier = demander_nom_fichier();
61     }
62
63     } while (!fichier_valide);
64
65     return 0;
66 }
```

[Retourner au texte.](#)

Contenu masqué n°37 : Correction

```
1  #include <iostream>
2  #include <limits>
3
4  int main()
5  {
6      std::cout << "Quel jour es-tu né ? ";
7      int jour { 0 };
8      while (!(std::cin >> jour) || jour < 1 || jour > 31)
9      {
10         if (std::cin.eof())
11         {
12             // Dans le cas où le flux est fermé, on renvoie une
               exception.
13             throw std::runtime_error("Le flux a été fermé !");
```


III. On passe la deuxième!

```
14     }
15     else if (std::cin.fail())
16     {
17         // En cas d'entrée invalide.
18         std::cout << "Entrée invalide. Recommence." <<
19             std::endl;
20         std::cin.clear();
21
22         std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
23             '\n');
24     }
25     else
26     {
27         // Cas où l'entrée n'est pas correcte.
28         std::cout << "Le jour doit être compris entre 1 et 31."
29             << std::endl;
30     }
31 }
32
33 std::cout << "Quel mois es-tu né ? ";
34 int mois { 0 };
35 while (!(std::cin >> mois) || mois < 1 || mois > 12)
36 {
37     if (std::cin.eof())
38     {
39         throw std::runtime_error("Le flux a été fermé !");
40     }
41     else if (std::cin.fail())
42     {
43         std::cout << "Entrée invalide. Recommence." <<
44             std::endl;
45         std::cin.clear();
46
47         std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
48             '\n');
49     }
50     else
51     {
52         std::cout << "Le mois doit être compris entre 1 et 12."
53             << std::endl;
54     }
55 }
56
57 std::cout << "Tu es né le " << jour << "/" << mois << "." <<
58     std::endl;
59 return 0;
60 }
```

[Retourner au texte.](#)

III.5. Des fonctions somme toute lambdas

Vous rappelez-vous du chapitre sur les algorithmes? Nous avons découvert que nous pouvons personnaliser, dans une certaine mesure, leur comportement. Mais nous n'avions pas d'autres choix que d'utiliser les prédicats déjà fournis par la bibliothèque standard. Ce chapitre va corriger ça.

Nous allons découvrir comment écrire **nos propres prédicats** en utilisant **les lambdas**.

III.5.1. Une lambda, c'est quoi ?

Une lambda est une fonction, **potentiellement anonyme**, destinée à être utilisée pour des **opérations locales**. Détaillons ces différentes caractéristiques.

- Une lambda est **une fonction**, c'est-à-dire un regroupement de plusieurs instructions, comme nous l'avons vu au chapitre dédié. Rien de bien nouveau.
- Une lambda peut être **anonyme**. Contrairement aux fonctions classiques qui portent obligatoirement un nom, un identificateur, une lambda peut être utilisée **sans qu'on lui donne de nom**.
- Une lambda s'utilise pour des **opérations locales**. Cela signifie qu'une lambda peut être directement écrite dans une fonction ou un algorithme, au plus près de l'endroit où elle est utilisée, au contraire d'une fonction qui est à un endroit bien défini, à part.

i

Les lambdas sont un concept universel

Si les syntaxes que nous allons découvrir sont celles de C++, les lambdas sont répandues et utilisées par de nombreux langages, comme C#, Haskell ou encore Python. Ce concept vient de la [programmation fonctionnelle](#) [↗](#), une autre façon de programmer et dont C++ a repris plusieurs éléments.

III.5.1.1. Pourquoi a-t-on besoin des lambdas ?

En effet, c'est une question à se poser. Après tout, nous avons déjà les fonctions et donc rien ne nous empêche d'utiliser les fonctions pour personnaliser nos algorithmes. C'est tout à fait possible. Seulement voilà, les fonctions présentent deux inconvénients.

- D'abord, le compilateur n'arrive pas à optimiser aussi bien que si on utilise les lambdas. Ceci est l'argument mineur.

III. On passe la deuxième!

- L'argument majeur, c'est qu'on crée une fonction **visible et utilisable par tout le monde**, alors qu'elle n'a pour objectif que d'être **utilisée localement**. On dit que la fonction a une **portée globale**, alors qu'il vaudrait mieux qu'elle ait une **portée locale**.

Les lambdas sont donc une solution à ces deux inconvénients. Bien entendu, C++ vous laisse libre de vos choix. Si vous voulez utiliser des fonctions là où des lambdas sont plus adaptées, rien ne vous l'interdit. Mais s'il y a des outils plus efficaces, autant les utiliser. 🍊

III.5.2. Syntaxe

La syntaxe pour déclarer une lambda a des points communs avec celles des fonctions classiques, mais aussi quelques différences. Examinons la ensemble.

```
1 [zone de capture](paramètres de la lambda) -> type de retour {  
    instructions }
```

- La **zone de capture**: par défaut, une lambda est en totale isolation et ne peut manipuler aucune variable de l'extérieur. Grace à cette zone, la lambda va pouvoir modifier des variables extérieures. Nous allons y revenir.
- Les **paramètres de la lambda**: exactement comme pour les fonctions, les paramètres de la lambda peuvent être présents ou non, avec utilisation de références et/ou **const** possible.
- Le **type de retour**: encore un élément qui nous est familier. Il est écrit après la flèche **->**. Il peut être omis dans quelques cas, mais nous allons l'écrire à chaque fois dans une volonté d'écrire un code clair et explicite.
- Les **instructions**: je ne vous ferai pas l'affront de détailler. 🍊

III.5.2.1. Quelques exemples simples

Ainsi, l'exemple le plus simple et minimaliste que je puisse vous donner serait celui-ci.

```
1 int main()  
2 {  
3     []{};  
4     // Ou version explicite :  
5     []() -> void {};  
6  
7     return 0;  
8 }
```

Personnalisons la lambda pour qu'elle prenne un message à afficher.

III. On passe la deuxième!

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     [](std::string const & message) -> void { std::cout <<
7         "Message reçu : " << message << std::endl; };
8     return 0;
9 }
```

Maintenant, que diriez-vous de la tester, avec un algorithme, pour en voir tout l'intérêt. Disons que l'on veut parcourir un tableau de chaînes de caractères.

J'ai formaté le code pour qu'il soit plus lisible, notamment en allant à la ligne pour les instructions de la lambda. Bien évidemment, vous n'êtes pas obligés de faire comme ça, mais sachez que c'est une écriture lisible et répandue.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <string>
4 #include <vector>
5
6 int main()
7 {
8     std::vector<std::string> const chaines { "Un mot",
9         "Autre chose", "Du blabla", "Du texte", "Des lettres" };
10
11     std::for_each(std::begin(chaines), std::end(chaines),
12         [](std::string const & message) -> void
13     {
14         std::cout << "Message reçu : " << message << std::endl;
15     });
16     return 0;
17 }
```

L'exemple précédent montre une lambda qui ne prend qu'un paramètre. En effet, `std::for_each` attend un **prédicat unaire**, c'est-à-dire que l'algorithme travaille sur un seul élément à la fois. La lambda ne prend donc qu'un seul paramètre.

L'exemple suivant reprend le tri par ordre décroissant, qu'on a déjà vu avec des foncteurs, mais cette fois avec une lambda. Notez que celle-ci prend deux paramètres. C'est parce que `std::sort` attend un **prédicat binaire**, parce que l'algorithme manipule et compare deux éléments en même temps.

III. On passe la deuxième!

```
1 #include <algorithm>
2 #include <array>
3 #include <iostream>
4
5 int main()
6 {
7     std::array<double, 4u> tableau { 1, 3.1415, 2.1878, 1.5 };
8
9     // Tri dans l'ordre décroissant.
10    std::sort(std::begin(tableau), std::end(tableau), [](double a,
11        double b) -> bool
12    {
13        return a > b;
14    });
15
16    for (auto nombre : tableau)
17    {
18        std::cout << nombre << std::endl;
19    }
20
21    return 0;
22 }
```

Continuons avec l'algorithme `std::find_if`. Celui-ci fonctionne exactement comme `std::find`, mais prend en plus un prédicat pour rechercher tout élément le vérifiant. Que diriez-vous, du coup, de chercher tous les éléments impairs d'une collection d'entiers? Comme on regarde élément par élément, il s'agit d'un prédicat unaire.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <vector>
4
5 int main()
6 {
7     std::vector<int> const nombres { 2, 4, 6, 8, 9, 10, 11, 13, 16
8         };
9     auto itérateur { std::begin(nombres) };
10
11    do
12    {
13        itérateur = std::find_if(itérateur, std::end(nombres),
14            [](int nombre) -> bool
15        {
16            return nombre % 2 != 0;
17        });
18
19        if (itérateur != std::end(nombres))
20        {
21            std::cout << *itérateur << " est impair\n";
22        }
23    } while (itérateur != std::end(nombres));
24 }
```

III. On passe la deuxième!

```
18     {
19         std::cout << "Nombre impair : " << *iterateur <<
                std::endl;
20         // On n'oublie pas d'incrémenter, sinon on bouclera
                indéfiniment.
21         ++iterateur;
22     }
23
24     } while (iterateur != std::end(nombres));
25
26     return 0;
27 }
```

Avez-vous noté, dans ces exemples, l'élégance des lambdas? On l'a juste sous les yeux, on voit directement ce que fait l'algorithme. Et en plus, on ne pollue pas l'espace global avec des fonctions à usage unique.

III.5.3. Exercices

III.5.3.1. Vérifier si un nombre est négatif

Nous avons vu, dans le chapitre consacré aux algorithmes, `std::all_of`, qui permet de vérifier que tous les éléments respectent un prédicat. Il y a un algorithme semblable, `std::any_of`, qui s'utilise pareil mais regarde si au moins un élément vérifie le prédicat.

Le but est donc de dire si une collection d'entiers est entièrement positive ou non.

👁 Correction vérifier si un nombre est négatif

III.5.3.2. Tri par valeur absolue

Rien de très compliqué, mais qui diriez-vous de trier un tableau dans l'ordre croissant en utilisant la [valeur absolue](#) des nombres? Cela se fait en utilisant la fonction `std::abs`, disponible dans le fichier d'en-tête `<cmath>`.

👁 Correction tri par valeur absolue

III.5.4. On va stocker ça où?

Imaginons maintenant que l'on veuille appliquer une lambda plusieurs fois. Va-t-on copier-coller le code? Vous connaissez déjà la réponse. Il suffit de **créer une variable**, tout simplement, comme nous l'avons fait jusque-là. La façon la plus simple est d'utiliser `auto`.



Syntaxe et question de goût

On peut, comme le démontre le code, utiliser la syntaxe de C++ moderne, à base d'accolades {}, ou bien utiliser l'initialisation héritée du C à base de égal =. Tout est une question de goût.

J'ai choisi la deuxième syntaxe car je la trouve plus lisible, mais vous restez libres de vos choix.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <vector>
4
5 int main()
6 {
7     auto lambda_avec_accolades { [](int nombre) -> void { std::cout
8         << "Nombre reçu : " << nombre << std::endl; } };
9     auto lambda_avec_egal = [](int nombre) -> void { std::cout <<
10         "Nombre reçu : " << nombre << std::endl; };
11
12     // S'utilise comme une fonction classique, de façon totalement
13     // transparente.
14     lambda_avec_accolades(5);
15
16     std::vector<int> const nombres { 1, 2, 3, 8, 94, 548 };
17     // Ou bien dans un algorithme, aucun problème.
18     std::for_each(std::cbegin(nombres), std::cend(nombres),
19         lambda_avec_egal);
20
21     return 0;
22 }
```

Stocker une lambda permet de garder les avantages de localité tout en évitant de répéter du code, dans le cas où elle est utilisée plusieurs fois.



Par curiosité, quel est le type explicite d'une lambda?

C'est un type complexe qui n'est pas spécifié par la norme. Le compilateur transforme la lambda en autre chose, et cet «autre chose» dépend donc du compilateur que vous utilisez. Les lambdas nous donnent donc un excellent exemple de l'intérêt et de la simplicité apportée par `auto`. 🍊

III.5.5. Paramètres génériques

Tout comme il est possible de laisser le compilateur deviner le type de retour, on peut lui laisser le soin de **déduire les paramètres de la lambda**, en utilisant `auto` à la place du type explicite des paramètres. Bien entendu, le fait de l'utiliser pour certains paramètres **ne nous empêche absolument pas d'avoir des paramètres au type explicitement écrit**.

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     // Le premier paramètre est générique, mais le deuxième est
7     // explicitement marqué comme étant une chaîne.
8     auto lambda = [](auto const & parametre, std::string const &
9                     message) -> void
10    {
11        std::cout << "Paramètre générique reçu : " << parametre <<
12        std::endl;
13        std::cout << "Message reçu : " << message << std::endl;
14    };
15
16    lambda(1, "Test avec un entier.");
17    lambda("str", "Test avec une chaîne de caractères.");
18    lambda(3.1415, "Test avec un flottant.");
19
20    return 0;
21 }
```

Du coup, grâce à ce nouveau pouvoir que nous avons, nous sommes en mesure d'utiliser un même lambda pour afficher deux collections de types différents.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <string>
4 #include <vector>
5
6 int main()
7 {
8     auto enumeration = [](auto const & parametre) -> void
9    {
10        std::cout << "Paramètre reçu : " << parametre << std::endl;
11    };
12
13    // On affiche une collection de std::string.
```


III. On passe la deuxième!

```
14     std::vector<std::string> const chaines { "Un mot",
15         "Autre chose", "Du blabla", "Du texte", "Des lettres" };
16     std::for_each(std::begin(chaines), std::end(chaines),
17         enumeration);
18
19     std::cout << std::endl;
20
21     // Pas de problème non plus avec des entiers.
22     std::vector<int> const nombres { 1, 5, -7, 67, -4454, 7888 };
23     std::for_each(std::begin(nombres), std::end(nombres),
24         enumeration);
25
26     return 0;
27 }
```

III.5.6. [T.P] Gérer les erreurs d'entrée— Partie VI

Avec les connaissances que vous avez, nous pouvons reprendre le code où nous l'avons laissé et commencer à réduire la duplication en utilisant une lambda. Faites-moi une entrée sécurisée mais sans dupliquer le code. Bonne chance. 🍊

```
1  #include <iostream>
2
3  int main()
4  {
5      int jour { 0 };
6      std::cout << "Quel jour es-tu né ? ";
7      // Entrée.
8
9      int mois { 0 };
10     std::cout << "Quel mois ? ";
11     // Entrée.
12
13     int annee { 0 };
14     std::cout << "Quelle année ? ";
15     // Entrée.
16
17     double taille { 0.0 };
18     std::cout << "Quelle taille ? ";
19     // Entree.
20
21     std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
22         annee << " et tu mesures " << taille << "m." << std::endl;
23     return 0;
24 }
```

👁 Correction T.P partie VI

Le code est déjà mieux qu'avant. Il reste cependant un problème: la lambda est locale à la fonction, donc si nous voulons utiliser notre fonction sécurisée à plein d'endroits différents, il va falloir trouver une autre solution. Nous verrons cela très bientôt.

III.5.7. [T.P] Gérer les erreurs d'entrée— Partie VII

Eh oui, nous allons encore enchaîner deux T.P d'affilée! Je l'ai découpé pour que vous puissiez l'aborder avec plus de facilité. 🍏

Relisons le chapitre sur les erreurs. Nous y avons vu une méthode pour non seulement vérifier que l'entrée ne faisait pas planter `std::cin`, mais en plus que la valeur donnée avait un sens. Nous pouvions ainsi vérifier qu'un jour était compris entre 1 et 31, ou qu'un mois est inclus entre 1 et 12. Mais pour cela, nous avons dû extraire notre code de la fonction `entree_securisee`.

Cependant, nous venons de voir que les lambdas acceptent des paramètres génériques avec `auto`. Alors, soyons fous, pourquoi ne pas **essayer de passer une autre lambda**?

```
1 #include <iostream>
2
3 int main()
4 {
5     auto lambda = [](auto parametre, auto fonction) -> void
6     {
7         fonction(parametre);
8     };
9
10    lambda(42, [](int entier) -> void { std::cout << entier <<
11        std::endl; });
12    return 0;
13 }
```

Ça marche parfaitement! Et si nous utilisions ce concept puissant en combinaison avec notre T.P précédent? On pourrait ainsi vérifier que le mois et le jour sont corrects, ou bien même que la personne est née après 1900. Vous avez à présent tous les indices qu'il vous faut pour résoudre ce cas. Bonne chance.

👁 Correction T.P partie VII

III.5.8. Capture en cours...

III.5.8.1. Capture par valeur

Abordons maintenant cette fameuse zone de capture. Par défaut, une lambda est en totale isolation du reste du code. Elle ne peut pas, par exemple, accéder aux variables de la fonction dans laquelle elle est écrite... sauf si on utilise la zone de capture pour dire **à quels objets la lambda peut accéder**. On peut bien sûr en capturer plusieurs, sans problème.

```
1 #include <iostream>
2
3 int main()
4 {
5     bool const un_booleen { true };
6     int const un_entier { 42 };
7     double const un_reel { 3.1415 };
8
9     auto lambda = [un_booleen, un_entier, un_reel]() -> void
10    {
11        std::cout << "Le booléen vaut " << std::boolalpha <<
12            un_booleen << "." << std::endl;
13        std::cout << "L'entier vaut " << un_entier << "." <<
14            std::endl;
15        std::cout << "Le réel vaut " << un_reel << "." <<
16            std::endl;
17    };
18
19    lambda();
20
21    return 0;
22 }
```

Remarquez que si l'on supprime `un_booleen` de la zone de capture, le code ne compile plus. On parle donc de zone de capture car **la lambda a capturé notre variable**, afin de l'utiliser. Cette capture est dite **par valeur**, ou **par copie**, car chaque variable est copiée.



Pour être exact

Il est possible, notamment si vous utilisez GCC ou MinGW (avec Qt Creator, entre autre) que vous avez deux warnings comme ceux ci-dessous.

```
1 warning: lambda capture 'un_booleen' is not required to be
   capture for this use. [-Wunused-lambda-capture]
2 warning: lambda capture 'un_entier' is not required to be
   capture for this use. [-Wunused-lambda-capture]
```

III. On passe la deuxième!



Selon la norme, il y a effectivement certains cas où l'on n'est pas obligé de capturer une variable pour l'utiliser. Cependant, ces cas ne sont pas très nombreux, difficiles à expliquer à ce stade de votre apprentissage et présents surtout par volonté de compatibilité avec l'ancienne norme C++03.

Visual Studio 2019 ne respecte pas la norme sur ce point et donc nous sommes **obligés de capturer toutes les variables que nous voulons utiliser**, si nous voulons que notre code compile avec. Voilà pourquoi je vous invite, très exceptionnellement, à **ignorer le warning** et à utiliser les mêmes codes que dans ce cours.

III.5.8.1.1. Modifier une variable capturée

Reprenons le code précédent et modifions-le pour ne garder qu'un entier modifiable (sans `const` donc). Capturez-le et essayez de le modifier. Vous allez obtenir une belle erreur, et je l'ai fait exprès. 🐱



Je ne comprends pas pourquoi je n'ai pas le droit de les modifier. Si c'est une capture par valeur, alors je manipule des copies bien distinctes des objets originaux, il ne devrait pas y avoir de problème. Pourquoi?

Il y a plusieurs réponses à cette question. L'une est apportée par [Herb Sutter](#) [↗](#), un grand programmeur, très investi dans C++ et l'écriture de sa norme.

Le fait qu'une variable capturée par copie ne soit pas modifiable semble avoir été ajouté parce qu'un utilisateur peut ne pas se rendre compte qu'il modifie une copie et non l'original.

Herb Sutter

Une autre sera détaillée plus tard dans le cours, car il nous manque encore des notions pour aborder ce qui se passe sous le capot.

Retenez qu'on ne peut pas, par défaut, modifier une variable capturée par copie. Heureusement, il suffit d'un simple mot-clef, `mutable`, placé juste après la liste des arguments et avant le type de retour, pour rendre cela possible.

```
1 #include <iostream>
2
3 int main()
4 {
5     int un_entier { 42 };
6
7     // Le mot-clef 'mutable' se place après les paramètres de la
8     // lambda et avant le type de retour.
9     auto lambda = [un_entier]() mutable -> void
10    {
11        un_entier = 0;
12    }
13 }
```

III. On passe la deuxième!

```
11         std::cout << "L'entier vaut " << un_entier <<
12             " dans la lambda." << std::endl;
13     };
14     lambda();
15     std::cout << "L'entier vaut " << un_entier << " dans main." <<
16         std::endl;
17     return 0;
18 }
```

Comme vous le montre le résultat de ce code, **la modification est locale à la lambda**, exactement comme avec les fonctions. Mais ce n'est pas le seul mode de capture existant, comme vous vous en doutez peut-être.

III.5.8.2. Capture par référence

Le principe est exactement le même, sauf que cette fois, on a affaire directement à la variable capturée et non à sa copie et pour ce faire, on rajoute l'esperluette & devant. Cela permet, comme pour les paramètres, d'éviter une copie potentiellement lourde, ou bien de sauvegarder les modifications faites à la variable.

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4
5  int main()
6  {
7      std::vector<int> const nombres { 8, 7, 5, 4, 31, 98, 2, 77, 648
8          };
9
10     int const diviseur { 2 };
11     int somme { 0 };
12
13     // Il n'y a aucun problème à mélanger les captures par valeur
14     // et par référence.
15     std::for_each(std::cbegin(nombres), std::cend(nombres),
16         [diviseur, &somme](int element) -> void
17     {
18         if (element % diviseur == 0)
19         {
20             somme += element;
21         }
22     });
23
24     std::cout << "La somme de tous les éléments divisibles par " <<
25         diviseur << " vaut " << somme << "." << std::endl;
```

III. On passe la deuxième!

```
22     return 0;
23 }
```

Notez qu'on peut capturer certaines variables par référence, d'autres par valeur. C'est exactement comme pour les paramètres d'une fonction, donc cela ne doit pas vous surprendre. 🍊

III.5.8.3. En résumé

- Les lambdas sont des fonctions anonymes, destinées à être utilisées localement.
- Elles réduisent la taille du code, le rendent plus lisible et évitent de créer des fonctions à portée globale alors qu'elles sont à usage local.
- Les paramètres peuvent être explicitement typés ou automatiquement déduit en utilisant `auto`.
- La véritable nouveauté des lambdas, par rapport aux fonctions, est la zone de capture, qui permet de manipuler, par référence et/ou valeur des objets de la portée englobante.
- On peut capturer explicitement ou bien certaines variables, ou bien capturer toutes celles de la portée englobante.

Contenu masqué

Contenu masqué n°38 :

Correction vérifier si un nombre est négatif

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4
5  int main()
6  {
7      std::vector<int> const nombres { 1, 2, 3, 4, 5, 6, -7 };
8
9      if (std::any_of(std::cbegin(nombres), std::cend(nombres),
10                     [](int nombre) -> bool { return nombre < 0; }))
11      {
12          std::cout << "Au moins un nombre est négatif." <<
13                      std::endl;
14      }
15      else
16      {
17          std::cout << "Tous les nombres sont positifs." <<
18                      std::endl;
19      }
20 }
```

III. On passe la deuxième!

```
17
18     return 0;
19 }
```

[Retourner au texte.](#)

Contenu masqué n°39 : Correction tri par valeur absolue

```
1  #include <algorithm>
2  #include <cmath>
3  #include <iostream>
4  #include <string>
5  #include <vector>
6
7  int main()
8  {
9      std::vector<int> nombres { -8, 5, 97, -789, 0, 87, -88, -8555,
10                               10000, 3 };
11      std::sort(std::begin(nombres), std::end(nombres), [](int a, int
12                                                         b) -> bool
13      {
14          return std::abs(a) < std::abs(b);
15      });
16      // Affichons pour vérifier.
17      for (int nombre : nombres)
18      {
19          std::cout << nombre << std::endl;
20      }
21      return 0;
22 }
```

Ne soyez pas surpris par la sortie du programme. Rappelez-vous que nous avons trié le résultat en utilisant la valeur absolue, mais que l’affichage nous montre les éléments triés, mais originaux.



[Retourner au texte.](#)

Contenu masqué n°40 :

Correction T.P partie VI

```
1 #include <iostream>
2 #include <limits>
3 #include <stdexcept>
4
5 int main()
6 {
7     auto entree_securisee = [](auto & variable) -> void
8     {
9         while (!(std::cin >> variable))
10         {
11             if (std::cin.eof())
12             {
13                 throw std::runtime_error("Le flux a été fermé !");
14             }
15             else if (std::cin.fail())
16             {
17                 std::cout << "Entrée invalide. Recommence." <<
18                     std::endl;
19                 std::cin.clear();
20
21                 std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
22                     '\n');
23             }
24         }
25     };
26
27     int jour { 0 };
28     std::cout << "Quel jour es-tu né ? ";
29     entree_securisee(jour);
30
31     int mois { 0 };
32     std::cout << "Quel mois ? ";
33     entree_securisee(mois);
34
35     int annee { 0 };
36     std::cout << "Quelle année ? ";
37     entree_securisee(annee);
38
39     double taille { 0.0 };
40     std::cout << "Quelle taille ? ";
41     entree_securisee(taille);
42
43     std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
44         annee << " et tu mesures " << taille << "m." << std::endl;
45     return 0;
46 }
```


Contenu masqué n°41 : Correction T.P partie VII

```
1 #include <iostream>
2 #include <limits>
3
4 int main()
5 {
6     auto entree_securisee = [](auto & variable, auto predicat) ->
7         void
8     {
9         while (!(std::cin >> variable) || !predicat(variable))
10         {
11             if (std::cin.eof())
12             {
13                 throw std::runtime_error("Le flux a été fermé !");
14             }
15             else if (std::cin.fail())
16             {
17                 std::cout << "Entrée invalide. Recommence." <<
18                     std::endl;
19                 std::cin.clear();
20
21                 std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
22                     '\n');
23             }
24             else
25             {
26                 std::cout << "Le prédicat n'est pas respecté !" <<
27                     std::endl;
28             }
29         }
30     };
31
32     int jour { 0 };
33     std::cout << "Quel jour es-tu né ? ";
34     entree_securisee(jour, [](int jour) -> bool { return jour > 0
35         && jour <= 31; });
36
37     int mois { 0 };
38     std::cout << "Quel mois ? ";
39     entree_securisee(mois, [](int mois) -> bool { return mois > 0
40         && mois <= 12; });
41
42     int annee { 0 };
```

III. On passe la deuxième!

```
36     std::cout << "Quelle année ? ";
37     entree_securisee(annee, [](int annee) -> bool { return annee >
      1900; });
38
39     double taille { 0.0 };
40     std::cout << "Quelle taille ? ";
41     entree_securisee(taille, [](double taille) -> bool { return
      taille > 0.0; });
42
43     std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
      annee << " et tu mesures " << taille << "m." << std::endl;
44     return 0;
45 }
```

Même si le problème soulevé dans le T.P précédent, à savoir qu'on ne peut pas utiliser la lambda ailleurs, reste vrai, notez comme nous avons amélioré notre fonction pour lui faire vérifier que non seulement la valeur est du bon type, mais qu'elle est correcte. [Retourner au texte.](#)

III.6. Envoyez le générique !

Nous avons déjà vu avec les lambdas qu'il est possible d'écrire du code **générique**. On en a vu l'intérêt, notamment pour éviter la duplication de code. Par exemple, notre fonction `entree_securisee` a pu être écrite une seule fois pour tous les types au lieu de notre ancienne version, qu'il fallait écrire une fois pour les `int` et une fois pour les `double`.

On a donc un double intérêt: le code est **plus facilement maintenable** puisque les modifications se font une seule fois et ainsi on évite de réécrire une nouvelle version pour chaque nouveau type.

Le problème, c'est qu'on ne sait faire ça qu'avec des lambdas. Ça serait dommage de se limiter à elles seules. Heureusement, en C++, il est possible de rendre n'importe quelle fonction générique. C'est justement ce que nous allons voir.

III.6.1. Quel beau modèle !

Pour créer des fonctions génériques en C++, on écrit ce que l'on appelle des **templates** (en français «modèle» ou «patron»): au lieu d'écrire directement une fonction, avec des types explicites, on va en écrire une version générique avec des **types génériques**. C'est ensuite le compilateur qui, quand on en aura besoin, utilisera ce modèle pour créer une fonction avec les bons types. On dit qu'il **instancie le modèle**.

Allons droit au but, voici la syntaxe générale, qui ressemble beaucoup à celle utilisée pour créer une fonction.

```
1 template <typename T1, typename T2, ...>
2 type_de_retour identificateur(paramètres)
3 {
4     instructions
5 }
```

Notez aussi l'apparition de deux mot-clés.

- Le mot-clé `template` prévient le compilateur qu'on n'écrit pas une fonction classique mais un modèle.
- Le mot-clé `typename` permet de déclarer un nouveau **nom de type générique**.

Le principe est le suivant: au lieu d'utiliser uniquement des types précis (`int`, `double`, `std::string`, etc) comme on en a l'habitude, on va définir des noms de type quelconques, c'est-à-dire **pouvant être remplacés par n'importe quel type** au moment de l'utilisation

III. On passe la deuxième!

de la fonction. Le compilateur va ensuite créer la fonction à notre place, en remplaçant les types génériques par les types réels des arguments qu'on va lui passer.

i

Parallèle

On peut ici remarquer un parallèle avec les concepts de paramètre et d'argument.

- Le type générique **représente** ce que le template attend comme type, jouant un rôle similaire à celui d'un paramètre de fonction.
- Le type réel est ce qui va être utilisé pour créer la fonction en remplaçant le type générique, à l'instar de l'argument qui est la valeur avec laquelle est appelée la fonction.

Un exemple valant mieux que mille mots, voici un template permettant d'afficher tous les éléments d'un `std::vector` dans la console. On a déjà vu une telle fonction, mais ici la différence est qu'elle va fonctionner **quel que soit le type des éléments dans le tableau**.

```
1 #include <iostream>
2 #include <vector>
3
4 template <typename T>
5 void afficher(std::vector<T> const & v)
6 {
7     for (auto const & e : v)
8     {
9         std::cout << e << std::endl;
10    }
11 }
12
13 int main()
14 {
15     std::vector<int> const tableau_entiers { 1, 3, 5, 7, 9 };
16     afficher(tableau_entiers);
17
18     std::cout << std::endl;
19
20     std::vector<double> const tableau_reels { 1.2, 3.1415, 12.5,
21                                             2.7 };
22     afficher(tableau_reels);
23
24     std::cout << std::endl;
25
26     std::vector<std::string> const tableau_chaines { "Hello",
27                                                     "World", "Les templates", u8"C'est génial !" };
28     afficher(tableau_chaines);
29
30     return 0;
31 }
```

III. On passe la deuxième!

Quand le compilateur voit ce template, il comprend qu'il faut l'instancier trois fois, car il y a trois appels avec trois types différents. Bien sûr, la mécanique interne dépend du compilateur, mais on peut imaginer qu'il génère un code comme celui-ci.

```
1 #include <iostream>
2 #include <vector>
3
4 void afficher(std::vector<int> const & v)
5 {
6     for (auto const & e : v)
7     {
8         std::cout << e << std::endl;
9     }
10 }
11
12 void afficher(std::vector<double> const & v)
13 {
14     for (auto const & e : v)
15     {
16         std::cout << e << std::endl;
17     }
18 }
19
20 void afficher(std::vector<std::string> const & v)
21 {
22     for (auto const & e : v)
23     {
24         std::cout << e << std::endl;
25     }
26 }
27
28 int main()
29 {
30     std::vector<int> const tableau_entiers { 1, 3, 5, 7, 9 };
31     afficher(tableau_entiers);
32
33     std::cout << std::endl;
34
35     std::vector<double> const tableau_reels { 1.2, 3.1415, 12.5,
36                                               2.7 };
37     afficher(tableau_reels);
38
39     std::cout << std::endl;
40
41     std::vector<std::string> const tableau_chaines { "Hello",
42                                                       "World", "Les templates", u8"C'est génial !" };
43     afficher(tableau_chaines);
44
45     return 0;
46 }
```

III. On passe la deuxième!

```
44 }
```



Principe de la programmation générique

On peut remarquer ici un principe de base de ce qu'on appelle la **programmation générique**: on se fiche du type réel des éléments du tableau, du moment qu'ils sont affichables!

En programmation générique, ce qui compte, ce n'est pas le type exact, mais c'est **ce que l'on peut faire avec**.

On peut faire encore mieux: on se fiche du type réel du conteneur, **du moment qu'il est itérable**. Essayons d'appliquer ce principe et d'écrire une fonction `afficher` qui marche avec n'importe quel type de conteneur itérable.

```
1 #include <array>
2 #include <iostream>
3 #include <list>
4 #include <string>
5 #include <vector>
6
7 template <typename Collection>
8 void afficher(Collection const & iterable)
9 {
10     for (auto const & e : iterable)
11     {
12         std::cout << e << std::endl;
13     }
14 }
15
16 int main()
17 {
18     // Un std::vector est itérable, donc aucun problème.
19     std::vector<int> tableau_entiers { 1, 3, 5, 7, 9 };
20     afficher(tableau_entiers);
21
22     std::cout << std::endl;
23
24     // De même que pour un std::array.
25     std::array<double, 4> const tableau_reels { 1.2, 3.1415, 12.5,
26         2.7 };
27     afficher(tableau_reels);
28
29     std::cout << std::endl;
30
31     // Ou bien une std::list.
```

III. On passe la deuxième!

```
31     std::list<std::string> const liste_chaines { "Hello", "World",  
32         "Les templates", u8"C'est génial !" };  
33     afficher(liste_chaines);  
34  
35     std::cout << std::endl;  
36  
37     // Une chaîne de caractères est tout à fait itérable.  
38     std::string const chaine {  
39         u8"Ça marche même avec des chaînes de caractères !" };  
40     afficher(chaine);  
41  
42     return 0;  
43 }
```

Pour vous aider à bien visualiser ce qui se passe, voici ce que peut produire le compilateur quand il a fini d'instancier le modèle.

```
1  #include <array>  
2  #include <iostream>  
3  #include <list>  
4  #include <string>  
5  #include <vector>  
6  
7  void afficher(std::vector<int> const & iterable)  
8  {  
9      for (auto const & e : iterable)  
10     {  
11         std::cout << e << std::endl;  
12     }  
13 }  
14  
15 void afficher(std::array<double, 4> const & iterable)  
16 {  
17     for (auto const & e : iterable)  
18     {  
19         std::cout << e << std::endl;  
20     }  
21 }  
22  
23 void afficher(std::list<std::string> const & iterable)  
24 {  
25     for (auto const & e : iterable)  
26     {  
27         std::cout << e << std::endl;  
28     }  
29 }  
30  
31 void afficher(std::string const & iterable)
```

III. On passe la deuxième!

```
32 {
33     for (auto const & e : iterable)
34     {
35         std::cout << e << std::endl;
36     }
37 }
38
39 int main()
40 {
41     std::vector<int> tableau_entiers { 1, 3, 5, 7, 9 };
42     afficher(tableau_entiers);
43
44     std::cout << std::endl;
45
46     std::array<double, 4> const tableau_reels { 1.2, 3.1415, 12.5,
47         2.7 };
48     afficher(tableau_reels);
49
50     std::cout << std::endl;
51
52     std::list<std::string> const liste_chaines { "Hello", "World",
53         "Les templates", u8"C'est génial !" };
54     afficher(liste_chaines);
55
56     std::cout << std::endl;
57
58     std::string const chaine {
59         u8"Ça marche même avec des chaînes de caractères !" };
60     afficher(chaine);
61
62     return 0;
63 }
```

Magique, non? 🧙🏻

III.6.1.1. La bibliothèque standard: une fourmilière de modèles

En fait, on a vu et utilisé un **nombre incalculable de fonctions génériques** dans la **bibliothèque standard**: celles renvoyant des itérateurs (`std::begin`, `std::end`, etc), les conteneurs, les algorithmes, etc. En effet, toutes ces fonctions peuvent s'utiliser avec des arguments de types différents. Pour `std::begin`, par exemple, il suffit que le type passé en argument ait la capacité de fournir un itérateur sur le premier élément.

Le principe même des templates est d'oublier le type pour se concentrer sur **ce que peut faire ce type**, et c'est très intéressant pour la bibliothèque standard puisqu'elle a vocation à être utilisée dans des contextes variés. En fait, sans les templates, il serait très difficile de concevoir la bibliothèque standard du C++. Cela nous donne un exemple concret de leur intérêt.

III.6.1.2. Un point de vocabulaire

Lorsqu'on parle de templates, on utilise un vocabulaire spécifique pour pouvoir parler de chaque étape dans leur processus de fonctionnement. Reprenons-le pour bien que vous le mémorisiez.

- Le template est la description générique de ce que fait la fonction.
- Lorsque le compilateur crée une fonction à partir du modèle pour l'utiliser avec des types donnés, on parle **d'instanciation**.
- Lorsqu'on appelle la fonction instanciée, le nom ne change pas de ce dont on a l'habitude. On parle toujours **d'appel de fonction**.

III.6.2. [T.P] Gérer les erreurs d'entrée - Partie VIII

Nous avons déjà, dans la partie précédente, éliminé la duplication de code en utilisant une lambda. Nous avons également appris à transmettre une fonction, ce qui permet d'ajouter des vérifications supplémentaires. Cependant, notre code était dans une portée locale. Maintenant, vous disposez de toutes les clés pour corriger ce problème.

Le code à sécuriser est toujours le même.

```
1  #include <iostream>
2
3  int main()
4  {
5      int jour { 0 };
6      std::cout << "Quel jour es-tu né ? ";
7      // Entrée.
8
9      int mois { 0 };
10     std::cout << "Quel mois ? ";
11     // Entrée.
12
13     int annee { 0 };
14     std::cout << "Quelle année ? ";
15     // Entrée.
16
17     double taille { 0.0 };
18     std::cout << "Quelle taille ? ";
19     // Entree.
20
21     std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
        annee << " et tu mesures " << taille << "m." << std::endl;
22     return 0;
23 }
```

Et voilà! Maintenant, `entree_securisee` est générique et utilisable à une portée globale, on a tout ce que l'on voulait! Comme vous voyez, on est de plus en plus efficace pour éviter la duplication de code.

III.6.3. Instanciation explicite

Il arrive parfois que le compilateur ne puisse pas déterminer avec quels types il doit instancier un template, typiquement lorsque les types génériques n'apparaissent pas dans les paramètres de la fonction. C'est en particulier le cas lorsqu'il n'y a tout simplement pas de paramètre.

Par exemple, imaginons que notre fonction `entree_securisee`, au lieu de prendre la variable à modifier en paramètre, crée une variable du bon type et renvoie le résultat.

```
1 template <typename T>
2 T entree_securisee()
3 {
4     // Initialisation par défaut, qui marche pour tous les types.
5     T variable {};
6
7     while (!(std::cin >> variable))
8     {
9         std::cout << "Entrée invalide. Recommence." << std::endl;
10        std::cin.clear();
11
12        std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
13                        '\n');
14    }
15    return variable;
16 }
```

Jusque-là, pas de problème. Essayons d'appeler la fonction.

```
1 #include <iostream>
2 #include <limits>
3
4 template <typename T>
5 T entree_securisee()
6 {
7     T variable {};
8     while (!(std::cin >> variable))
```

III. On passe la deuxième!

```
9      {
10          std::cout << "Entrée invalide. Recommence." << std::endl;
11          std::cin.clear();
12
13          std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
14                          '\n');
15      }
16
17      return variable;
18  }
19
20  int main()
21  {
22      std::cout << "Quel jour es-tu né ? ";
23      int jour { entree_securisee() };
24
25      std::cout << "Quel mois ? ";
26      int mois { entree_securisee() };
27
28      std::cout << "Quelle année ? ";
29      int annee { entree_securisee() };
30
31      std::cout << "Quelle taille ? ";
32      double taille { entree_securisee() };
33
34      std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
35          annee << " et tu mesures " << taille << "m." << std::endl;
36      return 0;
37  }
```

Et là, c'est le drame: impossible de compiler, il y a des erreurs remontées.

```
1 [Visual Studio]
2 Erreur C2783 'T entree_securisee(void)' : impossible de déduire
3   l'argument modèle pour 'T'.
4 Error C2783 'T entree_securisee(void)' : could not deduce template
5   argument for 'T'.
6 -----
7 [GCC]
8 prog.cc: In function 'int main()':
9 prog.cc:21:33: error: no matching function for call to
10   'entree_securisee()'
11   21 |         int jour { entree_securisee() };
12       |                     ^
13 prog.cc:5:3: note: candidate: 'template<class T> T
14   entree_securisee()'
```

III. On passe la deuxième!

```
13     5 | T entree_securisee()
14     | ^~~~~~
15 prog.cc:5:3: note:   template argument deduction/substitution
    failed:
16 prog.cc:21:33: note:   couldn't deduce template parameter 'T'
17   21 |         int jour { entree_securisee() };
18     |                             ^
19 prog.cc:24:33: error: no matching function for call to
    'entree_securisee()'
20   24 |         int mois { entree_securisee() };
21     |                             ^
22 prog.cc:5:3: note: candidate: 'template<class T> T
    entree_securisee()'
23     5 | T entree_securisee()
24     | ^~~~~~
25 prog.cc:5:3: note:   template argument deduction/substitution
    failed:
26 prog.cc:24:33: note:   couldn't deduce template parameter 'T'
27   24 |         int mois { entree_securisee() };
28     |                             ^
29 prog.cc:27:34: error: no matching function for call to
    'entree_securisee()'
30   27 |         int annee { entree_securisee() };
31     |                             ^
32 prog.cc:5:3: note: candidate: 'template<class T> T
    entree_securisee()'
33     5 | T entree_securisee()
34     | ^~~~~~
35 prog.cc:5:3: note:   template argument deduction/substitution
    failed:
36 prog.cc:27:34: note:   couldn't deduce template parameter 'T'
37   27 |         int annee { entree_securisee() };
38     |                             ^
39 prog.cc:30:38: error: no matching function for call to
    'entree_securisee()'
40   30 |         double taille { entree_securisee() };
41     |                             ^
42 prog.cc:5:3: note: candidate: 'template<class T> T
    entree_securisee()'
43     5 | T entree_securisee()
44     | ^~~~~~
45 prog.cc:5:3: note:   template argument deduction/substitution
    failed:
46 prog.cc:30:38: note:   couldn't deduce template parameter 'T'
47   30 |         double taille { entree_securisee() };
48     |                             ^
49
50 -----
51
52 [Clang]
```

III. On passe la deuxième!

```
53 prog.cc:21:16: error: no matching function for call to
    'entree_securisee'
54     int jour { entree_securisee() };
55                 ^~~~~~
56 prog.cc:5:3: note: candidate template ignored: couldn't infer
    template argument 'T'
57 T entree_securisee()
58   ^
59 prog.cc:24:16: error: no matching function for call to
    'entree_securisee'
60     int mois { entree_securisee() };
61                 ^~~~~~
62 prog.cc:5:3: note: candidate template ignored: couldn't infer
    template argument 'T'
63 T entree_securisee()
64   ^
65 prog.cc:27:17: error: no matching function for call to
    'entree_securisee'
66     int annee { entree_securisee() };
67                 ^~~~~~
68 prog.cc:5:3: note: candidate template ignored: couldn't infer
    template argument 'T'
69 T entree_securisee()
70   ^
71 prog.cc:30:21: error: no matching function for call to
    'entree_securisee'
72     double taille { entree_securisee() };
73                     ^~~~~~
74 prog.cc:5:3: note: candidate template ignored: couldn't infer
    template argument 'T'
75 T entree_securisee()
76   ^
77 4 errors generated.
```

Leur origine est simple: le compilateur n'a **aucun moyen de savoir avec quel type instancier le template**. Ce cas se présente aussi quand, pour un type générique attendu, on en donne plusieurs différents.

```
1  #include <iostream>
2
3  template <typename T>
4  T addition(T x, T y)
5  {
6      return x + y;
7  }
8
9  int main()
10 {
```

III. On passe la deuxième!

```
11 // Aucun problème, on a deux int ici.
12 std::cout << addition(1, 2) << std::endl;
13 // Oups. Le type déduit, c'est quoi alors ? int ou double ?
14 std::cout << addition(1, 3.1415) << std::endl;
15
16 return 0;
17 }
```

Dans le deuxième cas, il y a ambiguïté. Pour la lever, il suffit de préciser explicitement les types devant être utilisés pour l'instanciation, en les mettant entre chevrons juste avant les parenthèses de paramètres. La syntaxe ne vous est pas totalement inconnue, puisqu'elle est proche de celle utilisée pour les conteneurs (`std::vector<int>`, `std::list<double>`, etc).

```
1 #include <iostream>
2 #include <limits>
3
4 template <typename T>
5 T entree_securisee()
6 {
7     T variable {};
8     while (!(std::cin >> variable))
9     {
10         std::cout << "Entrée invalide. Recommence." << std::endl;
11         std::cin.clear();
12
13         std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
14             '\n');
15     }
16
17     return variable;
18 }
19
20 template <typename T>
21 T addition(T x, T y)
22 {
23     return x + y;
24 }
25
26 int main()
27 {
28     std::cout << "Quel jour es-tu né ? ";
29     int jour { entree_securisee<int>() };
30
31     std::cout << "Quel mois ? ";
32     int mois { entree_securisee<int>() };
33
34     std::cout << "Quelle année ? ";
35     int annee { entree_securisee<int>() };
36 }
```

III. On passe la deuxième!

```
34
35     std::cout << "Quelle taille ? ";
36     double taille { entree_securisee<double>() };
37
38     std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
        annee << " et tu mesures " << taille << "m." << std::endl;
39
40     std::cout << "1 + 3.1415 = " << addition<double>(1, 3.1415) <<
        std::endl;
41     return 0;
42 }
```



Instanciation explicite de template

Cette précision explicite des types pour l'instanciation sert à lever les ambiguïtés, mais elle peut aussi s'utiliser dans d'autres cas. Par exemple, vous pouvez tout à fait préciser les types si vous trouvez ça plus lisible, même s'il n'y a pas d'ambiguïté.

III.6.4. Ce type là, c'est un cas !

On a vu que les templates sont une fonctionnalité très puissante pour écrire du code **générique**, c'est-à-dire ne dépendant pas des types des paramètres mais des **services**, des **fonctionnalités** proposées par ces types. Cependant, parfois, on aimerait que, pour un type particulier, la fonction se comporte différemment.

Revenons au template `afficher` qu'on a défini précédemment.

```
1 #include <array>
2 #include <iostream>
3 #include <list>
4 #include <string>
5 #include <vector>
6
7 template <typename Collection>
8 void afficher(Collection const & iterable)
9 {
10     for (auto const & e : iterable)
11     {
12         std::cout << e << std::endl;
13     }
14 }
15
16 int main()
17 {
18     std::vector<int> tableau_entiers { 1, 3, 5, 7, 9 };
```

III. On passe la deuxième!

```
19     afficher(tableau_entiers);
20
21     std::cout << std::endl;
22
23     std::array<double, 4> const tableau_reels { 1.2, 3.1415, 12.5,
24         2.7 };
25     afficher(tableau_reels);
26
27     std::cout << std::endl;
28
29     std::list<std::string> const liste_chaines { "Hello", "World",
30         "Les templates.", u8"C'est génial !" };
31     afficher(liste_chaines);
32
33     std::cout << std::endl;
34
35     std::string const chaine {
36         u8"Ça marche même avec des chaînes de caractères !" };
37     afficher(chaine);
38
39     return 0;
40 }
```

Ce code donne la sortie suivante.

```
1 1
2 3
3 5
4 7
5 9
6
7 1.2
8 3.1415
9 12.5
10 2.7
11
12 Hello
13 World
14 Les templates.
15 C'est génial !
16
17 Ç
18 a
19
20 m
21 a
22 r
23 c
```


III. On passe la deuxième!

```
24 h
25 e
26
27 m
28 ê
29 m
30 e
31
32 a
33 v
34 e
35 c
36
37 d
38 e
39 s
40
41 c
42 h
43 a
44 î
45 n
46 e
47 s
48
49 d
50 e
51
52 c
53 a
54 r
55 a
56 c
57 t
58 è
59 r
60 e
61 s
62
63 !
```

Le résultat est assez désagréable à lire lorsqu'on utilise la fonction avec le type `std::string`. Heureusement, on a un moyen très simple de modifier le comportement de notre template pour un type de paramètre en particulier. C'est une notion que nous avons abordée dans le chapitre sur les fonctions: **la surcharge**. On va surcharger la fonction `afficher` pour les chaînes de caractères.

III. On passe la deuxième!

```
1 #include <array>
2 #include <iostream>
3 #include <list>
4 #include <string>
5 #include <vector>
6
7 template <typename Collection>
8 void afficher(Collection const & iterable)
9 {
10     for (auto const & e : iterable)
11     {
12         std::cout << e << std::endl;
13     }
14 }
15
16 // Version non-template ne fonctionnant que pour les chaînes de
17 // caractères.
18 void afficher(std::string const & str)
19 {
20     std::cout << str << std::endl;
21 }
22
23 int main()
24 {
25     std::vector<int> tableau_entiers { 1, 3, 5, 7, 9 };
26     afficher(tableau_entiers);
27
28     std::cout << std::endl;
29
30     std::array<double, 4> const tableau_reels { 1.2, 3.1415, 12.5,
31         2.7 };
32     afficher(tableau_reels);
33
34     std::cout << std::endl;
35
36     std::list<std::string> const liste_chaines { "Hello", "World",
37         "Les templates.", u8"C'est génial !" };
38     afficher(liste_chaines);
39
40     std::cout << std::endl;
41
42     std::string const chaine {
43         u8"Ça marche même avec des chaînes de caractères !" };
44     afficher(chaine);
45
46     return 0;
47 }
```

Ce qui donne un résultat bien plus lisible en sortie.

III. On passe la deuxième!

```
1 1
2 3
3 5
4 7
5 9
6
7 1.2
8 3.1415
9 12.5
10 2.7
11
12 hello
13 world
14 Les templates.
15 C'est génial !
16
17 Ça marche même avec des chaînes de caractères !
```

On a donc un template qui permet d'afficher toutes sortes de collections et un traitement à part pour les `std::string`. Bien entendu, on peut créer d'autres surcharges pour d'autres types, si on veut qu'ils aient un comportement bien précis.



Surcharge prioritaire

Le changement du comportement par surcharge fonctionne car le compilateur prend **en priorité les fonctions non-templates** lors du choix de la bonne surcharge. En effet, a priori, il devrait y avoir ambiguïté (et donc erreur), car il y a deux fonctions possibles: celle générée à partir du modèle et celle écrite explicitement.

Heureusement, le choix a été fait par les concepteurs du C++ d'accorder la priorité aux fonctions écrites explicitement. Cela permet donc de rendre valide ce genre de code.

III.6.4.1. En résumé

- Les templates permettent de gagner en généricité et donc de réduire la duplication de code.
- C'est une généralisation du principe de généricité, qu'on a déjà observé avec les lambdas.
- Cela permet une plus grande flexibilité ainsi que l'écriture de fonctions génériques dans la portée globale.
- La bibliothèque standard regorge de templates, car la généricité en est un aspect important.
- On peut préciser, ou non, explicitement les types lors de l'instanciation, mais certains cas rendent cette précision obligatoire.
- On peut surcharger une fonction générique pour personnaliser son comportement pour certains types explicitement donnés.

Contenu masqué

Contenu masqué n°42 : Correction T.P partie VIII

```
1 #include <iostream>
2 #include <limits>
3
4 template <typename T, typename Predicat>
5 void entree_securisee(T & variable, Predicat predicat)
6 {
7     while (!(std::cin >> variable) || !predicat(variable))
8     {
9         if (std::cin.eof())
10         {
11             throw std::runtime_error("Le flux a été fermé !");
12         }
13         else if (std::cin.fail())
14         {
15             std::cout << "Entrée invalide. Recommence." <<
16                 std::endl;
17             std::cin.clear();
18
19             std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
20                 '\n');
21         }
22         else
23         {
24             std::cout << "Le prédicat n'est pas respecté !" <<
25                 std::endl;
26         }
27     }
28 }
29
30 int main()
31 {
32     int jour { 0 };
33     std::cout << "Quel jour es-tu né ? ";
34     entree_securisee(jour, [](int jour) -> bool { return jour > 0
35         && jour <= 31; });
36
37     int mois { 0 };
38     std::cout << "Quel mois ? ";
39     entree_securisee(mois, [](int mois) -> bool { return mois > 0
40         && mois <= 12; });
41
42     int annee { 0 };
```

III. On passe la deuxième!

```
37     std::cout << "Quelle année ? ";
38     entree_securisee(annee, [](int annee) -> bool { return annee >
1900; });
39
40     double taille { 0.0 };
41     std::cout << "Quelle taille ? ";
42     entree_securisee(taille, [](double taille) -> bool { return
taille > 0.0; });
43
44     std::cout << "Tu es né le " << jour << "/" << mois << "/" <<
annee << " et tu mesures " << taille << "m." << std::endl;
45     return 0;
46 }
```



Templates et algorithmes standards

Avez-vous noté les deux lignes du début de notre modèle?

```
1  template <typename T, typename Predicat>
2  void entree_securisee(T & variable, Predicat predicat)
```

Grâce aux templates, nous pouvons passer une fonction sans effort ni difficulté. C'est de la même manière que procède la bibliothèque standard pour les algorithmes personnalisables.

Enfin, grâce à la surcharge, on peut combiner les deux versions de notre fonction, pour choisir, au choix, de passer ou non un prédicat.

```
1  template<typename T>
2  void entree_securisee(T& variable)
3  {
4      while (!(std::cin >> variable))
5      {
6          if (std::cin.eof())
7          {
8              break;
9          }
10         else if (std::cin.fail())
11         {
12             std::cout << "Entrée invalide. Recommence." <<
std::endl;
13             std::cin.clear();
14
15             std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');
16         }
17     }
18 }
```

III. On passe la deuxième!

```
17     {
18         std::cout << "Le prédicat n'est pas respecté !" <<
            std::endl;
19     }
20 }
21 }
22
23 template <typename T, typename Predicat>
24 void entree_securisee(T & variable, Predicat predicat)
25 {
26     while (!(std::cin >> variable) || !predicat(variable))
27     {
28         if (std::cin.eof())
29         {
30             break;
31         }
32         else if (std::cin.fail())
33         {
34             std::cout << "Entrée invalide. Recommence." <<
                std::endl;
35             std::cin.clear();
36
37             std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
                '\n');
38         }
39         else
40         {
41             std::cout << "Le prédicat n'est pas respecté !" <<
                std::endl;
42         }
43     }
44 }
```

[Retourner au texte.](#)

III.7. De nouvelles structures de données

Lors de ce cours, nous avons eu l'occasion d'utiliser des données sous de nombreuses formes: types de base, chaînes de caractères, collections de données, etc. Cependant, toutes ces formes de données sont finalement assez rudimentaires, et on souhaite parfois avoir plus de souplesse dans la manière dont sont structurées nos données.

Dans ce chapitre, nous allons justement découvrir de **nouvelles structures de données** qui vont nous offrir plus de flexibilité dans nos programmes. Pour chaque structure, nous verrons comment elle s'utilise et en quoi elle est utile.

III.7.1. struct— Un agrégat de données

En C++, il existe une structure de données permettant de regrouper plusieurs données en un seul bloc. Dans leur grande originalité, elle s'appelle... **structure**. 🍊



Point de vocabulaire

À partir de maintenant, j'emploierai le terme «structure» pour désigner l'agrégat que nous allons introduire et l'expression «structure de données» pour désigner les formes complexes de données de manière générale.

Les structures sont très utiles pour regrouper des données qui ont un lien entre elles et travailler dessus de manière uniforme, et non sur chaque donnée séparément. Voyons un exemple concret avec un petit exercice.

III.7.1.1. Stockage d'informations personnelles¹

Essayez de créer un programme qui demande des informations sur une personne et les stocke dans un fichier `Prenom.Nom.csv` sous le format suivant.

1	nom,prenom,sexe,age
---	---------------------

En sortie, la console devrait ressembler à quelque chose comme ci-dessous.

1. En total accord avec le RGPD.

III. On passe la deuxième!

```
1 Nom ? Lagrume
2 Prénom ? Clem
3 Sexe ? Fruit
4 Age ? 4
5
6 Ces données seront enregistrées dans le fichier Clem.Lagrume.csv.
```

Le programme va ensuite enregistrer ces informations dans un fichier `Clem.Lagrume.csv`, organisé comme suit.

```
1 Lagrume,Clem,Fruit,4
```

Bonne chance. 🍊

👁 Correction

III.7.1.2. Analyse de l'exercice

On a réussi à mener à bien l'exercice, mais on peut remarquer des pistes d'amélioration:

- Les fonctions `demander_infos` et `enregistrer_infos` prennent beaucoup de paramètres. Mais, à la limite, c'est du détail.
- Plus grave, si on ajoute une information sur une personne, il faudra modifier les prototypes de toutes nos fonctions! Cela porte gravement atteinte à l'évolutivité de nos programmes.
- Les données que l'on manipule sont liées et pourraient s'utiliser en tant qu'ensemble.

III.7.1.3. Introduction aux structures

La solution est justement d'utiliser une structure. Cela va nous permettre de créer un type, que l'on va par exemple appeler `InformationsPersonnelles`, regroupant l'intégralité des informations que l'on souhaite enregistrer. La syntaxe de création est la suivante.

```
1 struct identifiant
2 {
3     *membres*
4 };
```

- L'identifiant de la structure suit **les mêmes règles de nommage** que les identifiants de variables et de fonctions.

III. On passe la deuxième!

- En termes de convention, nous suivrons une convention appelée CamelCase: on colle les mots en commençant chaque mot par une majuscule. Il en existe d'autres, mais nous utiliserons celle-là car elle permet facilement de distinguer les variables et les types.
- La définition d'une structure doit **se terminer par un point virgule**. Il est fréquent de l'oublier.

Par exemple, une structure représentant les données personnelles d'une personne, dont on pourrait se servir dans l'exercice précédent, peut être définie comme suit.

```
1 struct InformationsPersonnelles
2 {
3     std::string prenom;
4     std::string nom;
5     std::string sexe;
6     int age;
7 };
```

III.7.1.3.1. Initialisation

Pour initialiser une structure, nous disposons de plusieurs façons de faire, qui ne vont pas vous surprendre, puisque nous les connaissons déjà pour les variables «classiques».

```
1 // Initialisation par défaut.
2 InformationsPersonnelles infos {};
3
4 // Initialisation à des valeurs choisies.
5 InformationsPersonnelles infos { "Clem", "Lagrume", "Fruit", 4 };
6
7 // Autre syntaxe, utilisant le signe = comme en C.
8 InformationsPersonnelles infos = { "Clem", "Lagrume", "Fruit", 4 };
```

Notez que, lorsqu'on initialise avec des valeurs choisies, il faut les donner dans l'ordre dans lequel elles sont définies dans la structure. On ne peut pas donner un entier comme première valeur, si celle-ci est déclarée dans la définition comme étant une chaîne de caractères.

III.7.1.3.2. Voici les membres

Pour manipuler un membre, c'est-à-dire une variable appartenant à la structure, il suffit d'utiliser la syntaxe `structure.membre`, qui fonctionne tant en lecture qu'en écriture.

```
1 #include <iostream>
2 #include <string>
3
```

III. On passe la deuxième!

```
4 struct InformationsPersonnelles
5 {
6     std::string prenom;
7     std::string nom;
8     std::string sexe;
9     int age;
10 };
11
12 int main()
13 {
14     InformationsPersonnelles info_clem { "Clem", "Lagrume",
15         "Fruit", 5 };
16
17     // Modification.
18     info_clem.age = 4;
19     // Utilisation en lecture.
20     std::cout << "Je m'appelle " << info_clem.prenom << " " <<
21         info_clem.nom << " et j'ai " << info_clem.age << " ans." <<
22         std::endl;
23
24     return 0;
25 }
```

```
1 Je m'appelle Clem Lagrume et j'ai 4 ans.
```

Bien entendu, comme toujours, la modification d'un membre n'est possible que si la variable n'est pas déclarée comme étant `const`.

III.7.1.4. Exercice

Sans plus attendre, modifions le code de l'exercice précédent pour utiliser une structure. C'est simple, modifiez-le pour utiliser une structure au lieu de se trimballer plein de variables.

👁 Correction

Et voilà! Le code est **plus lisible** et **plus évolutif**. Vous pouvez essayer d'ajouter une information à enregistrer, par exemple la couleur préférée ou la date de naissance, et vous verrez que vous aurez **uniquement des ajouts à faire**. C'est un critère important d'évolutivité du code. En particulier, les prototypes des fonctions **ne changeront pas**, seules les implémentations changeront. Du point de vue de l'utilisation, c'est transparent, on ne voit rien de changé, tout le code les utilisant n'aura donc pas à être réécrit.



Raccourci

Remarquez que, maintenant qu'on peut directement renvoyer le résultat de `demander_infos`, on n'est plus obligé de prendre en paramètres des références vers les variables à modifier. Cela permet donc d'écrire, non sans élégance:

```
1 auto infos = demander_infos();
```

III.7.2. `std::tuple`— Une collection hétérogène

Les structures ne sont pas le seul moyen de représenter un ensemble de données de types variés. Il existe un autre outil nommé `std::tuple`, qui est une **collection hétérogène de taille fixée**. Hétérogène, car on peut stocker des informations de types différents (comme `struct`), mais dont la taille ne varie pas (comme `std::array`).



Fichier à inclure

Pour utiliser les tuples, il faut inclure le fichier `<tuple>`.

III.7.2.1. Déclaration

Pour préciser les types des différents éléments, on utilise une syntaxe bien connue. Par exemple, si on veut créer un tuple pour contenir des informations personnelles, comme dans l'exercice précédent, c'est très simple.

```
1 using namespace std::literals;  
2 std::tuple { "Clem"s, "Lagrume"s, "Fruit"s, 4 };
```



N'oubliez pas le `s`

Je vous rappelle qu'avec les littéraux, pour que le type déduit soit bien `std::string`, vous devez rajouter le `s` après chaque littéral (ainsi que la ligne `using namespace std::literals`). C'est ici obligatoire pour que l'on crée un `std::tuple` contenant des `std::string` et non des chaînes héritées du C.

On a aussi une fonction `std::make_tuple` permettant de créer un tuple en déduisant les types.

III. On passe la deuxième!

```
1 using namespace std::literals;
2 auto infos = std::make_tuple("Clem"s, "Lagrume"s, "Fruit"s, 4);
```

III.7.2.2. Accès aux éléments

Comme pour toute collection, il faut aussi être capable d'accéder aux éléments. Pour cela, on utilise la fonction `std::get`. Celle-ci prend entre chevrons l'indice de l'élément auquel on veut accéder.

```
1 #include <iostream>
2 #include <string>
3 #include <tuple>
4
5 int main()
6 {
7     using namespace std::literals;
8     auto informations = std::make_tuple("Clem"s, "Lagrume"s,
9     "Fruit"s, 4);
10
11     // Accès en lecture.
12     std::cout << "Prénom : " << std::get<0>(informations) <<
13     std::endl;
14     std::cout << "Âge : " << std::get<3>(informations) <<
15     std::endl;
16
17     // Accès en écriture.
18     std::get<3>(informations) += 1; // Clem a maintenant 5 ans !
19     std::cout << "Âge maintenant : " << std::get<3>(informations)
20     << std::endl;
21
22     return 0;
23 }
```

```
1 Prénom : Clem
2 Âge : 4
3 Âge maintenant : 5
```

Dans le cas où tous les éléments d'un `std::tuple` ont un type unique, et **seulement dans ce cas**, on peut aussi y accéder en donnant à `std::get` le type de l'élément désiré.

III. On passe la deuxième!

```
1 #include <iostream>
2 #include <string>
3 #include <tuple>
4
5 int main()
6 {
7     using namespace std::literals;
8     // Prénom et salaire.
9     auto informations = std::make_tuple("Clem"s, 50000);
10
11     std::cout << "Prénom : " << std::get<std::string>(informations)
12               << std::endl;
13     std::cout << "Salaire : " << std::get<int>(informations) << "€"
14               << std::endl;
15
16     return 0;
17 }
```

```
1 Prénom : Clem
2 Salaire : 50000€
```

Si vous essayez d'utiliser cette syntaxe alors qu'il y a plusieurs éléments de même type, le code ne compilera tout simplement pas.

```
1 #include <iostream>
2 #include <string>
3 #include <tuple>
4
5 int main()
6 {
7     using namespace std::literals;
8     auto identite = std::make_tuple("Clem"s, "Lagrume"s);
9     // Ne compile pas car il y a deux éléments de même type.
10    // Comment choisir ?
11    std::get<std::string>(identite);
12
13    return 0;
14 }
```

III.7.2.3. `std::tuple` et fonctions

On peut tout à fait créer une fonction qui renvoie un `std::tuple`. C'est même une astuce qu'on utilise pour contourner la limite qu'impose C++ de ne pouvoir retourner qu'une seule

III. On passe la deuxième!

valeur à la fois. Il suffit de déclarer que la fonction renvoie un `std::tuple` et de préciser, entre chevrons, les types de ses éléments.

Notez, dans l'exemple suivant, la syntaxe alternative et concise pour renvoyer un `std::tuple`, introduite depuis C++17.

```
1 // Ligne nécessaire pour Visual Studio afin d'accéder à M_PI, la
   valeur de pi.
2 #define _USE_MATH_DEFINES
3
4 #include <cmath>
5 #include <iostream>
6 #include <string>
7 #include <tuple>
8
9 std::tuple<int, std::string> f()
10 {
11     using namespace std::literals;
12     // On retourne un std::tuple qui contient un entier et une
       chaîne de caractères.
13     return std::make_tuple(20, "Clem"s);
14 }
15
16 std::tuple<double, double> g(double angle)
17 {
18     // On retourne un std::tuple qui contient deux flottants.
19     // Notez la nouvelle syntaxe à base d'accolades.
20     return { std::cos(angle), std::sin(angle) };
21 }
22
23 int main()
24 {
25     std::tuple resultats_scolaires = f();
26     std::cout << "Tu t'appelles " <<
       std::get<std::string>(resultats_scolaires) <<
       " et tu as obtenu " << std::get<int>(resultats_scolaires)
       << " / 20." << std::endl;
27
28     std::tuple calculs = g(M_PI / 2.);
29     std::cout << "Voici le cosinus de PI / 2 : " <<
       std::get<0>(calculs) << std::endl;
30     std::cout << "Voici le sinus de PI / 2 : " <<
       std::get<1>(calculs) << std::endl;
31
32     return 0;
33 }
```

III. On passe la deuxième!

```
1 Tu t'appelles Clem et tu as obtenu 20 / 20.
2 Voici le cosinus de PI / 2 : 0
3 Voici le sinus de PI / 2 : 1
```

III.7.2.4. Équations horaires

Vous voulez un exemple concret d'utilisation des `std::tuple`? Vous aimez la physique? Vous allez être servi. Nous allons coder une fonction retournant les coordonnées $(x; y)$ d'un objet lancé à une vitesse v_0 et à un angle α , en fonction du temps t .



Point de vocabulaire

C'est ce qu'on appelle l'**équation horaire** de l'objet. Cette notion est abordée, en France, au lycée en Terminale. Ne vous inquiétez pas si vous ne connaissez pas, nous n'allons pas du tout rentrer dans les détails.

Un [cours de physique](#) vous expliquera qu'on obtient la position de l'objet en fonction du temps t grâce à deux équations.

$$\begin{cases} x(t) = v_0 \cdot \cos \alpha \cdot t \\ y(t) = -g \cdot \frac{t^2}{2} + v_0 \cdot \sin \alpha \cdot t \end{cases}$$

Donc, si on traduit tout ça en C++, on obtient quelque chose comme ça.

```
1 // Ligne nécessaire pour Visual Studio afin d'accéder à M_PI, la
   valeur de pi.
2 #define _USE_MATH_DEFINES
3
4 #include <cmath>
5 #include <iostream>
6 #include <string>
7 #include <tuple>
8
9 std::tuple<double, double> equations_mouvement_temps(double
   vitesse_initiale, double angle, double t)
10 {
11     // Accélération sur Terre.
12     double const g { 9.80665 };
13     double const x { vitesse_initiale * std::cos(angle) * t };
14     double const y { (-g * (t * t) / 2.) + (vitesse_initiale *
   std::sin(angle) * t) };
15
16     return { x, y };
```

III. On passe la deuxième!

```
17 }
18
19 int main()
20 {
21     // Vitesse de 8 mètres par seconde.
22     double const vitesse_initiale { 8.0 };
23     // On tire à un angle de pi / 4, soit 45°.
24     double const angle { M_PI / 4 };
25
26     // On simule sur 10 secondes, avec un calcul toutes les 100
27     // millisecondes.
28     for (double t { 0.0 }; t < 10; t += 0.1)
29     {
30         std::tuple coordonnees {
31             equations_mouvement_temps(vitesse_initiale, angle, t)
32         };
33
34         if (std::get<1>(coordonnees) < 0)
35         {
36             // On ne continue plus la simulation dès que y est au
37             // niveau du sol.
38             break;
39         }
40
41         std::cout << "À l'instant t = " << t << ", on a : " <<
42             std::endl;
43         std::cout << "x = " << std::get<0>(coordonnees) <<
44             std::endl;
45         std::cout << "y = " << std::get<1>(coordonnees) <<
46             std::endl << std::endl;
47     }
48
49     return 0;
50 }
```



Pourquoi s'encombrer des tuples? Que dois-je choisir?

Il est vrai que `std::tuple` semble présenter plus d'inconvénients qu'autres choses. Le code n'est pas très lisible avec tous ces `std::get`. De plus, contrairement à une structure dont l'accès à un élément est clair (comme `informations.prenom`), il ne saute pas aux yeux que `std::get<1>(informations)` représente potentiellement un prénom.

En fait, chaque solution a ses avantages et ses inconvénients, et on va donc choisir en conséquence selon les situations.

- Le tuple permet de rapidement avoir une collection hétérogène **sans avoir à définir un nouveau type**. Il est particulièrement utile lorsque l'on a besoin d'une collection hétérogène **uniquement pour une portée locale**, dans le corps d'une fonction par

III. On passe la deuxième!

exemple. Par contre, la syntaxe d'utilisation, notamment pour l'accès aux éléments, est peu lisible.

- La structure permet une **plus grande expressivité**, notamment par le fait que les éléments ont des noms. Elle est particulièrement adaptée pour une utilisation sur une portée plus globale.

En fait, le tuple est à la structure ce que la lambda est à la fonction, finalement. C++ vous laisse libre, c'est vous qui choisirez en fonction des cas.

III.7.3. `std::unordered_map`— Une table associative

Depuis le début de ce cours, nous avons vu les conteneurs `std::vector`, `std::array` et `std::string`. Nous avons également, très brièvement, survolé `std::list`. Maintenant, j'aimerais vous en présenter d'autres, qui sont un peu différents, mais très utiles.

Le premier conteneur se nomme `std::unordered_map`. Ce nom peut être traduit par «tableau associatif», «table associative» ou «carte associative», car son principe est **d'associer une valeur à une clé**. Ce principe, nous le connaissons déjà avec `std::vector`, qui associe une valeur quelconque à une clé entière. Ainsi, quand vous faites `tableau[4]`, vous utilisez une clé de type entier et qui vaut 4. Grâce à `std::unordered_map`, nous avons le choix du type de la clé.



Fichier à inclure

Pour utiliser ce nouveau conteneur, pensez à inclure le fichier `<unordered_map>`.

III.7.3.1. Un dictionnaire de langue Zestienne

Imaginez que vous vouliez créer un dictionnaire. Comme pour la version papier, nous voulons associer un mot à un ensemble de phrases. En C++, on peut traduire ça par une `std::unordered_map` dont la clé est de type `std::string`, ainsi que la valeur. On précise le type de la clé et de la valeur entre chevrons, car notre conteneur utilise les templates.

```
1 #include <iostream>
2 #include <string>
3 #include <unordered_map>
4
5 int main()
6 {
7     using namespace std::literals;
8
9     // Notez que chaque élément ajouté est lui-même entre accolades
10    {}.
11    std::unordered_map<std::string, std::string> dictionnaire
12    {
```

III. On passe la deuxième!

```
12      // Ajout d'une clé 'Clem' avec sa valeur 'La mascotte du
13      site Zeste de Savoir, toute gentille et tout mignonne.'
14      { "Clem"s,
15        "La mascotte du site Zeste de Savoir, toute gentille et toute mignonne",
16      },
17      { "mehdidou99"s, "Un des auteurs du tutoriel C++."s },
18      { "informaticienzero"s, "Un des auteurs du tutoriel C++."s
19      },
20      { "Taurre"s, "Un super valideur !"s },
21      { "Arius"s, "Un membre super sympa mais qui mord."s },
22      { "Gants de vaisselle"s,
23        "Objets présents sur le site et dont personne ne sait pourquoi."s
24      }
25  };
26
27  return 0;
28 }
```

Pour afficher le contenu de notre conteneur, on va récupérer les éléments un par un. On peut faire ça avec une simple boucle `for`, comme on en a l'habitude. Chaque élément est un `std::pair<T, U>`, où `T` est de même type que la clé et `U` de même type que la valeur. Ce type n'est rien d'autre qu'un cas particulier de tuple à deux valeurs uniquement.

Ce type fournit deux propriétés qui nous intéressent, qui sont `first` et `second`, renvoyant respectivement la clé et la valeur récupérée dans la `std::unordered_map`. On a donc maintenant toutes les informations pour écrire la boucle et afficher les éléments de notre tableau associatif.

```
1  #include <iostream>
2  #include <string>
3  #include <unordered_map>
4
5  int main()
6  {
7      using namespace std::literals;
8
9      std::unordered_map<std::string, std::string> dictionnaire
10     {
11         { "Clem"s,
12           "La mascotte du site Zeste de Savoir, toute gentille et toute mignonne",
13         },
14         { "mehdidou99"s, "Un des auteurs du tutoriel C++."s },
15         { "informaticienzero"s, "Un des auteurs du tutoriel C++."s
16         },
17         { "Taurre"s, "Un super valideur !"s },
18         { "Arius"s, "Un membre super sympa mais qui mord."s },
19         { "Gants de vaisselle"s,
20           "Objets présents sur le site et dont personne ne sait pourquoi."s
21         }
22     }
23 }
```

III. On passe la deuxième!

```
17     };
18
19     for (auto const & paire : dictionnaire)
20     {
21         std::cout << "Clé : " << paire.first << std::endl;
22         std::cout << "Valeur : " << paire.second << std::endl <<
            std::endl;
23     }
24
25     return 0;
26 }
```

```
1 Clé : mehdidou99
2 Valeur : Un des auteurs du tutoriel C++.
3
4 Clé : Clem
5 Valeur : La mascotte du site Zeste de Savoir, toute gentille et
    toute mignonne.
6
7 Clé : informaticienzero
8 Valeur : Un des auteurs du tutoriel C++.
9
10 Clé : Taurre
11 Valeur : Un super valideur !
12
13 Clé : Arius
14 Valeur : Un membre super sympa mais qui mord.
15
16 Clé : Gants de vaisselle
17 Valeur : Objets présents sur le site et dont personne ne sait
    pourquoi.
```



Petite précision

Je vous ai expliqué ce qu'est `std::pair`, mais j'ai utilisé `auto` à la place. Le code est ainsi plus court et plus lisible. Vous êtes néanmoins libre d'écrire explicitement `std::pair<std::string, std::string>` si vous le souhaitez.

III.7.3.2. Toujours plus de clés

Bien entendu, si notre conteneur n'est pas déclaré comme `const`, il est possible de le modifier, lui ajouter ou lui supprimer des éléments. Voyez par vous-mêmes plusieurs exemples ci-dessous.

III. On passe la deuxième!

```
1 #include <iostream>
2 #include <string>
3 #include <unordered_map>
4
5 int main()
6 {
7     using namespace std::literals;
8
9     std::unordered_map<std::string, std::string> dictionnaire
10    {
11        { "Clem"s,
12            "La mascotte du site Zeste de Savoir, toute gentille et toute mign
13            },
14        { "mehdidou99"s, "Un des auteurs du tutoriel C++."s },
15        { "informaticienzero"s, "Un des auteurs du tutoriel C++."s
16            },
17        { "Taurre"s, "Un super validateur !"s },
18        { "Arius"s, "Un membre super sympa mais qui mord."s },
19        { "Gants de vaisselle"s,
20            "Objets présents sur le site et dont personne ne sait pourquoi."s
21            }
22    };
23
24    // Insertion avec la fonction 'insert'.
25    dictionnaire.insert({ "C++"s, "Le langage que vous apprenez."s
26        });
27
28    // Plusieurs valeurs en même temps. Notez les doubles accolades
29    {}.
30    dictionnaire.insert({ { "Pacman++"s,
31        "C'est un des avatars de mehdidou99."s }, { "Un truc"s,
32        "Personne ne sait ce que c'est."s } });
33
34    // Ajout d'une nouvelle paire avec les crochets.
35    dictionnaire["nohar"] = "Un lama ancestral."s;
36
37    // Utilisation des crochets pour modifier la valeur associée à
38    une clé existante.
39    dictionnaire["Taurre"] = "Un validateur vraiment cool !"s;
40    // Accès en lecture.
41    std::cout << "Qui est Taurre ? " << dictionnaire["Taurre"] <<
42        std::endl << std::endl;
43
44    // Suppression par clé.
45    dictionnaire.erase("Un truc"s);
46
47    for (auto const & paire : dictionnaire)
48    {
49        std::cout << "Clé : " << paire.first << std::endl;
```

III. On passe la deuxième!

```
39         std::cout << "Valeur : " << paire.second << std::endl <<
           std::endl;
40     }
41
42     return 0;
43 }
```

```
1  Qui est Taurre ? Un validateur vraiment cool !
2
3  Clé : mehdidou99
4  Valeur : Un des auteurs du tutoriel C++.
5
6  Clé : C++
7  Valeur : Le langage que vous apprenez.
8
9  Clé : Clem
10 Valeur : La mascotte du site Zeste de Savoir, toute gentille et
    toute mignonne.
11
12 Clé : informaticienzero
13 Valeur : Un des auteurs du tutoriel C++.
14
15 Clé : Taurre
16 Valeur : Un validateur vraiment cool !
17
18 Clé : Arius
19 Valeur : Un membre super sympa mais qui mord.
20
21 Clé : Pacman++
22 Valeur : C'est un des avatars de mehdidou99.
23
24 Clé : Gants de vaisselle
25 Valeur : Objets présents sur le site et dont personne ne sait
    pourquoi.
26
27 Clé : nohar
28 Valeur : Un lama ancestral.
```

III.7.3.3. Cette clé est unique !

Abordons maintenant un point important de `std::unordered_map`, mais que j'ai passé sous silence. Il s'agit de **l'unicité de la clé**. Il ne peut y avoir qu'une seule valeur associée à une seule clé. Dit autrement, on ne peut pas assigner plusieurs valeurs à une même clé.

Quand on ajoute une nouvelle paire, il ne se passe rien de visible dans le cas où la clé existe déjà. C'est un peu gênant pour signaler l'erreur. Heureusement, la fonction `insert` retourne un

III. On passe la deuxième!

`std::pair` contenant deux informations qui nous intéressent.

- L'élément `first` est un itérateur sur l'élément inséré si la clé n'existait pas au préalable, ou sur l'élément correspondant à la clé si elle existe déjà. Cet itérateur est lui-même un `std::pair`, donc on accède à la clé avec `first` et à la valeur avec `second`.
- L'élément `second` est un booléen indiquant si l'élément a été inséré (`true`) ou non (`false`).

```
1 #include <iostream>
2 #include <string>
3 #include <unordered_map>
4
5 int main()
6 {
7     using namespace std::literals;
8
9     std::unordered_map<std::string, double> constantes
10    {
11        { "pi"s, 3.14159 },
12        { "e"s, 2.71828 }
13    };
14
15    auto paire_phi = constantes.insert({ "phi"s, 1.61803 });
16    if (paire_phi.second)
17    {
18        std::cout << "La valeur de " << paire_phi.first->first <<
19            " a bien été ajoutée." << std::endl;
20    }
21
22    auto paire_pi = constantes.insert({ "pi"s, 3.0 });
23    if (!paire_pi.second)
24    {
25        std::cout << "La clé " << paire_pi.first->first <<
26            " existe déjà et vaut " << paire_pi.first->second <<
27            "." << std::endl;
28    }
29
30    return 0;
31 }
```

```
1 La valeur de phi a bien été ajoutée.
2 La clé pi existe déjà et vaut 3.14159.
```

Dans le cas où vous souhaitez ajouter une valeur si elle n'existe pas, ou bien mettre à jour la valeur associée à une clé si celle-ci existe, utilisez la fonction `insert_or_assign`, qui fonctionne presque pareil. Elle renvoie une paire semblable, avec toujours le booléen signalant l'insertion (`true`) ou la mise à jour (`false`).

III. On passe la deuxième!

```
1 #include <iostream>
2 #include <string>
3 #include <unordered_map>
4
5 int main()
6 {
7     using namespace std::literals;
8
9     std::unordered_map<std::string, double> constantes
10    {
11        { "pi"s, 3.14159 },
12        { "e"s, 2.0 }
13    };
14
15    auto paire_e = constantes.insert_or_assign("e"s, 2.71828);
16    if (paire_e.second)
17    {
18        std::cout << "La valeur a bien été ajoutée." << std::endl;
19    }
20    else
21    {
22        std::cout << "La valeur a bien été mise à jour." <<
23            std::endl;
24    }
25
26    for (auto const & paire : constantes)
27    {
28        std::cout << "La constante " << paire.first << " vaut " <<
29            paire.second << "." << std::endl;
30    }
31
32    return 0;
33 }
```

```
1 La valeur a bien été mise à jour.
2 La constante pi vaut 3.14159.
3 La constante e vaut 2.71828.
```

III.7.3.4. Cherchez un élément

Comme l'accès à un élément par les crochets permet également d'ajouter une nouvelle paire clé / valeur, l'utiliser peut parfois être délicat. Voyez par vous-mêmes l'exemple suivant.

III. On passe la deuxième!

```
1 #include <iostream>
2 #include <string>
3 #include <unordered_map>
4
5 int main()
6 {
7     using namespace std::literals;
8
9     std::unordered_map<std::string, std::string> membres
10    {
11        { "mehdidou99"s, "Un auteur du cours C++."s },
12        { "informaticienzero"s, "Un auteur du cours C++."s }
13    };
14
15    // Notez l'accent.
16    std::string const pseudo { "informaticienzéro" };
17    // ah, ça ne va pas afficher ce que vous attendiez.
18    std::cout << "Qui est informaticienzero ? " << membres[pseudo]
19              << std::endl;
20
21    return 0;
22 }
```

```
1 Qui est informaticienzero ?
```

Il existe heureusement une fonction `find`, qui attend une clé et qui renvoie un itérateur sur l'élément concerné si la clé existe, ou bien un itérateur sur la fin de la collection sinon.

```
1 #include <iostream>
2 #include <string>
3 #include <unordered_map>
4
5 int main()
6 {
7     using namespace std::literals;
8
9     std::unordered_map<std::string, std::string> membres
10    {
11        { "mehdidou99"s, "Un auteur du cours C++."s },
12        { "informaticienzero"s, "Un auteur du cours C++."s }
13    };
14
15    // Notez l'accent.
16    std::string const pseudo { "informaticienzéro" };
17    if (membres.find(pseudo) != std::end(membres))
```


III. On passe la deuxième!

```
18     {
19         // On a trouvé le membre.
20         std::cout << "Qui est informaticienzero ? " <<
            membres[pseudo] << std::endl;
21     }
22     else
23     {
24         std::cout << "Ce membre n'existe pas." << std::endl;
25     }
26
27     return 0;
28 }
```

```
1 Ce membre n'existe pas.
```

III.7.3.5. Un exemple concret

Grâce à ce que nous savons, nous pouvons faire un petit programme qui compte les occurrences de chaque mot au sein d'une phrase. Il suffit d'avoir un tableau associatif avec `std::string` pour clé et `int` pour valeur.

```
1 #include <iostream>
2 #include <sstream>
3 #include <string>
4 #include <unordered_map>
5
6 void occurrences_mots(std::string const & phrase)
7 {
8     std::istringstream iss { phrase };
9     std::string mot {};
10
11     std::unordered_map<std::string, int> occurrences {};
12     while (iss >> mot)
13     {
14         ++occurrences[mot];
15     }
16
17     for (auto const & paire : occurrences)
18     {
19         std::cout << "Le mot '" << paire.first << "' apparait " <<
            paire.second << " fois." << std::endl;
20     }
21 }
22
```

III. On passe la deuxième!

```
23 int main()
24 {
25     occurrences_mots("Bonjour, ceci est un test avec une phrase qui est lon
26     return 0;
27 }
```

```
1 Le mot 'Bonjour,' apparait 1 fois.
2 Le mot 'ceci' apparait 1 fois.
3 Le mot 'est' apparait 3 fois.
4 Le mot 'un' apparait 1 fois.
5 Le mot 'une' apparait 2 fois.
6 Le mot 'test' apparait 1 fois.
7 Le mot 'qui' apparait 1 fois.
8 Le mot 'avec' apparait 1 fois.
9 Le mot 'français.' apparait 1 fois.
10 Le mot 'phrase' apparait 2 fois.
11 Le mot 'longue' apparait 1 fois.
12 Le mot 'et' apparait 1 fois.
13 Le mot 'en' apparait 1 fois.
```

III.7.4. `std::unordered_set`— Représenter un ensemble

Le deuxième conteneur se nomme `std::unordered_set`. C'est la représentation en C++ d'un ensemble au sens mathématique du terme, c'est-à-dire un ensemble d'éléments du même type, où chacun n'est présent qu'une seule et unique fois. C'est un peu comme une table associative dans laquelle il n'y aurait que des clés et pas de valeur.



Fichier à inclure

Pour utiliser ce conteneur, il faut inclure le fichier d'en-tête `<unordered_set>`.

III.7.4.1. Inscription sur Zeste de Savoir

Sur le présent site, vous pouvez vous inscrire, mais il faut que votre pseudo soit unique. Vous ne pouvez donc pas réutiliser le pseudo de quelqu'un d'autre. On pourrait donc imaginer stocker les pseudos dans un `std::unordered_set`. Vous allez voir que l'utilisation de ce conteneur ressemble fortement à celle de `std::unordered_map`.

III. On passe la deuxième!

```
1 #include <iomanip>
2 #include <iostream>
3 #include <string>
4 #include <unordered_set>
5
6 void ajouter_membre(std::unordered_set<std::string> & membres,
7                     std::string const & pseudo)
8 {
9     auto it = membres.insert(pseudo);
10    if (it.second)
11    {
12        std::cout << "Bienvenu sur Zeste de Savoir, " << pseudo <<
13            "." << std::endl;
14    }
15    else
16    {
17        // std::quoted permet d'ajouter automatiquement des
18        // guillemets doubles autour de la valeur qu'on lui passe.
19        std::cout << "Désolé, le pseudo " << std::quoted(pseudo) <<
20            " est déjà pris." << std::endl;
21    }
22 }
23
24 int main()
25 {
26     std::unordered_set<std::string> pseudos { "mehdidou99",
27         "informaticienzero" };
28
29     // Ajouts.
30     pseudos.insert("Dwayne");
31     pseudos.insert("artragis");
32     pseudos.insert("Taurre");
33
34     ajouter_membre(pseudos, "gbdivers");
35     ajouter_membre(pseudos, "informaticienzero");
36
37     // Suppression.
38     pseudos.erase("Dwayne");
39
40     // Parcours for.
41     for (auto const & cle : pseudos)
42     {
43         std::cout << cle << std::endl;
44     }
45
46     // Recherche de membres.
47     if (pseudos.find("Taurre") != std::end(pseudos))
48     {
49         std::cout << "Taurre est un membre." << std::endl;
50     }
51 }
```

III. On passe la deuxième!

```
45     }
46
47     if (pseudos.find("Gabbro") == std::end(pseudos))
48     {
49         std::cout << "Le pseudo " << std::quoted("Gabbro") <<
50             " est libre." << std::endl;
51     }
52     return 0;
53 }
```

Comme vous le voyez, l'utilisation est très proche. La différence se trouve simplement dans l'absence de valeurs liées aux clés. Pour choisir entre les deux conteneurs, il faut simplement **réfléchir à ce que vous voulez stocker**. Un ensemble de clés uniques? Choisissez `std::unordered_set`. Si vous devez associer à ces clés des valeurs, alors partez sur `std::unordered_map`.

III.7.5. Un peu d'ordre, voyons!

Ces deux conteneurs sont très utiles, mais comme vous l'avez remarqué, ils ne sont pas ordonnés. Si, par exemple, vous avez un `std::unordered_set` de chaînes de caractères, vous verrez qu'elles ne sont pas affichées dans l'ordre alphabétique.

```
1  #include <iostream>
2  #include <string>
3  #include <unordered_set>
4
5  int main()
6  {
7      std::unordered_set<std::string> const mots { "Archive", "Base",
8          "Conduite", "Développement", "Entrain" };
9
10     for (auto const & mot : mots)
11     {
12         std::cout << mot << std::endl;
13     }
14     return 0;
15 }
```

```
1  Développement
2  Archive
3  Entrain
4  Conduite
```

III. On passe la deuxième!

5 Base

En interne, les clés sont ordonnées et triées, mais selon **leur «hash»**. Non, je n'ai pas fait de faute de français. Un *hash* est le résultat d'une [fonction de hachage](#). En très bref, c'est une fonction qui transforme une donnée quelconque en une donnée de taille fixée. Pour notre conteneur, cela signifie qu'il ne va plus travailler avec des chaînes de caractères ou autres clés potentiellement grosses, mais avec **une version de taille fixe et plus petite**, ce qui est **beaucoup plus rapide**.

Il existe une version alternative des conteneurs `std::unordered_map` et `std::unordered_set`, qui n'ordonne pas les éléments selon le hash des clés mais **directement avec les clés**. Ces nouveaux conteneurs se nomment, respectivement, `std::map` et `std::set`.



Fichier à inclure

Pour les utiliser, il faut respectivement inclure les fichiers `<map>` et `<set>`.

En remplaçant `std::unordered_set` par sa version triée par clé dans l'exemple précédent, on obtient bien un affichage par ordre alphabétique.

```
1 #include <iostream>
2 #include <string>
3 #include <set>
4
5 int main()
6 {
7     std::set<std::string> const mots { "Archive", "Base",
8         "Conduite", "Développement", "Entrain" };
9
10    for (auto const & mot : mots)
11    {
12        std::cout << mot << std::endl;
13    }
14
15    return 0;
16 }
```

```
1 Archive
2 Base
3 Conduite
4 Développement
5 Entrain
```



Du coup, la seule différence, c'est une question d'affichage et d'ordre?

La structure interne et les algorithmes employés pour ajouter, supprimer ou chercher des éléments ne sont pas les mêmes. De manière générale, les versions non ordonnées sont plus rapides d'utilisation et ce sont celles qui sont utilisées dans 90% des cas. Les 10% restants se répartissent en deux points.

- On veut **conserver l'ordre des clés**.
- On a vraiment **énormément de données** (de l'ordre du million d'éléments), auquel cas la version non ordonnée est moins rapide que la version ordonnée, de part leur structure interne.

Leur utilisation est très semblable à leurs cousins non ordonnés, vous n'aurez donc aucun de mal à les manipuler. 😊

III.7.6. Une longue énumération

Terminons ce chapitre avec les énumérations. Leur but est de permettre une **liste de valeurs fortement typées et fixe**. Fortement typées car les valeurs sont d'un type bien précis et défini. Fixe car le nombre d'éléments est fixé et immuable.

On s'en sert typiquement pour **limiter le champs des possibilités** pour un type. Ainsi, au lieu de stocker le jour de la semaine sous forme de `std::string`, ce qui n'offre pas de protection si l'utilisateur rentre n'importe quoi, on va utiliser une énumération. Ainsi, celui-ci n'a pas d'autres choix que d'utiliser une des valeurs que nous avons nous-mêmes définis. C'est beaucoup plus souple que de devoir faire plein de vérifications sur une chaîne de caractères.

Pour déclarer une énumération, c'est très simple.

```
1 enum class Nom_enumeration
2 {
3     Valeur1,
4     Valeur2
5     // Etc.
6 };
```

Concrètement appliqué à notre exemple des jours de la semaine, on obtient le code suivant.

```
1 enum class Jour
2 {
3     Lundi,
4     Mardi,
5     Mercredi,
6     Jeudi,
```

III. On passe la deuxième!

```
7     Vendredi,  
8     Samedi,  
9     Dimanche  
10  };
```

Pour utiliser l'une des valeurs de l'énumération, c'est aussi très simple. On fait `Nom_enumération::Valeur` (notez les deux-points `::`, comme dans `std::`). Examinez le code suivant qui montre un cas d'utilisation.

```
1  #include <iostream>  
2  
3  enum class Jour  
4  {  
5      Lundi,  
6      Mardi,  
7      Mercredi,  
8      Jeudi,  
9      Vendredi,  
10     Samedi,  
11     Dimanche  
12 };  
13  
14 void message(Jour jour_semaine)  
15 {  
16     if (jour_semaine == Jour::Samedi || jour_semaine ==  
17         Jour::Dimanche)  
18     {  
19         std::cout << "Bon week-end !" << std::endl;  
20     }  
21     else  
22     {  
23         std::cout << "Courage !" << std::endl;  
24     }  
25 }  
26  
27 int main()  
28 {  
29     Jour const aujourd'hui { Jour::Mardi };  
30     message(aujourd'hui);  
31     return 0;  
32 }
```

Ainsi, si vous avez une liste de choix de valeurs qui est limitée, **une énumération peut être la bonne solution**. Les mois de l'année, les couleurs d'un jeu de carte, les directions d'une boussole sont de bons cas d'utilisation. En plus de moins se casser la tête pour la vérification des valeurs, vous créez un nouveau type, clairement défini; en effet, `JourSemaine` est bien plus parlant qu'une `std::string`. Le code est rendu **plus clair** et c'est une très bonne pratique.





Passage par copie

Une énumération est très légère car elle utilise en interne des entiers. C'est pour cette raison qu'il est inutile de la passer par référence.

III.7.6.1. En résumé

- Les structures sont des agrégats de données permettant de lier et de manipuler plus aisément des données liées entre elles.
- Les tuples sont des collections hétérogènes et fixées, qui permettent de lier des données entre elles sans créer pour autant de nouveau type.
- Grâce aux tuples et aux structures, nous sommes en mesure de retourner plusieurs variables d'un coup.
- Les `std::unordered_map` sont des tables associatives, qui permettent d'associer une clé de type quelconque à une valeur de type quelconque.
- Les `std::unordered_set` sont des ensembles au sens mathématique du terme et permettent de stocker un ensemble de clés uniques.
- Il existe des versions triées par clé de ces conteneurs, `std::map` et `std::set`, mais qui sont beaucoup plus lentes.
- Une énumération permet de définir un type avec un ensemble de valeurs fixe et clairement défini.

Contenu masqué

Contenu masqué n°43 : Correction

```
1 #include <fstream>
2 #include <iostream>
3 #include <limits>
4 #include <string>
5
6 template<typename T>
7 void entree_securisee(T & variable)
8 {
9     while (!(std::cin >> variable))
10     {
11         std::cout << "Entrée invalide. Recommence." << std::endl;
12         std::cin.clear();
13
14         std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
15             '\n');
```


III. On passe la deuxième!

```
14     }
15 }
16
17 template <typename T, typename Predicat>
18 void entree_securisee(T & variable, Predicat predicat)
19 {
20     while (!(std::cin >> variable) || !predicat(variable))
21     {
22         std::cout << "Entrée invalide. Recommence." << std::endl;
23         std::cin.clear();
24
25         std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
26             '\n');
27     }
28 }
29
30 void demander_infos(std::string & nom, std::string & prenom,
31     std::string & sexe, int & age)
32 {
33     std::cout << "Nom ? ";
34     entree_securisee(nom);
35     std::cout << "Prénom ? ";
36     entree_securisee(prenom);
37     std::cout << "Sexe ? ";
38     entree_securisee(sexe);
39     std::cout << "Age ? ";
40     entree_securisee(age, [](int & age) { return age >= 0; });
41 }
42
43 std::string enregistrer_infos(std::string const & nom, std::string
44     const & prenom, std::string const & sexe, int age)
45 {
46     std::string nom_fichier { prenom + "." + nom + ".csv" };
47     std::ofstream fichier { nom_fichier };
48     fichier << nom << ',' << prenom << ',' << sexe << ',' << age;
49
50     return nom_fichier;
51 }
52
53 int main()
54 {
55     std::string nom { "" };
56     std::string prenom { "" };
57     std::string sexe { "" };
58     int age { 0 };
59
60     demander_infos(nom, prenom, sexe, age);
61     auto nom_fichier = enregistrer_infos(nom, prenom, sexe, age);
```

III. On passe la deuxième!

```
58     std::cout << std::endl <<
        "Ces données seront enregistrées dans le fichier " <<
        nom_fichier << std::endl;
59
60     return 0;
61 }
```

[Retourner au texte.](#)

Contenu masqué n°44 : Correction

```
1  #include <fstream>
2  #include <iostream>
3  #include <limits>
4  #include <string>
5
6
7  template<typename T>
8  void entree_securisee(T & variable)
9  {
10     while (!(std::cin >> variable))
11     {
12         std::cout << "Entrée invalide. Recommence." << std::endl;
13         std::cin.clear();
14
15         std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
16                         '\n');
17     }
18 }
19
20 template <typename T, typename Predicat>
21 void entree_securisee(T & variable, Predicat predicat)
22 {
23     while (!(std::cin >> variable) || !predicat(variable))
24     {
25         std::cout << "Entrée invalide. Recommence." << std::endl;
26         std::cin.clear();
27
28         std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
29                         '\n');
30     }
31 }
32
33 struct InformationsPersonnelles
34 {
```

III. On passe la deuxième!

```
31     std::string prenom;
32     std::string nom;
33     std::string sexe;
34     int age;
35 };
36
37 InformationsPersonnelles demander_infos()
38 {
39     InformationsPersonnelles infos {};
40     std::cout << "Nom ? ";
41     entree_securisee(infos.nom);
42     std::cout << "Prénom ? ";
43     entree_securisee(infos.prenom);
44     std::cout << "Sexe ? ";
45     entree_securisee(infos.sexe);
46     std::cout << "Age ? ";
47     entree_securisee(infos.age, [](int & age) { return age >= 0;
48         });
49     return infos;
50 }
51
52 std::string enregistrer_infos(InformationsPersonnelles const &
53     infos)
54 {
55     std::string nom_fichier { infos.prenom + "." + infos.nom +
56         ".csv" };
57     std::ofstream fichier { nom_fichier };
58
59     fichier << infos.nom << ',' << infos.prenom << ',' <<
60         infos.sexe << ',' << infos.age;
61
62     return nom_fichier;
63 }
64
65 int main()
66 {
67     auto infos = demander_infos();
68     auto nom_fichier = enregistrer_infos(infos);
69     std::cout << std::endl <<
70         "Ces données seront enregistrées dans le fichier " <<
71         nom_fichier << std::endl;
72
73     return 0;
74 }
```

[Retourner au texte.](#)

III.8. Reprendrez-vous un peu de sucre syntaxique ?

Le sucre se trouve partout aujourd'hui. Dans les gâteaux, dans les biscuits, dans les boissons et même dans notre code! Sauf que dans ce dernier cas, on parle de **sucre syntaxique**. Ce terme, inventé par [Peter J. Landin](#), un informaticien britannique, désigne des syntaxes plus lisibles ou plus faciles à écrire qu'on utilise pour se faciliter la vie.

Le but de ce chapitre va donc être d'**utiliser** et même d'**écrire du sucre syntaxique**.

III.8.1. Ce type est trop long !

Vous vous souvenez du chapitre sur les flux? Nous y avons découvert comment traiter des chaînes de caractères comme des flux, avec les types `std::istream` et `std::ostream`. Mais il faut avouer que c'est assez long à écrire et fatigant à la longue.

Heureusement, en C++, on peut définir des **alias de types**, des synonymes, qui permettent d'éviter l'utilisation de types longs à écrire ou bien de types composés. Cela se fait très simplement, en utilisant le mot-clef `using`.

```
1 // Désormais, écrire 'oss' dans le code revient à écrire
  'std::ostream'.
2 using oss = std::ostream;
3
4 // Désormais, écrire 'ListeResultats' dans le code revient à écrire
  'std::vector<std::tuple<std::string, int>>'.
5 using ListeResultats = std::vector<std::tuple<std::string, int>>;
```

Même la bibliothèque standard utilise le concept. Ainsi, `std::string` est lui-même l'alias d'un type `std::basic_string<char>`. Notez que c'est quand même beaucoup plus agréable à écrire.

III.8.2. Décomposons tout ça

Dans le chapitre précédent, nous avons souligné la lourdeur et le manque d'expressivité de l'accès aux éléments d'un tuple. Il est vrai qu'une expression comme `std::get<2>(tuple)` ne nous donne guère d'information sur le sens de ce que nous manipulons. Heureusement, il existe des syntaxes alternatives, que nous allons voir.

III.8.2.1. En pleine décomposition

Une des nouveautés de C++17, ce sont les décompositions (en anglais «*structured bindings*»). Ce concept est très simple. Son but est de déclarer et initialiser plusieurs variables d'un coup, depuis un tuple ou une structure. Et comme un code vaut mille mots, regardez par vous-mêmes. J'ai repris un exemple du chapitre précédent, en l'adaptant.

```
1  #define _USE_MATH_DEFINES
2
3  #include <cmath>
4  #include <iostream>
5  #include <string>
6  #include <tuple>
7
8  std::tuple<int, std::string> f()
9  {
10     using namespace std::literals;
11     return { 20, "Clem"s };
12 }
13
14 std::tuple<double, double> g(double angle)
15 {
16     return { std::cos(angle), std::sin(angle) };
17 }
18
19 int main()
20 {
21     // AVANT
22     // std::tuple resultats_scolaires = f();
23     // std::cout << "Tu t'appelles " <<
24         std::get<std::string>(resultats_scolaires) << " et tu as
25         obtenu " << std::get<int>(resultats_scolaires) << " / 20."
26         << std::endl;
27
28     // Maintenant, on sait qu'on a affaire à un nom et un entier.
29     auto[note, nom] = f();
30     std::cout << "Tu t'appelles " << nom << " et tu as obtenu " <<
31         note << " / 20." << std::endl;
32
33     // AVANT
34     // std::tuple calculs = g(M_PI / 2.);
35     // std::cout << "Voici le cosinus de PI / 2 : " <<
36         std::get<0>(calculs) << std::endl;
37     // std::cout << "Voici le sinus de PI / 2 : " <<
38         std::get<1>(calculs) << std::endl;
39
40     // Maintenant, on sait qu'on a affaire à deux valeurs
41     mathématiques.
42     auto[cosinus, sinus] = g(M_PI / 4.);
```

III. On passe la deuxième!

```
36     std::cout << "Voici le cosinus de PI / 4 : " << cosinus <<
      std::endl;
37     std::cout << "Voici le sinus de PI / 4 : " << sinus <<
      std::endl;
38
39     return 0;
40 }
```

Notez comment le code se voit simplifié et clarifié. Nous avons remplacé une ligne aussi peu claire que `std::get<0>(calculs)` par `cosinus`. D'un coup d'œil, vous voyez ce que fait la variable et ce que nous manipulons. Le code est plus lisible et donc de meilleure qualité. N'oubliez pas qu'un code est bien plus souvent lu qu'écrit.

On peut bien entendu utiliser des références, si l'on veut que des modifications sur les variables se répercutent sur le tuple lié. L'exemple suivant est tiré d'un [billet](#) de @germinolegrand.

```
1  #include <cassert>
2  #include <iostream>
3  #include <string>
4  #include <tuple>
5
6  int main()
7  {
8      using namespace std::literals;
9      auto bookmark = std::make_tuple("https://zestedesavoir.com"s,
      "Zeste de Savoir"s, 30);
10
11     auto&[url, titre, visites] = bookmark;
12     ++visites;
13
14     assert(std::get<2>(bookmark) == 31);
15     return 0;
16 }
```

Enfin, ce que nous venons de voir est tout à fait applicable aux structures. L'exemple suivant, bien que bidon, vous le montre.

```
1  #include <iostream>
2  #include <string>
3  #include <tuple>
4
5  struct Personne
6  {
7      std::string nom;
8      std::string prenom;
9      int age;
```

III. On passe la deuxième!

```
10 };
11
12 Personne f()
13 {
14     return { "Lagrume", "Clem", 4 };
15 }
16
17 int main()
18 {
19     auto[nom, prenom, age] = f();
20     std::cout << "Voici " << prenom << " " << nom << " et elle a "
21         << age << " ans." << std::endl;
22     return 0;
23 }
```

```
1 Voici Clem Lagrume et elle a 4 ans.
```

Une décomposition peut déclarer exactement autant de variables que d'éléments composant la structure ou le tuple, ni plus, ni moins. Même si nous ne voulons récupérer que certaines variables, il faut tout de même créer des variables inutiles qui ne seront plus utilisées par la suite.

III.8.2.2. Et avant, on faisait comment ?

Les décompositions sont apparues en C++17, mais avant, il existait une autre façon de faire, qui existe toujours soit dit en passant. Il s'agit de `std::tie`. Le principe est identique, puisque on veut assigner à des variables les différents éléments d'un tuple ou d'une structure. Sauf qu'ici, les variables doivent être déclarées au préalable.

```
1 #define _USE_MATH_DEFINES
2
3 #include <cmath>
4 #include <iostream>
5 #include <tuple>
6
7 std::tuple<double, double, double> f(double angle)
8 {
9     return { std::cos(angle), std::sin(angle), std::tan(angle) };
10 }
11
12 int main()
13 {
14     // Création au préalable.
15     double cosinus {};
```

III. On passe la deuxième!

```
16     double sinus {};  
17     double tangente {};  
18  
19     std::tie(cosinus, sinus, tangente) = f(M_PI);  
20  
21     std::cout << "Cosinus de pi : " << cosinus << std::endl;  
22     std::cout << "Sinus de pi : " << sinus << std::endl;  
23     std::cout << "Tangente de pi : " << tangente << std::endl;  
24  
25     return 0;  
26 }
```

```
1 Cosinus de pi : -1  
2 Sinus de pi : 0  
3 Tangente de pi : 0
```

Cette solution présente a comme inconvénient majeur que les variables doivent être déclarées à l'avance et donc qu'**on ne peut pas profiter de l'inférence de type**. Par contre, l'avantage qu'elle a sur la décomposition, c'est qu'elle permet **d'ignorer certaines valeurs** qui ne nous intéressent pas et que nous ne voulons pas récupérer, grâce à `std::ignore`.

```
1 #define _USE_MATH_DEFINES  
2  
3 #include <cmath>  
4 #include <iostream>  
5 #include <tuple>  
6  
7 std::tuple<double, double, double> f(double angle)  
8 {  
9     return { std::cos(angle), std::sin(angle), std::tan(angle) };  
10 }  
11  
12 int main()  
13 {  
14     // Seul le cosinus m'intéresse.  
15     double cosinus {};  
16  
17     std::tie(cosinus, std::ignore, std::ignore) = f(M_PI);  
18  
19     std::cout << "Cosinus de pi : " << cosinus << std::endl;  
20     return 0;  
21 }
```

Un autre avantage, c'est de pouvoir réutiliser une même variable plusieurs fois, alors que la décomposition C++17 ne nous le permet pas.

III.8.3. La surcharge d'opérateurs

Lorsque l'on crée un type, il arrive que l'on souhaite réaliser certaines opérations communes sur ce type, comme l'injection dans un flux (pour de l'affichage par exemple), la réalisation d'opérations arithmétiques (addition, soustraction, etc) ou encore d'opérations de comparaison.

III.8.3.1. Le problème...

Imaginons que nous voulions créer un type pour manipuler des fractions. On aimerait pouvoir les additionner, les soustraire, etc, en bref, faire des opérations dessus. Sauf qu'on ne peut pas se contenter de faire un simple `+` ou `-`, comme c'est le cas avec des `int` ou des `double`.

```
1 #include <iostream>
2 #include <string>
3
4 struct Fraction
5 {
6     int numérateur;
7     int dénominateur;
8 };
9
10 int main()
11 {
12     Fraction const un_demi { 1, 2 };
13     Fraction const trois_quarts { 3, 4 };
14     Fraction const resultat { un_demi + trois_quarts };
15
16     std::cout << "1/2 + 3/4 font " << resultat.numérateur << "/" <<
17         resultat.dénominateur << std::endl;
18     return 0;
19 }
```

```
1 [Visual Studio]
2 Erreur C2676 '+' binaire : 'const Fraction' ne définit pas
   cet opérateur ou une conversion vers un type acceptable pour
   l'opérateur prédéfini
3 Erreur (active) E0349 aucun opérateur "+" ne correspond à
   ces opérandes
4
5 -----
6
7 [GCC]
8 prog.cc: In function 'int main()':
9 prog.cc:14:39: error: no match for 'operator+' (operand types are
   'const Fraction' and 'const Fraction')
```

III. On passe la deuxième!

```
10      14 |      Fraction const resultat { un_demi + trois_quarts };
11          |                               ~~~~~^~~~~~
12
13 -----
14
15 [Clang]
16 prog.cc:14:39: error: invalid operands to binary expression ('const
    Fraction' and 'const Fraction')
17     Fraction const resultat { un_demi + trois_quarts };
18                               ~~~~~ ^ ~~~~~
19 1 error generated.
```

On doit alors écrire des fonctions pour faire ça. Mais si on les mélange, ça peut très vite devenir indigeste à lire.

```
1  #include <iostream>
2  #include <string>
3
4  struct Fraction
5  {
6      int numerateur;
7      int denominateur;
8  };
9
10 Fraction additionner(Fraction const & a, Fraction const & b)
11 {
12     int numerateur { a.numerateur * b.denominateur + b.numerateur *
        a.denominateur };
13     int denominateur { a.denominateur * b.denominateur };
14
15     return { numerateur, denominateur };
16 }
17
18 Fraction soustraire(Fraction const & a, Fraction const & b)
19 {
20     int numerateur { a.numerateur * b.denominateur - b.numerateur *
        a.denominateur };
21     int denominateur { a.denominateur * b.denominateur };
22
23     return { numerateur, denominateur };
24 }
25
26 int main()
27 {
28     Fraction const un_demi { 1, 2 };
29     Fraction const trois_quarts { 3, 4 };
30     Fraction const deux_huitièmes { 2, 8 };
31
```

III. On passe la deuxième!

```
32 // Arf, c'est dur à lire.  
33 Fraction const resultat { soustraire(additionner(un_demi,  
34     trois_quarts), deux_huitièmes) };  
35     std::cout << "1/2 + 3/4 - 2/8 font " << resultat.numerateur <<  
36     "/" << resultat.denominateur << std::endl;  
37 return 0;  
37 }
```

On peut également parler de l’affichage qui est galère et bien d’autres choses encore. Il faut donc trouver une solution à cet énorme problème de lisibilité. L’idéal, ça serait de pouvoir utiliser les opérateurs avec nos types personnels. Et heureusement, C++ ne nous laisse pas seuls face à ce problème!

III.8.3.2. ...et la solution

Dans le chapitre introduisant les fonctions, nous avons vu que l’on peut surcharger une fonction pour la faire fonctionner avec des paramètres de types différents. Eh bien, bonne nouvelle, **les opérateurs sont des fonctions comme les autres**.

Autrement dit, tous les opérateurs, aussi bien arithmétiques (+, *, etc), de comparaison (<, >, ==, etc), ou de flux (<, >), sont appelables et surchargeables. Simplement, ce sont des fonctions avec une syntaxe spéciale faite pour rendre le code plus lisible.

Ainsi, on va pouvoir surcharger nos opérateurs pour nos types personnels et ainsi éliminer le problème d’expressivité du code qu’on a soulevé précédemment.

III.8.3.3. Cas concret— Les fractions

III.9. Les opérateurs arithmétiques

Nous allons aborder ici les opérateurs `+`, `-`, `*` et `/`, qui nous permettront de réaliser, respectivement, des additions, des soustractions, des multiplications et des divisions de fractions. Voyons déjà à quoi ressemblent, de manière générale, les opérateurs en question.

```
1 T operator+(T const & lhs, T const & rhs);
2 T operator-(T const & lhs, T const & rhs);
3 T operator*(T const & lhs, T const & rhs);
4 T operator/(T const & lhs, T const & rhs);
```

Dans notre cas, ils se traduisent ainsi. Notez que les prototypes de `+` et de `-` ressemblent beaucoup aux fonctions `additionner` et `soustraire` que nous avons écrites.

```
1 #include <iostream>
2 #include <string>
3
4 struct Fraction
5 {
6     int numérateur;
7     int dénominateur;
8 };
9
10 // Ça ressemble à quelque chose de connu, hein ?
11 Fraction operator+(Fraction const & a, Fraction const & b)
12 {
13     int numérateur { a.numérateur * b.dénominateur + b.numérateur *
14                     a.dénominateur };
15     int dénominateur { a.dénominateur * b.dénominateur };
16     return { numérateur, dénominateur };
17 }
18
19 Fraction operator-(Fraction const & a, Fraction const & b)
20 {
21     int numérateur { a.numérateur * b.dénominateur - b.numérateur *
22                     a.dénominateur };
23     int dénominateur { a.dénominateur * b.dénominateur };
24     return { numérateur, dénominateur };
```

III. On passe la deuxième!

```
25 }
26
27 int main()
28 {
29     Fraction const un_demi { 1, 2 };
30     Fraction const trois_quarts { 3, 4 };
31     Fraction const deux_huitiemes { 2, 8 };
32
33     Fraction const resultat { un_demi + trois_quarts -
34                             deux_huitiemes };
35     std::cout << "1/2 + 3/4 - 2/8 font " << resultat.numerateur <<
36               "/" << resultat.denominateur << std::endl;
37
38     return 0;
39 }
```

```
1 1/2 + 3/4 - 2/8 font 64/64
```



Remarque

Le résultat de 64/64 peut vous surprendre, mais il est tout à fait normal, parce qu'on ne simplifie pas la fraction.

Entraînez-vous en implémentant aussi la multiplication et la division. Le code est très proche de celui qu'on vient d'écrire.

III.9.0.0.1. Les opérateurs de comparaison

Continuons en essayant maintenant de comparer des fractions entre elles, pour savoir notamment si deux fractions sont égales ou non, comparer la grandeur de deux fractions, etc. C'est le même principe, les prototypes peuvent se déduire assez facilement.

```
1 bool operator==(T const& lhs, T const& rhs);
2 bool operator!=(T const& lhs, T const& rhs);
3 bool operator<(T const& lhs, T const& rhs);
4 bool operator>(T const& lhs, T const& rhs);
5 bool operator<=(T const& lhs, T const& rhs);
6 bool operator>=(T const& lhs, T const& rhs);
```

Et si vous êtes effrayés à l'idée d'implémenter autant d'opérateurs différents, rappelez-vous de la logique booléenne. Vous verrez, ça réduit énormément la quantité de code.

III. On passe la deuxième!

```
1  #include <iostream>
2  #include <string>
3
4  struct Fraction
5  {
6      int numérateur;
7      int dénominateur;
8  };
9
10 // On vérifie que a et b sont égaux.
11 bool operator==(Fraction const & a, Fraction const & b)
12 {
13     Fraction const gauche { a.numérateur * b.dénominateur,
14                             a.dénominateur * b.dénominateur };
15     Fraction const droite { b.numérateur * a.dénominateur,
16                             b.dénominateur * a.dénominateur };
17     return gauche.numérateur == droite.numérateur &&
18            gauche.dénominateur == droite.dénominateur;
19 }
20
21 // Pour aller plus vite, il suffit de retourner la négation de la
22 // condition 'a == b'.
23 bool operator!=(Fraction const & a, Fraction const & b)
24 {
25     return !(a == b);
26 }
27
28 // On vérifie si a est plus petit que b.
29 bool operator<(Fraction const & a, Fraction const & b)
30 {
31     if (a.numérateur == b.numérateur)
32     {
33         return a.dénominateur > b.dénominateur;
34     }
35     else if (a.dénominateur == b.dénominateur)
36     {
37         return a.numérateur < b.numérateur;
38     }
39     else
40     {
41         Fraction const gauche { a.numérateur * b.dénominateur,
42                                 a.dénominateur * b.dénominateur };
43         Fraction const droite { b.numérateur * a.dénominateur,
44                                 b.dénominateur * a.dénominateur };
45         return gauche.numérateur < droite.numérateur;
46     }
47 }
```

III. On passe la deuxième!

```
43 // Pour simplifier, il suffit de vérifier que a n'est ni inférieur,
    ni égal à b.
44 bool operator>(Fraction const & a, Fraction const & b)
45 {
46     return !(a < b) && a != b;
47 }
48
49 // Pour simplifier, il suffit de vérifier que a est inférieur ou
    égal à b.
50 bool operator<=(Fraction const & a, Fraction const & b)
51 {
52     return a < b || a == b;
53 }
54
55 // Pour simplifier, il suffit de vérifier que a est supérieur ou
    égal à b.
56 bool operator>=(Fraction const & a, Fraction const & b)
57 {
58     return a > b || a == b;
59 }
60
61 int main()
62 {
63     Fraction const trois_septieme { 3, 7 };
64     Fraction const un_quart { 1, 4 };
65
66     std::cout << std::boolalpha;
67     std::cout << "Est-ce que 3/7 est plus petit que 1/4 ? " <<
        (trois_septieme < un_quart) << std::endl;
68     std::cout << "Est ce que 3/7 est plus grand que 1/4 ? " <<
        (trois_septieme > un_quart) << std::endl;
69
70     Fraction const neuf_vingteunieme { 9, 21 };
71     std::cout << "Est-ce que 3/7 et 9/21 sont égaux ? " <<
        (trois_septieme == neuf_vingteunieme) << std::endl;
72
73     return 0;
74 }
```

```
1 Est-ce que 3/7 est plus petit que 1/4 ? false
2 Est ce que 3/7 est plus grand que 1/4 ? true
3 Est-ce que 3/7 et 9/21 sont égaux ? true
```

Aviez-vous pensé à l'astuce consistant à réutiliser le code déjà écrit pour simplifier celui à écrire? Si oui, c'est que vous commencez à acquérir les bons réflexes de développeur. 🍊

III. On passe la deuxième!

III.9.0.0.2. Les opérateurs de manipulation des flux

Il nous reste le problème d’affichage de notre type `Fraction`. Pour l’instant, nous sommes obligés d’aller chercher le numérateur et le dénominateur nous-mêmes. Si notre type contient plein d’informations, toutes aller les récupérer devient vite pénible.

Les opérateurs de manipulation des flux `<` et `>` sont, fort heureusement, surchargeables. Voyez leur prototype.

```
1 std::ostream& operator<<(std::ostream & os, T const & t);
2 std::istream& operator>>(std::istream & is, T & t);
```

Ce qui peut surprendre, c’est la fait que l’on renvoie une référence sur un flux à chaque fois. En fait, c’est le flux que l’on reçoit en argument qui est renvoyé. Ceci a pour but d’enchaîner des appels à l’opérateur pour pouvoir écrire du code tel que celui-ci.

```
1 std::cout << t << std::endl;
```

Il sera interprété ensuite comme une suite d’appel à l’opérateur `<` par le compilateur.

```
1 operator<<(operator<<(std::cout, t), std::endl);
```

i

Type retourné

Notez que les types retournés sont `std::ostream` et `std::istream`. Ce sont des **versions génériques de flux de sortie et d’entrée**. Ainsi, en renvoyant ces types, on s’assure que notre type peut être utilisé avec des flux vers l’entrée / sortie standards, vers des fichiers, des chaînes de caractères, etc.

Dans notre cas, celui qui nous intéresse le plus, c’est `operator<`, afin de pouvoir afficher nos fractions sur la sortie standard.

```
1 #include <iostream>
2 #include <string>
3
4 struct Fraction
5 {
6     int numerateur;
7     int denominateur;
8 };
9
10 Fraction operator+(Fraction const & a, Fraction const & b)
```


III. On passe la deuxième!

```
11 {
12     int numérateur { a.numérateur * b.denominateur + b.numérateur *
13         a.denominateur };
14     int denominateur { a.denominateur * b.denominateur };
15     return { numérateur, denominateur };
16 }
17
18 std::ostream & operator<<(std::ostream & flux, Fraction const &
19     fraction)
20 {
21     return flux << fraction.numérateur << "/" <<
22         fraction.denominateur;
23 }
24
25 int main()
26 {
27     Fraction const un_quart { 1, 4 };
28     Fraction const neuf_vingteunieme { 9, 21 };
29     std::cout << "1/4 + 9/21 = " << un_quart + neuf_vingteunieme <<
30         std::endl;
31     return 0;
32 }
```

```
1 1/4 + 9/21 = 57/84
```

Si vous le souhaitez, vous pouvez vous amuser à implémenter l'opérateur `>`, pour, par exemple, être capable de créer des `Fraction` depuis des chaînes de caractères ou en lisant un fichier.

III.9.0.0.3. Plus et moins unaires

Il existe encore beaucoup d'opérateurs, nous n'allons donc pas pouvoir tous les aborder dans ce chapitre. J'aimerais néanmoins vous parler du plus et du moins unaires, qui permettent de faire des opérations comme `+variable` ou `-variable`, exactement comme des entiers (entre autres).

Le modèle est très simple, le voici ci-dessous pour les deux opérateurs.

```
1 T operator+(T const & t);
2 T operator-(T const & t);
```

Dans notre code, cela nous permet donc de créer, notamment, des fractions négatives.

III. On passe la deuxième!

```
1 #include <cmath>
2 #include <iostream>
3 #include <string>
4
5 struct Fraction
6 {
7     int numerateur;
8     int denominateur;
9 };
10
11 Fraction operator-(Fraction const & fraction)
12 {
13     return { -std::abs(fraction.numerateur),
14             std::abs(fraction.denominateur) };
15 }
16
17 std::ostream & operator<<(std::ostream & flux, Fraction const &
18 fraction)
19 {
20     return flux << fraction.numerateur << "/" <<
21             fraction.denominateur;
22 }
23
24 int main()
25 {
26     Fraction const un_quart { 1, 4 };
27     std::cout << "L'opposé de 1/4 est " << -un_quart << "." <<
28             std::endl;
29     return 0;
30 }
```

```
1 L'opposé de 1/4 est -1/4.
```

III.9.0.0.4. Aller plus loin

Pour pratiquer, je vous propose de finir ce petit projet de fractions. Voici quelques idées en vrac.

- Il n'y a aucune gestion des erreurs. Comment y remédier?
- il n'y a pas de tests unitaires.
- Renvoyer l'opposé d'une fraction.
- Inverser une fraction.
- Rendre une fraction irréductible. Pour cela, il faut diviser le numérateur et le dénominateur par leur PGCD.

III.9.0.1. Et la bibliothèque standard ?

Elle utilise à foison la surcharge d'opérateurs. Et vous en avez utilisé énormément sans le savoir. Examinons quelques cas.

III.9.0.1.1. Concaténation de chaînes

```
1 std::string const hello { "Hello" };
2 std::string const world { "World" };
3
4 std::string const texte { hello + " " + world };
```

Quand vous concaténez des chaînes de caractères, vous faites en réalité appel à `operator+` appliqué au type `std::string`. L'utilisation de cet opérateur est parfois décrié par certains, qui le jugent peu adapté. Reste que, celui-ci ou un autre, le code est raccourci et plus agréable à lire.

III.9.0.1.2. Accès à un élément

```
1 std::vector<int> const tableau { 1, 2, 3, 4 };
2 int const nombre { tableau[0] };
```

Dans l'exemple précédent, vous pouvez remplacer `std::vector` par `std::array` ou `std::map`, le principe est toujours le même. Derrière, c'est en effet `operator[]` qui est appelé. Celui-ci attend un entier comme argument, ce qui ne doit pas vous surprendre, puisque vous êtes habitués à la manipulation de tableaux maintenant.

III.9.0.1.3. Définition de littéraux `std::string`

```
1 using namespace std::literals;
2 auto chaine { "Hello, chaine C++"s }
```

Vous vous demandez où est la surcharge dans le code précédent? Allez, je vous aide, c'est le `s` en fin de chaîne. Eh oui, même quelque chose d'aussi simple est en fait une surcharge de `operator""`. Ce petit nouveau, arrivé avec C++11, permet de définir des suffixes, qui s'ajoutent après des littéraux, afin d'en changer le type.

Dans le cas exposé ci-dessus, le suffixe `s` permet de transformer le type du littéral `"Hello, chaine C++"`, qui était un type hérité du C, en un `std::string`. Sans cette surcharge, il aurait fallu écrire `std::string { "Hello, chaine C++" }`, ce qui est plus long et moins lisible.

III.9.0.2. Exercice—Calculs avec des durées

Pratiquons pour mieux retenir avec un petit exercice, dans lequel on va implémenter quelques opérations pour un type `Duree` qui, comme son nom l'indique, représente une durée. Voici déjà comment on définit une durée dans notre code.

```
1 struct Duree
2 {
3     int secondes;
4 };
```

Comme vous voyez, on stocke les durées sous forme de secondes. L'exercice consiste à implémenter une liste d'opérations.

- Créer une durée à partir de secondes, minutes (optionnel), heures (optionnel).
- Renvoyer la durée sous forme de secondes.
- Renvoyer la durée sous forme de tuple {minutes, secondes}. Le signe de la durée sera porté par les minutes.
- Renvoyer la durée sous forme de tuple {heures, minutes, secondes}. Le signe de la durée sera porté par les heures.
- Additionner des durées.
- Renvoyer l'opposé d'une durée.
- Soustraire des durées.
- Comparer des durées.
- Injecter une durée dans un flux sortant, au format `H:mm:ss`. Le signe de la durée sera mis devant les heures.



Définition d'une durée

Ici, on considère une durée comme un intervalle de temps entre deux événements. Par conséquent, une durée peut être négative; les opérations comme prendre l'opposé d'une durée ont donc un sens.

III.9.0.2.1. Indice

Voici, pour vous aider, les prototypes des fonctions à implémenter.

👁 Prototypes

III.9.0.2.2. Corrigé

Voici le corrigé complet avec les implémentations.

III.9.0.3. Sur le bon usage de la surcharge

La surcharge d'opérateurs est un outil puissant, il faut donc faire attention à ne pas l'utiliser n'importe comment. Notamment, C++ ne vérifie pas ce que vous mettez dans votre implémentation. Vous pourriez alors surcharger l'opérateur `/` pour effectuer une addition. Dans ce cas, vous violeriez la **sémantique** des opérateurs, c'est-à-dire que vous leur feriez faire autre chose que ce pour quoi ils sont prévus normalement. C'est une **très mauvaise pratique**, à bannir absolument.

L'autre point, c'est que, même si vous pouvez surcharger des opérateurs, **cela n'a pas toujours de sens de le faire**. Ainsi, additionner deux personnes ou deux comptes en banque **n'a aucun sens et n'est pas correct**. Nous reviendrons sur cet aspect plus tard dans le cours. Retenez que ce n'est pas parce que vous pouvez que vous devez.

III.9.0.4. En résumé

- On peut définir des alias de type grâce au mot-clé `using`.
- On peut simplifier l'utilisation de `std::tuple` et même de structures grâce aux décompositions.
- La surcharge d'opérateurs permet d'augmenter la lisibilité et la qualité de notre code.
- Il est possible de surcharger de nombreux opérateurs.
- Elle doit être utilisée à bon escient, en ne faisant pas faire aux opérateurs des opérations qui violent leur sémantique.
- La surcharge d'opérateurs est un bonus qui, s'il est appréciable, ne doit pas pour autant être constamment et systématiquement utilisé.

Contenu masqué

Contenu masqué n°45 : Prototypes

```
1 Duree creer_duree(int secondes, int minutes = 0, int heures = 0);
2 int en_secondes(Duree d);
3 std::tuple<int, int> en_minutes(Duree d);
4 std::tuple<int, int, int> en_heures(Duree d);
5
6 Duree operator+(Duree const& lhs, Duree const& rhs);
7 Duree operator-(Duree const& d);
```

III. On passe la deuxième!

```
8 Duree operator-(Duree const& lhs, Duree const& rhs);
9
10 bool operator==(Duree const& lhs, Duree const& rhs);
11 bool operator!=(Duree const& lhs, Duree const& rhs);
12 bool operator<(Duree const& lhs, Duree const& rhs);
13 bool operator>(Duree const& lhs, Duree const& rhs);
14 bool operator<=(Duree const& lhs, Duree const& rhs);
15 bool operator>=(Duree const& lhs, Duree const& rhs);
16
17 std::ostream& operator<<(std::ostream& os, Duree const & d);
```

[Retourner au texte.](#)

Contenu masqué n°46 : Corrigé complet

```
1 #include <iostream>
2 #include <tuple>
3
4 struct Duree
5 {
6     int secondes;
7 };
8
9 Duree creer_duree(int secondes, int minutes = 0, int heures = 0);
10 int en_secondes(Duree d);
11 std::tuple<int, int> en_minutes(Duree d);
12 std::tuple<int, int, int> en_heures(Duree d);
13
14 Duree operator+(Duree const& lhs, Duree const& rhs);
15 Duree operator-(Duree const& d);
16 Duree operator-(Duree const& lhs, Duree const& rhs);
17
18 bool operator==(Duree const& lhs, Duree const& rhs);
19 bool operator!=(Duree const& lhs, Duree const& rhs);
20 bool operator<(Duree const& lhs, Duree const& rhs);
21 bool operator>(Duree const& lhs, Duree const& rhs);
22 bool operator<=(Duree const& lhs, Duree const& rhs);
23 bool operator>=(Duree const& lhs, Duree const& rhs);
24
25 std::ostream& operator<<(std::ostream& os, Duree d);
26
27 int main()
28 {
29
30
```

III. On passe la deuxième!

```
31     return 0;
32 }
33
34 Duree creer_duree(int secondes, int minutes, int heures)
35 {
36     return Duree{secondes + 60 * minutes + 3600 * heures};
37 }
38
39 int en_secondes(Duree d)
40 {
41     return d.secondes;
42 }
43
44 std::pair<int, int> div_eucl_ordinaire(int i, int div)
45 {
46     if(i >= 0)
47         return {i / div, i % div};
48
49     return {i / div - 1, i % div + 60};
50 }
51
52 std::tuple<int, int> en_minutes(Duree d)
53 {
54     return div_eucl_ordinaire(en_secondes(d), 60);
55 }
56
57 std::tuple<int, int, int> en_heures(Duree d)
58 {
59     auto [minutes, secondes] = en_minutes(d);
60     auto heures{0};
61     std::tie(heures, minutes) = div_eucl_ordinaire(minutes, 60);
62     return {heures, minutes, secondes};
63 }
64
65
66 Duree operator+(Duree const& lhs, Duree const& rhs)
67 {
68     return Duree{lhs.secondes + rhs.secondes};
69 }
70
71 Duree operator-(Duree const& d)
72 {
73     return Duree{-d.secondes};
74 }
75
76 Duree operator-(Duree const& lhs, Duree const& rhs)
77 {
78     return lhs + (-rhs);
79 }
80
```

III. On passe la deuxième!

```
81
82 bool operator==(Duree const& lhs, Duree const& rhs)
83 {
84     return lhs.secondes == rhs.secondes;
85 }
86
87 bool operator!=(Duree const& lhs, Duree const& rhs)
88 {
89     return !(lhs == rhs);
90 }
91
92 bool operator<(Duree const& lhs, Duree const& rhs)
93 {
94     return lhs.secondes < rhs.secondes;
95 }
96
97 bool operator>(Duree const& lhs, Duree const& rhs)
98 {
99     return lhs.secondes > rhs.secondes;
100 }
101
102 bool operator<=(Duree const& lhs, Duree const& rhs)
103 {
104     return !(lhs > rhs);
105 }
106
107 bool operator>=(Duree const& lhs, Duree const& rhs)
108 {
109     return !(lhs < rhs);
110 }
111
112 std::ostream& operator<<(std::ostream& os, Duree const & duree)
113 {
114     auto [heures, minutes, secondes] = en_heures(duree);
115
116     auto format = [](int i)
117     {
118         auto res = std::to_string(i);
119         if(std::size(res) == 1)
120             res = '0' + res;
121         return res;
122     };
123
124     return os << heures << ':' << format(minutes) << ':' <<
125         format(secondes) << 's';
126 }
```

[Retourner au texte.](#)

III.10. [T.P] Un gestionnaire de discographie

Depuis le début de ce cours, nous avons eu l'occasion de découvrir de nombreux concepts, que l'on a pratiqués lors des exercices et des mini-T.P.

Dans ce chapitre, nous allons pour la première fois attaquer un gros T.P, dont le but va être d'utiliser toutes les notions acquises jusqu'à présent pour produire un programme complet et fonctionnel. C'est votre baptême du feu, en quelque sorte. Mais ne vous inquiétez pas, on va y aller pas à pas. 🍌

III.10.1. L'énoncé

Lors de ce T.P, vous allez créer un gestionnaire de discographie en ligne de commande. C'est un logiciel qui va permettre d'ajouter des morceaux avec album et artiste à votre discographie. Il donnera alors la possibilité d'afficher votre discographie de plusieurs manières différentes, ainsi que de l'enregistrer et de la charger à partir de fichiers.

Toutes les actions devront être faisables par l'utilisateur en ligne de commande. Il aura une petite invite de commande attendant qu'il rentre ses ordres.

```
1 >
```

III.10.1.1. Ajout de morceaux

Pour ajouter un morceau à la discographie, l'utilisateur le fera simplement en tapant une commande de cette forme:

```
1 > ajouter morceau | album | artiste
```

Les informations sont optionnelles. Par exemple, l'utilisateur doit pouvoir taper ça:

```
1 > ajouter morceau | album
```

III. On passe la deuxième!

Ou ça:

```
1 > ajouter morceau | | artiste
```

Ou encore ceci:

```
1 > ajouter morceau
```

Et même ceci:

```
1 > ajouter
```

Lorsque des informations sont omises, la discographie enregistrera le morceau avec les mentions «Morceau inconnu», «Album inconnu» ou «Artiste inconnu».

III.10.1.2. Affichage de la discographie

L'utilisateur doit pouvoir afficher sa discographie de trois manières différentes, que l'on va illustrer par un exemple. Les commandes d'affichage commenceront toutes par le même mot.

```
1 > afficher
```

Prenons pour exemple la discographie suivante.

Morceau	Album	Artiste
Bad	Bad	Michael Jackson
Bloody Well Right	Crime of the Century	Supertramp
It's Raining Again	...Famous Last Words...	Supertramp
Buffalo Soldier	Confrontation	Bob Marley and the Wailers
The Way You Make Me Feel	Bad	Michael Jackson

III.10.1.2.1. Affichage par ordre alphabétique des morceaux

L'utilisateur pourra taper `> afficher morceaux`, auquel cas le programme devra lui afficher la liste des morceaux par ordre alphabétique.

III. On passe la deuxième!

```
1 > afficher morceaux
2 --> Bad | Bad | Michael Jackson
3 --> Bloody Well Right | Crime of the Century | Supertramp
4 --> Buffalo Soldier | Confrontation | Bob Marley and the Wailers
5 --> It's Raining Again | ...Famous Last Words... | Supertramp
6 --> The Way You Make Me Feel | Bad | Michael Jackson
```

III.10.1.2.2. Affichage par album

Si l'utilisateur tape `> afficher albums`, le programme devra lui répondre avec la liste des albums, avec une sous-liste des morceaux de ces albums, le tout étant trié par ordre alphabétique.

```
1 > afficher albums
2 --> Bad | Michael Jackson
3   /--> Bad
4   /--> The Way You Make Me Feel
5 --> Confrontation | Bob Marley and the Wailers
6   /--> Buffalo Soldier
7 --> Crime of the Century | Supertramp
8   /--> Bloody Well Right
9 --> ...Famous Last Words... | Supertramp
10  /--> It's Raining Again
```

III.10.1.2.3. Affichage par artiste

C'est le même principe: si l'utilisateur tape `> afficher artistes`, le programme répondra une liste des auteurs, avec une sous-liste d'albums contenant elle-même une sous-liste de morceaux.

```
1 > afficher artistes
2 --> Michael Jackson
3   /--> Bad
4     /--> Bad
5     /--> The Way You Make Me Feel
6 --> Bob Marley and the Wailers
7   /--> Confrontation
8     /--> Buffalo Soldier
9 --> Supertramp
10  /--> Crime of the Century
11    /--> Bloody Well Right
12 --> Supertramp
13  /--> ...Famous Last Words...
```

III. On passe la deuxième!

```
14      /--> It's Raining Again
```

Bien sûr, si vous voulez adapter un peu l’affichage pour qu’il vous plaise plus, vous pouvez. Après tout, l’un des intérêts de savoir programmer, c’est de pouvoir faire des programmes qui nous plaisent!

III.10.1.3. Enregistrement et chargement d’une discographie

L’utilisateur doit pouvoir enregistrer sa discographie sur le disque, pour pouvoir la sauvegarder lorsqu’il arrête le programme. Il doit aussi pouvoir la charger depuis un fichier. Ces deux fonctionnalités s’utiliseront à travers deux commandes.

```
1 > enregistrer nom_fichier
```

```
1 > charger nom_fichier
```

III.10.1.4. Quitter le programme

Lorsqu’il souhaite quitter le programme, l’utilisateur aura simplement à taper la commande qui suit.

```
1 > quitter
```

III.10.1.5. Gestion des erreurs

Le programme devra être robuste aux erreurs de saisie: si l’utilisateur tape une commande invalide, le programme ne devra pas crasher ou avoir un comportement bizarre. Il doit ignorer la commande et signaler à l’utilisateur que la commande est invalide.

```
1 > afficher qssqdqs
2 Erreur : Commande invalide.
```

Le programme devra aussi être capable de gérer les erreurs dues à l’impossibilité d’ouvrir un fichier, que ce soit en lecture ou en écriture.

III.10.1.6. Dernières remarques

Ce T.P est votre première confrontation à la conception logicielle. La difficulté ne réside plus seulement dans la compréhension et l'utilisation du langage, mais aussi dans la conception: il va vous falloir réfléchir à comment organiser votre code.

L'objectif est que vous réfléchissiez à une manière d'avoir un code propre, robuste, agréable à lire, ouvert aux évolutions, et bien sûr fonctionnel.

C'est une tâche difficile, mais rassurez-vous: comme c'est votre première fois, nous vous proposons un guide pour pouvoir avancer dans le T.P. Prenez cependant le temps de réfléchir et cassez-vous la tête sur le T.P sans utiliser le guide. Cela vous confrontera à la difficulté et vous fera progresser. Ensuite, vous pourrez lire le guide pour vous aiguiller vers une solution. Et comme d'habitude, on vous proposera un corrigé complet à la fin.

III.10.2. Guide de résolution

III.10.2.1. Analyse des besoins

Pour concevoir un programme efficacement, il faut analyser les besoins auxquels il doit répondre—autrement dit les tâches qu'il doit être capable de réaliser—pour arriver à des problèmes plus atomiques, auxquels on peut répondre directement par un morceau de code, une fonction, une structure de données...

Ici, les besoins sont définis par l'énoncé; on va donc partir de là et les découper en petits problèmes.

III.10.2.1.1. Un interpréteur de commande

Déjà, l'utilisateur donne ses instructions au moyen d'une ligne de commande. Notre logiciel doit donc être capable de lire et comprendre les commandes de l'utilisateur, ainsi que de détecter les commandes invalides.

Il y a plusieurs types de commande: les commandes d'ajout de morceau, les commandes de sauvegarde/chargement de la bibliographie, et bien sûr les commandes d'affichage. Celles-ci se décomposent ensuite en plusieurs modes d'affichage. On a donc deux tâches distinctes.

- Lire et interpréter les commandes.
- Exécuter les commandes.

On va avoir une fonction d'exécution pour chaque type de commande, et des fonctions qui se chargeront de lire les commandes et d'appeler la fonction d'exécution adéquate; ce seront nos fonctions d'interprétation. Nous reviendrons un peu plus tard sur l'implémentation, mais on a déjà une idée du schéma de fonctionnement global.

III.10.2.1.2. Le besoin de stocker la discographie

Évidemment, il nous faut un moyen de mémoriser la discographie, que ce soit pendant l'exécution du programme (stockage en mémoire de travail dans une variable) ou sur le long terme (stockage sur disque dans un fichier). On dégage donc ici deux besoins techniques:

- Le premier est de pouvoir représenter la bibliographie dans notre programme; en C++, cela revient à peu de choses près à définir un type adapté. C'est par cet aspect que nous rentrerons en premier dans les détails techniques; le choix de commencer par là est pertinent car cette représentation est centrale et sera omniprésente dans le reste du code.
- Le second est de réussir à sauvegarder et charger cette représentation dans un fichier; on doit entre autres choisir une représentation de la bibliothèque sous un format texte.

III.10.2.1.3. Le besoin de gérer les erreurs d'entrée

L'énoncé explique qu'il faut gérer les erreurs d'entrée - et ce de manière *user friendly*. Plus précisément, il faut que nos fonctions d'interprétation détectent les erreurs. Elles doivent aussi pouvoir les remonter. Pour cela, on peut tout simplement utiliser le système d'exceptions. On y reviendra plus en détails un peu plus loin.

III.10.2.2. Attaquons le code !

On peut maintenant aborder la conception plus précise des différents points abordés précédemment, ainsi que l'implémentation. Comme on l'a dit, on va commencer par un point central: la représentation de la discographie.

III.10.2.2.1. Représentation de la discographie

Une discographie peut simplement être vue comme une collection de morceaux. D'après l'énoncé, les informations dont on a besoin pour un morceau sont son nom, son album et son artiste. Ainsi:

- Un artiste a comme unique donnée son nom.
- Un album a lui aussi un nom.
- Un morceau a pour données importantes son nom, ainsi que l'album et l'artiste correspondants.
- Une discographie est une collection de morceaux, de taille variable évidemment.

Ces constats étant fait, cela devrait vous aider à trouver des structures de données adaptées.

☞ Structures de données

III. On passe la deuxième!

III.10.2.2.2. Fonctions de base

Une fois nos structures de données définies, il est commun de créer des fonctions de base qui vont nous permettre de les manipuler de manière agréable. C'est typiquement le moment de créer des surcharges pour certains opérateurs pour nos types nouvellement créés. Notamment, on aimerait pouvoir afficher facilement nos différentes structures, ainsi que les lire et les écrire dans un fichier. On va donc avoir besoin de surcharger les opérateurs de flux.

Je vous suggère dans un premier temps d'essayer de trouver les prototypes de telles fonctions, puis d'essayer de les implémenter.

👁 Prototypes

👁 Implémentations

Comme nous sommes des programmeurs consciencieux, nous voulons écrire des tests pour nous assurer que ce que nous avons déjà écrit fonctionne bien. En plus de vérifier qu'une entrée valide est acceptée, nous devons aussi penser aux cas limites ainsi qu'aux erreurs, pour mesurer la résistance et la bonne conception de notre programme.

Par exemple, "Frica | Frica | Carla's Dreams" et "Levels | Levels | Avicii" sont des entrées valides et complètes, même si dans la seconde il y a beaucoup d'espaces. De même, "Subeme la radio | | Enrique Iglesias" et "Duel of the fates | |" aussi, quand bien même certaines informations sont manquantes.

👁 Tests unitaires

III.10.2.2.3. Le stockage dans un fichier

Maintenant que l'on a construit nos structures de base pour pouvoir représenter une bibliographie en mémoire, on peut créer des fonctions pour sauvegarder et charger une discographie dans un fichier.

👁 Implémentation

III.10.2.2.4. L'affichage de la discographie

Autre point qu'on a soulevé, l'affichage. Nous voudrions pouvoir afficher la discographie par albums, par artistes ou par titres. Plus précisément, on remarque que pour chaque mode d'affichage, il y a en réalité deux composantes distinctes: la manière d'afficher à proprement parler, mais aussi l'ordre dans lequel sont affichés les morceaux, c'est-à-dire le tri de la discographie.

III. On passe la deuxième!

Nous dégageons ainsi six fonctions à implémenter: une de tri et une d'affichage pour chaque mode d'affichage.

☉ Prototypes

☉ Implémentation

i

La manière dont j'ai remarqué le fait qu'il y avait deux parties bien distinctes à envisager - et donc à traiter dans des fonctions séparées - est très intéressante, donc j'aimerais vous l'expliquer.

Au départ, le tri se faisait directement dans les fonctions d'affichage. Or des fonctions d'affichage, vu leurs rôles, ne devraient pas avoir à modifier la discographie; en bon programmeur qui respecte la *const-correctness*, je souhaitais donc que ces trois fonctions prennent en paramètres une `Discographie const&` et non pas une `Discographie&`. Mais le tri modifie la discographie, donc je ne pouvais pas. C'est comme ça que j'ai remarqué que la logique d'affichage comprenait en fait deux sous-tâches indépendantes, à traiter séparément.

Ainsi, on est dans un cas où faire attention à la *const-correctness* permet d'améliorer la conception du code.

Pour lier tout ça, on peut créer une quatrième fonction qui lancera les fonctions du mode d'affichage demandé. Comme nos possibilités d'affichage sont limitées, on peut les regrouper dans une énumération.

☉ Prototypes

☉ Implémentation

III.10.2.2.5. Le système de commandes

Il nous reste encore à interagir avec l'utilisateur, à l'aide du système de commandes. On a fait ressortir deux problèmes principaux, l'interprétation et l'exécution des commandes. On gardera en tête le fait qu'il faut gérer les erreurs.

☉ Prototypes

Essayez maintenant d'implémenter de telles fonctions.

👁 Implémentations

III.10.2.2.6. Finalisation

Maintenant que toutes nos fonctionnalités sont programmées, il suffit de tout intégrer en écrivant une fonction `main`, qui aura pour principal mission d'initialiser le programme (en lançant d'abord les tests) et de faire tourner une boucle d'invite de commandes jusqu'à ce que l'utilisateur décide de quitter.

👁 Fonction principale

Et voilà! On a construit pas à pas toutes les briques de notre programme, jusqu'à avoir tout le code pour le faire fonctionner. La démarche à retenir est la suivante: il faut réfléchir à une manière de découper notre problème initial—ici «faire une discographie»—en sous-problèmes, par exemple «lire et comprendre une commande», qui eux-mêmes seront découpés en sous-problèmes, jusqu'à arriver à des briques de base simples à réaliser—par exemple «créer un type représentant les morceaux».

III.10.3. Corrigé complet

Bon, j'espère que vos phases de recherches et d'avancées à l'aide du guide ont été fructueuses!

Que vous ayez réussi ou pas à avoir un résultat fonctionnel, l'important est que vous ayez bien réfléchi au T.P. C'est en vous confrontant à vos difficultés que vous progresserez.

Bref, trêve de bavardages, voici sans plus attendre une solution possible au T.P, issue du guide ci-dessus.

👁 Corrigé complet



Une façon parmi d'autres

Ce code n'est pas LA solution, mais une parmi tant d'autres. Le vôtre est différent, mais vous êtes malgré tout arrivés à la solution? Bravo, vous pouvez être fiers. 🍊

III.10.4. Conclusion et pistes d'améliorations

Voilà qui conclut ce TP, dans lequel vous avez pu vous confronter pour la première fois à la conception d'un programme complet, plus ambitieux que les petits exercices. Cela nous a permis de voir la démarche qu'il faut avoir en tête pour pouvoir créer du code de qualité, c'est-à-dire expressif et surtout évolutif.

Justement, cette évolutivité vous permet d'ajouter des fonctionnalités à votre programme à votre guise, et ce de manière efficace. Vous pouvez ajouter ce que vous voulez. Ci-dessous, voici quelques idées.

- Ajouter des options d'affichage comme l'affichage des morceaux d'un seul album ou artiste, grâce à des commandes du type `> afficher artiste Dave Brubeck` par exemple.
- Ajouter des options de tri des morceaux, par BPM par exemple.
- Avoir un fichier de configuration du logiciel permettant à l'utilisateur de choisir son invite de commandes, son séparateur dans les affichages et dans les commandes d'ajout, les puces à utiliser dans les listes affichées, etc. En bref, rendre le logiciel configurable. Un tel fichier pourrait ressembler à ça, pour les réglages donnant les mêmes affichages que ceux qu'on a codés en dur.

```
1 invite=">"
2 separateur_affichage="|"
3 separateur_ajout="|"
4 puce_niveau1="-->"
5 puce_niveau2="/-->"
6 puce_niveau3="/-->"
```

- Notre code n'est pas entièrement testé. Il ne tient qu'à vous d'écrire les tests manquants.

Les possibilités d'ajout sont infinies, vous pouvez faire ce que vous souhaitez. 🍊

N'hésitez pas, lorsque vous voulez ajouter des fonctionnalités, à continuer avec la même démarche: commencez par réfléchir, décomposez ce que vous voulez faire en petits problèmes, créez des types pratiques s'il le faut, etc.



Conseil d'ami

Je vous conseille de lire le chapitre suivant avant de faire des ajouts au TP. En effet, vous serez alors capables de mieux organiser votre code, ce qui vous aidera à le faire évoluer pour ajouter des nouvelles fonctionnalités.

III.10.4.1. En résumé

- Vous vous êtes confrontés pour la première fois à la réalité de la conception d'un programme informatique.
- On a pu en tirer une démarche pour avoir un code lisible et évolutif.

III. On passe la deuxième!

- On a également pu voir l'importance de la réflexion si on veut obtenir un code de qualité.
- On a aussi vu comment les notions vues jusqu'à présent peuvent se combiner en pratique, pour en tirer le meilleur et ainsi obtenir cette qualité.
- Nous avons vu quelques pistes possibles d'amélioration, mais il ne tient qu'à vous d'ajouter les fonctionnalités que vous souhaitez, comme vous le souhaitez.

Contenu masqué

Contenu masqué n°47 : Structures de données

Un choix possible est le suivant: représenter les artistes, albums et morceaux par des structures simples, et la discographie par un alias, à l'aide de `using`.

```
1 struct Artiste
2 {
3     std::string nom;
4 };
5
6 struct Album
7 {
8     std::string nom;
9 };
10
11 struct Morceau
12 {
13     std::string nom;
14     Album album;
15     Artiste compositeur;
16 };
17
18 // Une discographie n'est rien d'autre qu'un ensemble de morceaux.
19 using Discographie = std::vector<Morceau>;
```

[Retourner au texte.](#)

Contenu masqué n°48 :

Prototypes

```
1 std::ostream & operator<<(std::ostream & sortie, Artiste const &
  artiste);
2 std::ostream & operator<<(std::ostream & sortie, Album const &
  album);
3 std::ostream & operator<<(std::ostream & sortie, Morceau const &
  morceau);
4 std::istream & operator>>(std::istream & entree, Morceau &
  morceau);
```

[Retourner au texte.](#)

Contenu masqué n°49 : Implémentations

```
1 // Fonction utilisée pour nettoyer les entrées.
2 // Elle sera réutilisée à plusieurs endroits.
3 std::string traitement_chaine(std::string const & chaine)
4 {
5     std::string copie { chaine };
6
7     // D'abord on enlève les espaces au début...
8     auto premier_non_espace { std::find_if_not(std::begin(copie),
9         std::end(copie), isspace) };
10    copie.erase(std::begin(copie), premier_non_espace);
11
12    // ...puis à la fin.
13    std::reverse(std::begin(copie), std::end(copie));
14    premier_non_espace = std::find_if_not(std::begin(copie),
15        std::end(copie), isspace);
16    copie.erase(std::begin(copie), premier_non_espace);
17    std::reverse(std::begin(copie), std::end(copie));
18
19    return copie;
20 }
21
22 // On surcharge l'opérateur << pour être capable d'afficher le type
23 // Artiste.
24 std::ostream & operator<<(std::ostream & sortie, Artiste const &
25     artiste)
26 {
27     sortie << artiste.nom;
28     return sortie;
29 }
```

III. On passe la deuxième!

```
26
27 // On surcharge l'opérateur << pour être capable d'afficher le type
    Album.
28 std::ostream & operator<<(std::ostream & sortie, Album const &
    album)
29 {
30     sortie << album.nom;
31     return sortie;
32 }
33
34 // On surcharge l'opérateur << pour être capable d'afficher le type
    Morceau.
35 std::ostream & operator<<(std::ostream & sortie, Morceau const &
    morceau)
36 {
37     sortie << morceau.nom << " | " << morceau.album << " | " <<
        morceau.compositeur;
38     return sortie;
39 }
40
41 // Conversion d'un flux d'entrée (std::cin, chaîne, fichier) en
    type Morceau.
42 std::istream & operator>>(std::istream & entree, Morceau & morceau)
43 {
44     std::string mot {};
45     std::ostringstream flux {};
46
47     // Récupération du nom du morceau.
48     while (entree >> mot && mot != "|")
49     {
50         flux << mot << " ";
51     }
52
53     std::string nom_morceau { flux.str() };
54     if (std::empty(nom_morceau))
55     {
56         nom_morceau = "Morceau inconnu";
57     }
58     morceau.nom = traitement_chaine(nom_morceau);
59     flux.str(std::string {});
60
61     // Récupération du nom de l'album.
62     while (entree >> mot && mot != "|")
63     {
64         flux << mot << " ";
65     }
66
67     std::string nom_album { flux.str() };
68     if (std::empty(nom_album))
69     {
```

III. On passe la deuxième!

```
70     nom_album = "Album inconnu";
71 }
72 morceau.album.nom = traitement_chaine(nom_album);
73 flux.str(std::string {});
74
75 // Récupération du nom de l'artiste.
76 while (entree >> mot)
77 {
78     flux << mot << " ";
79 }
80
81 std::string nom_artiste { flux.str() };
82 if (std::empty(nom_artiste))
83 {
84     nom_artiste = "Artiste inconnu";
85 }
86 morceau.compositeur.nom = traitement_chaine(nom_artiste);
87 flux.str(std::string {});
88
89 return entree;
90 }
```

Remarquez l'utilisation des flux de chaînes de caractères pour gagner en clarté et en élégance. Ils seront beaucoup utilisés dans ce T.P.

Vous pouvez aussi voir que l'on a défini une fonction en plus, la fonction `traitement_chaine`: c'est une fonction utilitaire créée pour éviter la répétition de code. [Retourner au texte.](#)

Contenu masqué n°50 : Tests unitaires

```
1 // Test qu'une entrée complète est valide.
2 void test_creation_morceau_entree_complete()
3 {
4     std::istringstream entree { "Frica | Frica | Carla's Dreams" };
5     Morceau morceau {};
6
7     entree >> morceau;
8
9     assert(morceau.nom == "Frica" &&
10            u8"Le nom du morceau doit être Frica.");
11     assert(morceau.album.nom == "Frica" &&
12            u8"Le nom de l'album doit être Frica.");
13     assert(morceau.compositeur.nom == "Carla's Dreams" &&
14            u8"Le nom du compositeur doit être Carla's Dreams.");
15 }
```

III. On passe la deuxième!

```
13
14 // Test d'une entrée complète avec beaucoup d'espaces.
15 void test_creation_morceau_entree_espaces_partout()
16 {
17     std::istringstream entree {
18         "Levels | Levels | Avicii" };
19     Morceau morceau {};
20
21     entree >> morceau;
22
23     assert(morceau.nom == "Levels" &&
24         u8"Le nom du morceau doit être Levels.");
25     assert(morceau.album.nom == "Levels" &&
26         u8"Le nom de l'album doit être Levels.");
27     assert(morceau.compositeur.nom == "Avicii" &&
28         u8"Le nom du compositeur doit être Avicii.");
29 }
30
31 // Test d'une entrée avec seulement le nom de la chanson est
32 // valide.
33 void test_creation_morceau_entree_chanson_artiste()
34 {
35     std::istringstream entree {
36         "Subeme la radio | | Enrique Iglesias" };
37     Morceau morceau {};
38
39     entree >> morceau;
40
41     assert(morceau.nom == "Subeme la radio" &&
42         u8"Le nom du morceau doit être Subeme la radio.");
43     assert(morceau.album.nom == "Album inconnu" &&
44         u8"Le nom de l'album doit être Album inconnu.");
45     assert(morceau.compositeur.nom == "Enrique Iglesias" &&
46         u8"Le nom du compositeur doit être Enrique Iglesias.");
47 }
48
49 // Test d'une entrée avec seulement le nom de la chanson.
50 void test_creation_morceau_entree_chanson_uniquement()
51 {
52     std::istringstream entree { "Duel of the fates | |" };
53     Morceau morceau {};
54
55     entree >> morceau;
56
57     assert(morceau.nom == "Duel of the fates" &&
58         u8"Le nom du morceau doit être Duel of the Fates.");
59     assert(morceau.album.nom == "Album inconnu" &&
60         u8"Le nom de l'album doit être Album inconnu.");
61     assert(morceau.compositeur.nom == "Artiste inconnu" &&
62         u8"Le nom du compositeur doit être Artiste inconnu.");
63 }
```

III. On passe la deuxième!

```
51 }
52
53 // Test d'une entrée vide.
54 void test_creation_morceau_entree_vide()
55 {
56     std::istringstream entree { "| |" };
57     Morceau morceau {};
58
59     entree >> morceau;
60
61     assert(morceau.nom == "Morceau inconnu" &&
62            u8"Le nom du morceau doit être Morceau inconnu.");
63     assert(morceau.album.nom == "Album inconnu" &&
64            u8"Le nom de l'album doit être Album inconnu.");
65     assert(morceau.compositeur.nom == "Artiste inconnu" &&
66            u8"Le nom du compositeur doit être Artiste inconnu.");
67 }
```

[Retourner au texte.](#)

Contenu masqué n°51 : Implémentation

```
1 void enregistrement(Discographie const & discographie, std::string
   const & nom_fichier)
2 {
3     std::ofstream fichier { nom_fichier };
4     if (!fichier)
5     {
6         throw
7             std::runtime_error("Impossible d'ouvrir le fichier en écriture.");
8     }
9     for (Morceau const & morceau : discographie)
10    {
11        fichier << morceau << std::endl;
12    }
13 }
14
15 void chargement(Discographie & discographie, std::string const &
   nom_fichier)
16 {
17     std::ifstream fichier { nom_fichier };
18     if (!fichier)
19     {
```


III. On passe la deuxième!

```
20         throw
           std::runtime_error("Impossible d'ouvrir le fichier en lecture.");
21     }
22
23     std::string ligne {};
24     while (std::getline(fichier, ligne))
25     {
26         Morceau morceau {};
27
28         std::istringstream flux { ligne };
29         flux >> morceau;
30
31         discographie.push_back(morceau);
32     }
33 }
```

[Retourner au texte.](#)

Contenu masqué n°52 : Prototypes

```
1 void tri_morceau(Discographie& discographie);
2 void tri_album(Discographie& discographie);
3 void tri_artiste(Discographie& discographie);
4
5 void affichage_morceau(Discographie const & discographie)
6 void affichage_album(Discographie const & discographie);
7 void affichage_artiste(Discographie const & discographie);
```

[Retourner au texte.](#)

Contenu masqué n°53 : Implémentation

```
1 void tri_morceau(Discographie& discographie)
2 {
3     std::sort(std::begin(discographie), std::end(discographie),
4               [](Morceau const & lhs, Morceau const & rhs)
5               {
6                   return lhs.nom < rhs.nom;
7               });
7 }
```

III. On passe la deuxième!

```
8
9 void tri_album(Discographie& discographie)
10 {
11     std::sort(std::begin(discographie), std::end(discographie),
12               [](Morceau const & lhs, Morceau const & rhs)
13               {
14                 if (lhs.album.nom < rhs.album.nom)
15                     return true;
16
17                 return lhs.album.nom == rhs.album.nom && lhs.nom <
18                     rhs.nom;
19             });
20 }
21
22 void tri_artiste(Discographie& discographie)
23 {
24     std::sort(std::begin(discographie), std::end(discographie),
25               [](Morceau const & lhs, Morceau const & rhs)
26               {
27                 if (lhs.compositeur.nom < rhs.compositeur.nom)
28                     return true;
29
30                 if (lhs.compositeur.nom == rhs.compositeur.nom)
31                 {
32                     if (lhs.album.nom < rhs.album.nom)
33                         return true;
34
35                     return lhs.album.nom == rhs.album.nom &&
36                         lhs.nom < rhs.nom;
37                 }
38
39                 return false;
40             });
41 }
42
43 void affichage_morceau(Discographie const & discographie)
44 {
45     for (Morceau const & morceau : discographie)
46     {
47         std::cout << "--> " << morceau << std::endl;
48     }
49 }
50
51 void affichage_album(Discographie const & discographie)
52 {
53     Album album_precedent{};
54     for (Morceau const & morceau : discographie)
55     {
56         if (morceau.album.nom != album_precedent.nom)
57         {
```

III. On passe la deuxième!

```
54         std::cout << "--> " << morceau.album <<
55         " | " << morceau.compositeur <<
56         std::endl;
57     }
58     std::cout << "\t/--> " << morceau.nom << std::endl;
59     album_precedent = morceau.album;
60 }
61
62 void affichage_artiste(Discographie const & discographie)
63 {
64     Artiste artiste_precedent{};
65     Album album_precedent{};
66
67     for (Morceau const & morceau : discographie)
68     {
69         if (morceau.compositeur.nom !=
70             artiste_precedent.nom)
71         {
72             std::cout << "--> " << morceau.compositeur
73             << std::endl;
74         }
75         if (morceau.album.nom != album_precedent.nom)
76         {
77             std::cout << "\t/--> " << morceau.album <<
78             std::endl;
79         }
80
81         std::cout << "\t\t/--> " << morceau.nom <<
82         std::endl;
83
84         artiste_precedent = morceau.compositeur;
85         album_precedent = morceau.album;
86     }
87 }
```

[Retourner au texte.](#)

Contenu masqué n°54 :

Prototypes

```
1 enum class Affichage { Artiste, Album, Morceau };
2
3 void affichage(Discographie& discographie, Affichage
    type_affichage);
```

[Retourner au texte.](#)

Contenu masqué n°55 : Implémentation

```
1 // L'ensemble des possibilités d'affichage de la discographie.
2 enum class Affichage { Artiste, Album, Morceau };
3
4 // Fonction liant ce qui précède pour gérer toute la logique
  d'affichage.
5 void affichage(Discographie& discographie, Affichage
    type_affichage)
6 {
7     if (type_affichage == Affichage::Album)
8     {
9         tri_album(discographie);
10        affichage_album(discographie);
11    }
12    else if (type_affichage == Affichage::Artiste)
13    {
14        tri_artiste(discographie);
15        affichage_artiste(discographie);
16    }
17    else if (type_affichage == Affichage::Morceau)
18    {
19        tri_morceau(discographie);
20        affichage_morceau(discographie);
21    }
22    else
23    {
24        // Par exemple si on met à jour l'énumération mais
        qu'on oublie d'ajouter la fonction
        correspondante.
25        throw
            std::runtime_error("Commande d'affichage inconnue.");
26    }
27 }
```

III. On passe la deuxième!

On utilise ici une exception qui ne devrait en principe jamais être lancée, car nous ne tomberons jamais dans le `else`. Mais si, plus tard, on modifiait le programme pour ajouter une nouvelle forme d’affichage, sans mettre à jour cette partie du code, l’exception nous le rappellera.
[Retourner au texte.](#)

Contenu masqué n°56 : Prototypes

```
1 // L'ensemble des actions qu'il est possible de faire.
2 enum class Commande { Afficher, Ajouter, Enregistrer, Charger,
   Quitter };
3
4 // Affichage et récupération de ce l'utilisateur écrit.
5 std::string recuperer_commande();
6 // Analyse du texte reçu en entrée.
7 std::tuple<Commande, std::string> analyser_commande(std::string
   const & commande_texte);
8 // La commande à exécuter ainsi que la suite des instructions.
9 bool executer_commande(Discographie & discographie, Commande
   commande, std::string const & instructions);
```

Remarquez la création d’une énumération pour représenter les différents types de commande; ce n’est pas nécessaire, c’est juste un type de base de plus pour pouvoir lire élégamment le type de commande dans la console.
[Retourner au texte.](#)

Contenu masqué n°57 : Implémentations

```
1 // Récupération de ce que l'utilisateur écrit.
2 std::string recuperer_commande()
3 {
4     std::cout << "> ";
5
6     std::string commande {};
7     std::getline(std::cin, commande);
8     return commande;
9 }
10
11 // L'ensemble des actions qu'il est possible de faire.
12 enum class Commande { Afficher, Ajouter, Enregistrer, Charger,
   Quitter };
13
14 // Analyse du texte reçu en entrée.
```

III. On passe la deuxième!

```
15 std::tuple<Commande, std::string> analyser_commande(std::string
    const & commande_texte)
16 {
17     // Pour traiter la chaîne comme étant un flux, afin d'en
        extraire les données.
18     std::istringstream flux { commande_texte };
19
20     std::string premier_mot {};
21     flux >> premier_mot;
22     premier_mot = traitement_chaine(premier_mot);
23
24     std::string instructions {};
25     std::getline(flux, instructions);
26     instructions = traitement_chaine(instructions);
27
28     if (premier_mot == "afficher")
29     {
30         // Le mode d'affichage.
31         return { Commande::Afficher, instructions };
32     }
33     else if (premier_mot == "ajouter")
34     {
35         // Les informations à ajouter.
36         return { Commande::Ajouter, instructions };
37     }
38     else if (premier_mot == "enregistrer")
39     {
40         // Le fichier à écrire.
41         return{ Commande::Enregistrer, instructions };
42     }
43     else if (premier_mot == "charger")
44     {
45         // Le fichier à lire.
46         return { Commande::Charger, instructions };
47     }
48     else if (premier_mot == "quitter")
49     {
50         // Chaîne vide, car on quitte le programme sans autre
            instruction.
51         return { Commande::Quitter, std::string {} };
52     }
53     else
54     {
55         // On a reçu du texte qui est incorrect.
56         throw std::runtime_error("Commande invalide.");
57     }
58 }
59
60 // La commande à exécuter ainsi que la suite des instructions.
```

III. On passe la deuxième!

```
61 bool executer_commande(Discographie & discographie, Commande
    commande, std::string const & instructions)
62 {
63     if (commande == Commande::Afficher)
64     {
65         if (instructions == "artistes")
66         {
67             affichage(discographie, Affichage::Artiste);
68         }
69         else if (instructions == "albums")
70         {
71             affichage(discographie, Affichage::Album);
72         }
73         else if (instructions == "morceaux")
74         {
75             affichage(discographie, Affichage::Morceau);
76         }
77         else
78         {
79             // Si on se trouve ici, alors c'est qu'il y a une
                erreur dans les instructions d'affichage.
80             throw
                std::runtime_error("Commande d'affichage inconnue.");
81         }
82     }
83     else if (commande == Commande::Ajouter)
84     {
85         std::istringstream flux { instructions };
86         Morceau morceau {};
87         flux >> morceau;
88         discographie.push_back(morceau);
89     }
90     else if (commande == Commande::Charger)
91     {
92         chargement(discographie, instructions);
93     }
94     else if (commande == Commande::Enregistrer)
95     {
96         enregistrement(discographie, instructions);
97     }
98     else if (commande == Commande::Quitter)
99     {
100         // Plus rien à faire, on quitte.
101         return false;
102     }
103
104     return true;
105 }
```

Comme on l'a dit, on utilise les exceptions pour faire remonter les erreurs en cas de commande

III. On passe la deuxième!

invalide. C'est un usage logique des exceptions, puisque les erreurs de commande de l'utilisateur sont indépendantes de la volonté du programmeur. Les exceptions sont donc toutes indiquées pour gérer ce genre de cas. [Retourner au texte.](#)

Contenu masqué n°58 : Fonction principale

```
1 int main()
2 {
3     // Lancement préalable des tests unitaires.
4     test_creation_morceau_entree_complete();
5     test_creation_morceau_entree_espaces_partout();
6     test_creation_morceau_entree_chanson_artiste();
7     test_creation_morceau_entree_chanson_uniquement();
8     test_creation_morceau_entree_vide();
9
10    // On préremplit la discographie.
11    Artiste const dave_brubeck { "Dave Brubeck" };
12    Artiste const secret_garden { "Secret Garden" };
13    Artiste const indochine { "Indochine" };
14
15    Album const time_out { "Time Out" };
16    Album const songs_from_a_secret_garden {
17        "Songs from a Secret Garden" };
18    Album const l_aventurier { "L'aventurier" };
19    Album const paradize { "Paradize" };
20
21    Morceau const take_five { "Take Five", dave_brubeck, time_out
22        };
23    Morceau const blue_rondo_a_la_turk { "Blue Rondo a la Turk",
24        dave_brubeck, time_out };
25    Morceau const nocturne { "Nocturne", secret_garden,
26        songs_from_a_secret_garden };
27    Morceau const aventurier { "L'aventurier", indochine,
28        l_aventurier };
29    Morceau const j_ai_demande_a_la_lune {
30        "J'ai demandé à la lune", indochine, paradize };
31
32    Discographie discographie { take_five, blue_rondo_a_la_turk,
33        nocturne, aventurier, j_ai_demande_a_la_lune };
34
35    bool continuer { true };
36    do
37    {
38        try
39        {
40            std::string entree { recuperer_commande() };
41        }
42        catch (...)
43        {
44            continuer = false;
45        }
46    } while (continuer);
47}
```


III. On passe la deuxième!

```
34         auto[commande, instructions] =
35             analyser_commande(entree);
36         instructions = traitement_chaine(instructions);
37         continuer = executer_commande(discographie, commande,
38             instructions);
39     }
40     catch (std::runtime_error const & exception)
41     {
42         std::cout << "Erreur : " << exception.what() <<
43             std::endl;
44     }
45     } while (continuer);
46     return 0;
47 }
```

Il ne faut pas oublier de gérer les erreurs remontées par notre programme. Ici ce n'est pas compliqué, il suffit d'expliquer l'erreur à l'utilisateur et de lui redemander une entrée. [Retourner au texte.](#)

Contenu masqué n°59 : Corrigé complet

```
1  #include <algorithm>
2  #include <cassert>
3  #include <cctype>
4  #include <fstream>
5  #include <iostream>
6  #include <sstream>
7  #include <string>
8  #include <tuple>
9  #include <vector>
10
11 std::string traitement_chaine(std::string const & chaine)
12 {
13     std::string copie{ chaine };
14
15     // D'abord on enlève les espaces au début...
16     auto premier_non_espace{
17         std::find_if_not(std::begin(copie), std::end(copie),
18             isspace) };
19     copie.erase(std::begin(copie), premier_non_espace);
20
21     // ...puis à la fin.
22     std::reverse(std::begin(copie), std::end(copie));
23 }
```

III. On passe la deuxième!

```
21     premier_non_espace = std::find_if_not(std::begin(copie),
22         std::end(copie), isspace);
23     copie.erase(std::begin(copie), premier_non_espace);
24     std::reverse(std::begin(copie), std::end(copie));
25     return copie;
26 }
27
28 struct Artiste
29 {
30     std::string nom;
31 };
32
33 // On surcharge l'opérateur << pour être capable d'afficher le type
34 // Artiste.
35 std::ostream & operator<<(std::ostream & sortie, Artiste const &
36     artiste)
37 {
38     sortie << artiste.nom;
39     return sortie;
40 }
41
42 struct Album
43 {
44     std::string nom;
45 };
46
47 // On surcharge l'opérateur << pour être capable d'afficher le type
48 // Album.
49 std::ostream & operator<<(std::ostream & sortie, Album const &
50     album)
51 {
52     sortie << album.nom;
53     return sortie;
54 }
55
56 struct Morceau
57 {
58     std::string nom;
59     Artiste compositeur;
60     Album album;
61 };
62
63 // On surcharge l'opérateur << pour être capable d'afficher le type
64 // Morceau.
65 std::ostream & operator<<(std::ostream & sortie, Morceau const &
66     morceau)
67 {
68     sortie << morceau.nom << " | " << morceau.album << " | " <<
69         morceau.compositeur;
```

III. On passe la deuxième!

```
63         return sortie;
64     }
65
66     // Conversion d'un flux d'entrée (std::cin, chaîne, fichier) en
        type Morceau.
67     std::istream & operator>>(std::istream & entree, Morceau & morceau)
68     {
69         std::string mot{};
70         std::ostringstream flux{};
71
72         // Récupération du nom du morceau.
73         while (entree >> mot && mot != "|")
74         {
75             flux << mot << " ";
76         }
77
78         std::string nom_morceau{ flux.str() };
79         if (std::empty(nom_morceau))
80         {
81             nom_morceau = "Morceau inconnu";
82         }
83         morceau.nom = traitement_chaine(nom_morceau);
84         flux.str(std::string{});
85
86         // Récupération du nom de l'album.
87         while (entree >> mot && mot != "|")
88         {
89             flux << mot << " ";
90         }
91
92         std::string nom_album{ flux.str() };
93         if (std::empty(nom_album))
94         {
95             nom_album = "Album inconnu";
96         }
97         morceau.album.nom = traitement_chaine(nom_album);
98         flux.str(std::string{});
99
100        // Récupération du nom de l'artiste.
101        while (entree >> mot)
102        {
103            flux << mot << " ";
104        }
105
106        std::string nom_artiste{ flux.str() };
107        if (std::empty(nom_artiste))
108        {
109            nom_artiste = "Artiste inconnu";
110        }
111        morceau.compositeur.nom = traitement_chaine(nom_artiste);
```

III. On passe la deuxième!

```
112     flux.str(std::string{});
113
114     return entree;
115 }
116
117 // Test qu'une entrée complète est valide.
118 void test_creation_morceau_entree_complexe()
119 {
120     std::istringstream entree{ "Frica | Fricka | Carla's Dreams"
121                               };
122     Morceau morceau{};
123
124     entree >> morceau;
125
126     assert(morceau.nom == "Frica" &&
127            u8"Le nom du morceau doit être Fricka.");
128     assert(morceau.album.nom == "Fricka" &&
129            u8"Le nom de l'album doit être Fricka.");
130     assert(morceau.compositeur.nom == "Carla's Dreams" &&
131            u8"Le nom du compositeur doit être Carla's Dreams.");
132 }
133
134 // Test d'une entrée complète avec beaucoup d'espaces.
135 void test_creation_morceau_entree_espaces_partout()
136 {
137     std::istringstream entree{
138         "Levels | Levels | Avicii" };
139     Morceau morceau{};
140
141     entree >> morceau;
142
143     assert(morceau.nom == "Levels" &&
144            u8"Le nom du morceau doit être Levels.");
145     assert(morceau.album.nom == "Levels" &&
146            u8"Le nom de l'album doit être Levels.");
147     assert(morceau.compositeur.nom == "Avicii" &&
148            u8"Le nom du compositeur doit être Avicii.");
149 }
150
151 // Test d'une entrée avec seulement le nom de la chanson est
152 // valide.
153 void test_creation_morceau_entree_chanson_artiste()
154 {
155     std::istringstream entree{
156         "Subeme la radio | | Enrique Iglesias" };
157     Morceau morceau{};
158
159     entree >> morceau;
```

III. On passe la deuxième!

```
151     assert(morceau.nom == "Subeme la radio" &&
152            u8"Le nom du morceau doit être Subeme la radio.");
153     assert(morceau.album.nom == "Album inconnu" &&
154            u8"Le nom de l'album doit être Album inconnu.");
155     assert(morceau.compositeur.nom == "Enrique Iglesias" &&
156            u8"Le nom du compositeur doit être Enrique Iglesias.");
157 }
158 // Test d'une entrée avec seulement le nom de la chanson.
159 void test_creation_morceau_entree_chanson_uniquement()
160 {
161     std::istringstream entree{ "Duel of the fates | |" };
162     Morceau morceau{};
163
164     entree >> morceau;
165
166     assert(morceau.nom == "Duel of the fates" &&
167            u8"Le nom du morceau doit être Duel of the Fates.");
168     assert(morceau.album.nom == "Album inconnu" &&
169            u8"Le nom de l'album doit être Album inconnu.");
170     assert(morceau.compositeur.nom == "Artiste inconnu" &&
171            u8"Le nom du compositeur doit être Artiste inconnu.");
172 }
173 // Test d'une entrée vide.
174 void test_creation_morceau_entree_vide()
175 {
176     std::istringstream entree{ "| |" };
177     Morceau morceau{};
178
179     entree >> morceau;
180
181     assert(morceau.nom == "Morceau inconnu" &&
182            u8"Le nom du morceau doit être Morceau inconnu.");
183     assert(morceau.album.nom == "Album inconnu" &&
184            u8"Le nom de l'album doit être Album inconnu.");
185     assert(morceau.compositeur.nom == "Artiste inconnu" &&
186            u8"Le nom du compositeur doit être Artiste inconnu.");
187 }
188 using Discographie = std::vector<Morceau>;
189
190 void tri_morceau(Discographie& discographie)
191 {
192     std::sort(std::begin(discographie), std::end(discographie),
193               [](Morceau const & lhs, Morceau const & rhs)
194               {
195                 return lhs.nom < rhs.nom;
196             });
197 }
```

III. On passe la deuxième!

```
191
192 void tri_album(Discographie& discographie)
193 {
194     std::sort(std::begin(discographie), std::end(discographie),
195               [](Morceau const & lhs, Morceau const & rhs)
196               {
197                 if (lhs.album.nom < rhs.album.nom)
198                     return true;
199
200                 return lhs.album.nom == rhs.album.nom && lhs.nom <
201                     rhs.nom;
202             });
203 }
204
205 void tri_artiste(Discographie& discographie)
206 {
207     std::sort(std::begin(discographie), std::end(discographie),
208               [](Morceau const & lhs, Morceau const & rhs)
209               {
210                 if (lhs.compositeur.nom < rhs.compositeur.nom)
211                     return true;
212
213                 if (lhs.compositeur.nom == rhs.compositeur.nom)
214                 {
215                     if (lhs.album.nom < rhs.album.nom)
216                         return true;
217
218                     return lhs.album.nom == rhs.album.nom &&
219                         lhs.nom < rhs.nom;
220                 }
221
222                 return false;
223             });
224 }
225
226 // Affichage de la discographie par morceau, dans l'ordre
227 // alphabétique.
228 void affichage_morceau(Discographie const & discographie)
229 {
230     for (Morceau const & morceau : discographie)
231     {
232         std::cout << "--> " << morceau << std::endl;
233     }
234 }
235
236 // Affichage de la discographie par album, dans l'ordre
237 // alphabétique.
238 void affichage_album(Discographie const & discographie)
239 {
240     Album album_precedent{};
```

III. On passe la deuxième!

```
235     for (Morceau const & morceau : discographie)
236     {
237         if (morceau.album.nom != album_precedent.nom)
238         {
239             std::cout << "--> " << morceau.album <<
                " | " << morceau.compositeur <<
                std::endl;
240         }
241         std::cout << "\t/--> " << morceau.nom << std::endl;
242
243         album_precedent = morceau.album;
244     }
245 }
246
247 // Affichage de la discographie par artiste, dans l'ordre
    alphabétique.
248 void affichage_artiste(Discographie const & discographie)
249 {
250     Artiste artiste_precedent{};
251     Album album_precedent{};
252
253     for (Morceau const & morceau : discographie)
254     {
255         if (morceau.compositeur.nom !=
                artiste_precedent.nom)
256         {
257             std::cout << "--> " << morceau.compositeur
                << std::endl;
258         }
259
260         if (morceau.album.nom != album_precedent.nom)
261         {
262             std::cout << "\t/--> " << morceau.album <<
                std::endl;
263         }
264
265         std::cout << "\t\t/--> " << morceau.nom <<
                std::endl;
266
267         artiste_precedent = morceau.compositeur;
268         album_precedent = morceau.album;
269     }
270 }
271
272 // L'ensemble des possibilités d'affichage de la discographie.
273 enum class Affichage { Artiste, Album, Morceau };
274
275 void affichage(Discographie& discographie, Affichage
    type_affichage)
276 {
```

III. On passe la deuxième!

```
277     if (type_affichage == Affichage::Album)
278     {
279         tri_album(discographie);
280         affichage_album(discographie);
281     }
282     else if (type_affichage == Affichage::Artiste)
283     {
284         tri_artiste(discographie);
285         affichage_artiste(discographie);
286     }
287     else if (type_affichage == Affichage::Morceau)
288     {
289         tri_morceau(discographie);
290         affichage_morceau(discographie);
291     }
292     else
293     {
294         // Par exemple si on met à jour l'énumération mais
295         // qu'on oublie d'ajouter la fonction
296         // correspondante.
297         throw
298             std::runtime_error("Commande d'affichage inconnue.");
299     }
300 }
301
302 void enregistrement(Discographie const & discographie, std::string
303     const & nom_fichier)
304 {
305     std::ofstream fichier{ nom_fichier };
306     if (!fichier)
307     {
308         throw
309             std::runtime_error("Impossible d'ouvrir le fichier en écri
310 ");
311     }
312
313     for (Morceau const & morceau : discographie)
314     {
315         fichier << morceau << std::endl;
316     }
317 }
318
319 void chargement(Discographie & discographie, std::string const &
320     nom_fichier)
321 {
322     std::ifstream fichier{ nom_fichier };
323     if (!fichier)
324     {
325         throw
326             std::runtime_error("Impossible d'ouvrir le fichier en lect
327 ");
328     }
329 }
```


III. On passe la deuxième!

```
320
321     std::string ligne{};
322     while (std::getline(fichier, ligne))
323     {
324         Morceau morceau{};
325
326         std::istringstream flux{ ligne };
327         flux >> morceau;
328
329         discographie.push_back(morceau);
330     }
331 }
332
333 // Récupération de ce l'utilisateur écrit.
334 std::string recuperer_commande()
335 {
336     std::cout << "> ";
337
338     std::string commande{};
339     std::getline(std::cin, commande);
340     return commande;
341 }
342
343 // L'ensemble des actions qu'il est possible de faire.
344 enum class Commande { Afficher, Ajouter, Enregistrer, Charger,
345     Quitter };
346
347 // Analyse du texte reçu en entrée.
348 std::tuple<Commande, std::string> analyser_commande(std::string
349     const & commande_texte)
350 {
351     // Pour traiter la chaîne comme étant un flux, afin d'en
352     // extraire les données.
353     std::istringstream flux{ commande_texte };
354
355     std::string premier_mot{};
356     flux >> premier_mot;
357     premier_mot = traitement_chaine(premier_mot);
358
359     std::string instructions{};
360     std::getline(flux, instructions);
361     instructions = traitement_chaine(instructions);
362
363     if (premier_mot == "afficher")
364     {
365         // Le mode d'affichage.
366         return { Commande::Afficher, instructions };
367     }
368     else if (premier_mot == "ajouter")
369     {
370         // Le mode d'ajout.
371         return { Commande::Ajouter, instructions };
372     }
373     else if (premier_mot == "charger")
374     {
375         // Le mode de chargement.
376         return { Commande::Charger, instructions };
377     }
378     else if (premier_mot == "enregistrer")
379     {
380         // Le mode d'enregistrement.
381         return { Commande::Enregistrer, instructions };
382     }
383     else if (premier_mot == "quitter")
384     {
385         // Le mode de sortie.
386         return { Commande::Quitter, instructions };
387     }
388     else
389     {
390         // Commande inconnue.
391         return { Commande::Quitter, instructions };
392     }
393 }
```

III. On passe la deuxième!

```
367         // Les informations à ajouter.
368         return { Commande::Ajouter, instructions };
369     }
370     else if (premier_mot == "enregistrer")
371     {
372         // Le fichier à écrire.
373         return{ Commande::Enregistrer, instructions };
374     }
375     else if (premier_mot == "charger")
376     {
377         // Le fichier à lire.
378         return { Commande::Charger, instructions };
379     }
380     else if (premier_mot == "quitter")
381     {
382         // Chaîne vide, car on quitte le programme sans
383         // autre instruction.
384         return { Commande::Quitter, std::string {} };
385     }
386     else
387     {
388         // On a reçu du texte qui est incorrect.
389         throw std::runtime_error("Commande invalide.");
390     }
391 }
392 // La commande à exécuter ainsi que la suite des instructions.
393 bool executer_commande(Discographie & discographie, Commande
394     commande, std::string const & instructions)
395 {
396     if (commande == Commande::Afficher)
397     {
398         if (instructions == "artistes")
399         {
400             affichage(discographie,
401                 Affichage::Artiste);
402         }
403         else if (instructions == "albums")
404         {
405             affichage(discographie, Affichage::Album);
406         }
407         else if (instructions == "morceaux")
408         {
409             affichage(discographie,
410                 Affichage::Morceau);
411         }
412     }
413     else
414     {
415     }
```

III. On passe la deuxième!

```
411         // Si on se trouve ici, alors c'est qu'il y
412         // a une erreur dans les instructions
413         // d'affichage.
414         throw
415         std::runtime_error("Commande d'affichage inconnue.");
416     }
417 }
418 else if (commande == Commande::Ajouter)
419 {
420     std::istringstream flux{ instructions };
421     Morceau morceau{};
422     flux >> morceau;
423     discographie.push_back(morceau);
424 }
425 else if (commande == Commande::Charger)
426 {
427     chargement(discographie, instructions);
428 }
429 else if (commande == Commande::Enregistrer)
430 {
431     enregistrement(discographie, instructions);
432 }
433 else if (commande == Commande::Quitter)
434 {
435     // Plus rien à faire, on quitte.
436     return false;
437 }
438
439 return true;
440 }
441
442 int main()
443 {
444     // Lancement préalable des tests unitaires.
445     test_creation_morceau_entree_complete();
446     test_creation_morceau_entree_espaces_partout();
447     test_creation_morceau_entree_chanson_artiste();
448     test_creation_morceau_entree_chanson_uniquement();
449     test_creation_morceau_entree_vide();
450
451     // On préremplit la discographie.
452     Artiste const dave_brubeck{ "Dave Brubeck " };
453     Artiste const secret_garden{ "Secret Garden" };
454     Artiste const indochine{ "Indochine" };
455
456     Album const time_out{ "Time Out" };
457     Album const songs_from_a_secret_garden{
458         "Songs from a Secret Garden" };
459     Album const l_aventurier{ "L'aventurier" };
460     Album const paradise{ "Paradize" };
```

III. On passe la deuxième!

```
457 Morceau const take_five{ "Take Five", dave_brubeck,  
458     time_out };  
459 Morceau const blue_rondo_a_la_turk{ "Blue Rondo a la Turk",  
    dave_brubeck, time_out };  
460 Morceau const nocturne{ "Nocturne", secret_garden,  
    songs_from_a_secret_garden };  
461 Morceau const aventurier{ "L'aventurier", indochine,  
    l_aventurier };  
462 Morceau const j_ai_demande_a_la_lune{  
    "J'ai demandé à la lune", indochine, paradize };  
463  
464 Discographie discographie{ take_five, blue_rondo_a_la_turk,  
    nocturne, aventurier, j_ai_demande_a_la_lune };  
465  
466 bool continuer{ true };  
467 do  
468 {  
469     try  
470     {  
471         std::string entree{ recuperer_commande() };  
472         auto[commande, instructions] =  
            analyser_commande(entree);  
473         instructions =  
            traitement_chaine(instructions);  
474         continuer = executer_commande(discographie,  
            commande, instructions);  
475     }  
476     catch (std::runtime_error const & exception)  
477     {  
478         std::cout << "Erreur : " <<  
            exception.what() << std::endl;  
479     }  
480 } while (continuer);  
481  
482 return 0;  
483  
484 }
```

[Retourner au texte.](#)

III.11. Découpons du code— Les fichiers

Lors du chapitre précédent, nous avons codé notre premier programme d'assez grande ampleur. Et avec l'ampleur que prend notre code, vient un problème qui était caché par la taille réduite de ce qu'on avait fait jusqu'alors: **l'organisation du code**.

En effet, jusqu'à maintenant, on mettait simplement tout notre code dans le fichier `main.cpp`. Le plus sophistiqué qu'on ait fait en termes d'organisation est d'utiliser les prototypes pour pouvoir reléguer les implémentations après la fonction `main` et ainsi alléger un peu la lecture du code.

Jusque-là ça suffisait, mais on voit déjà avec le TP que dès que le projet prend un peu d'ampleur, ça devient un sacré fouillis. Et encore, ça n'est rien comparé à certains gros projets. Par exemple, à l'heure où j'écris ces lignes, le code du noyau Linux comporte 17 millions de lignes!

C'est pourquoi on va maintenant apprendre **comment séparer son code en plusieurs fichiers**, ce qui va nous permettre de mieux l'organiser.

III.11.1. Le principe

Nous savons déjà séparer les prototypes de leur implémentation, ce qui est déjà une première étape vers une réorganisation. Mais ce n'est pas suffisant. Il faut maintenant séparer le code en plusieurs fichiers. Et il en existe deux types.

III.11.1.1. Le fichier d'en-tête

Les fichiers d'en-tête contiennent **les déclarations des types et fonctions que l'on souhaite créer**. Nous sommes habitués à les manipuler depuis le début de notre aventure avec C++. C'est grâce à eux que nous sommes en mesure d'utiliser la bibliothèque standard. Le concept n'est pas nouveau, on passe simplement de consommateur à créateur.

Les fichiers d'en-tête peuvent tout à fait **inclure d'autres fichiers d'en-tête**, tant standards que personnels. C'est le cas par exemple où notre fichier d'en-tête fait appel à des composants de la bibliothèque standard, comme, entre autres, `std::string` ou `std::vector`.

III. On passe la deuxième!

III.11.1.1.1. Sur Visual Studio

Ouvrez votre projet avec Visual Studio. Vous tombez sur un écran semblable au mien.

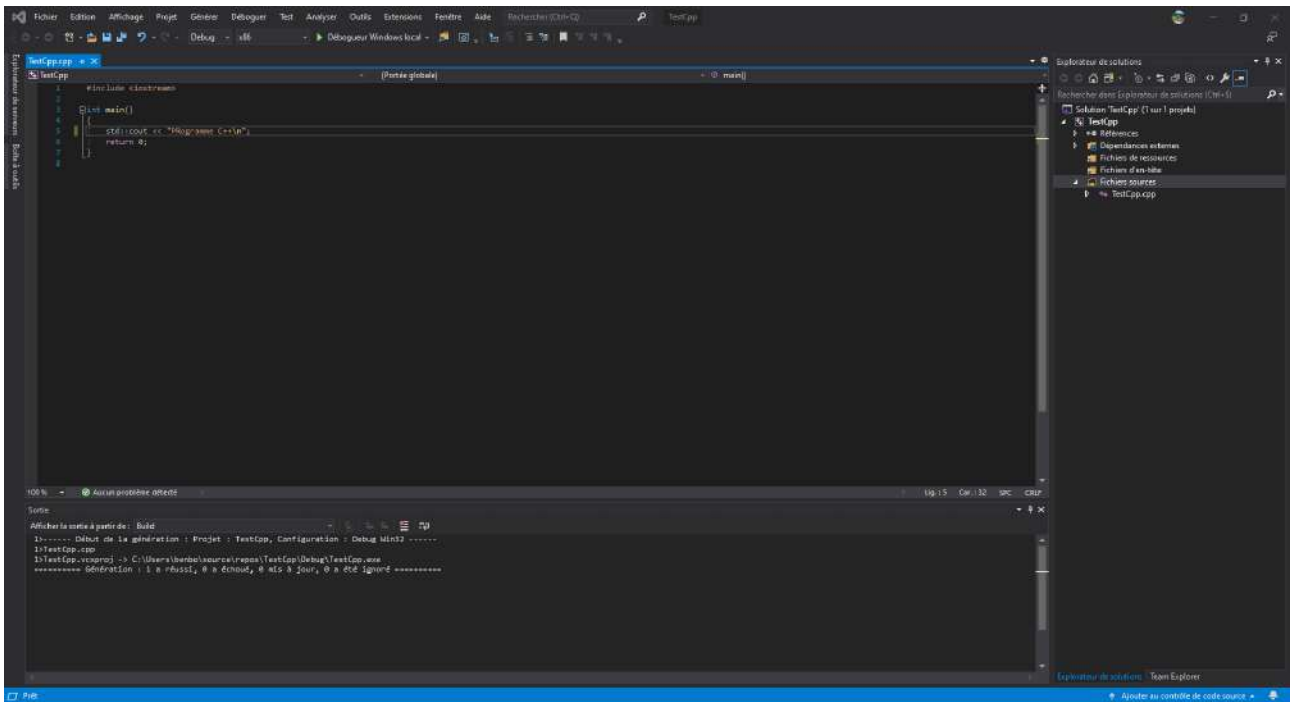


FIGURE III.11.1. – Page principale de Visual Studio avant l’ajout d’un fichier d’en-tête au projet.

Faites un clic-droit sur le nom du projet (ici, *TestCpp*), puis cliquez sur **Ajouter -> Nouvel élément...**. Ce faisant, vous arrivez sur l’écran suivant.

III. On passe la deuxième!

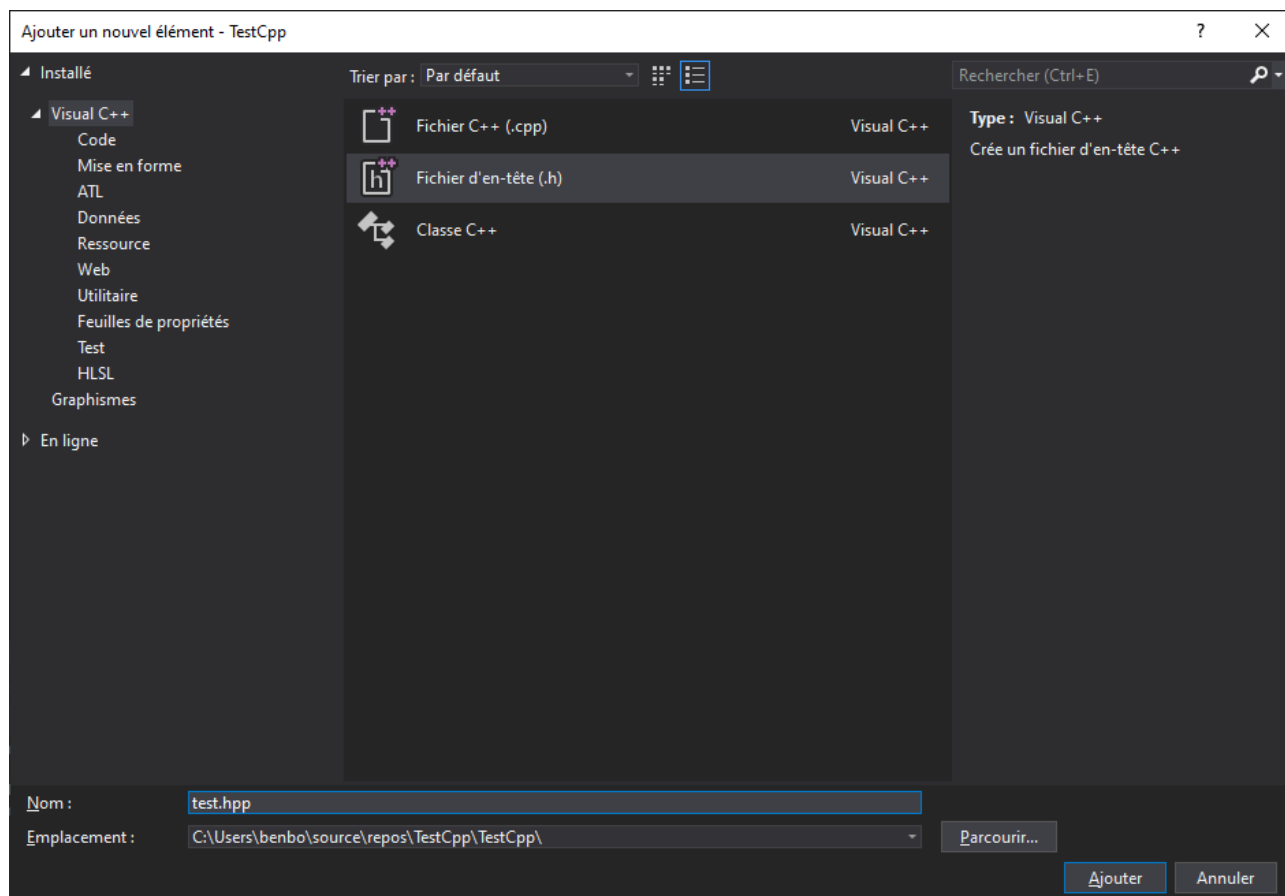


FIGURE III.11.2. – Choisissez bien «Fichier d'en-tête (.h)».

Choisissez bien **Fichier d'en-tête (.h)** puis donnez un nom à votre fichier. Ici, j'ai choisi **test.hpp**. Notez que j'ai changé l'extension, de **.h** vers **.hpp**, et je vous invite à faire de même. Enfin, cliquez sur **Ajouter**. Vous arriverez sur l'écran principal, avec le fichier ajouté au projet.

III.11.1.1.2. Sur QtCreator



Non supporté

Je vous rappelle que les explications qui suivent seront peut-être périmées ou inexactes. D'avance désolé si c'est le cas.

Ouvrez votre projet avec Qt Creator, ce qui devrait vous donner quelque chose comme moi.

III. On passe la deuxième!

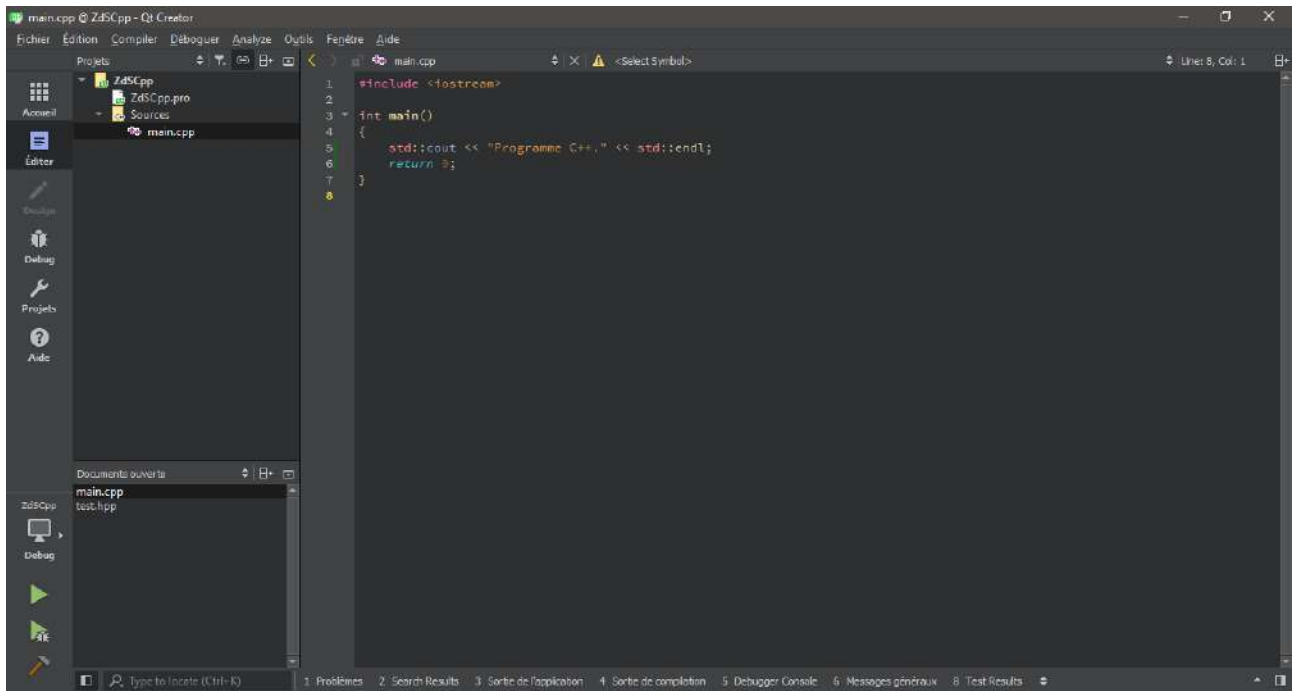


FIGURE III.11.3. – Page principale de Qt Creator avant l'ajout d'un fichier d'en-tête au projet.

Faîtes un clic-droit sur le nom du projet (ici, *ZdSCpp*), puis cliquez sur **Add new...**. Vous tombez sur la fenêtre suivante.

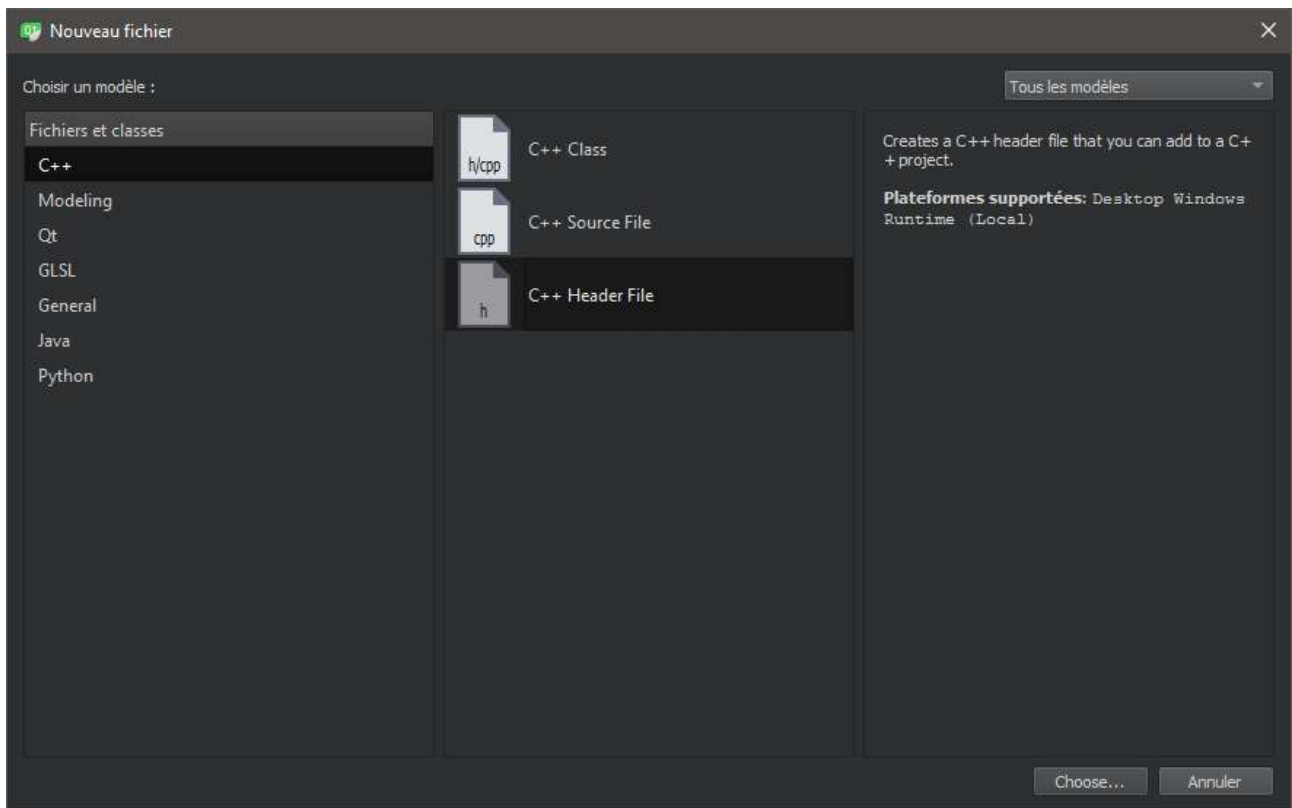


FIGURE III.11.4. – Il faut choisir «C++ Header File».

III. On passe la deuxième!

Cliquez sur **C** + **Header File**, puis sur **Choose...** en bas à droite. Vous passez au deuxième écran.

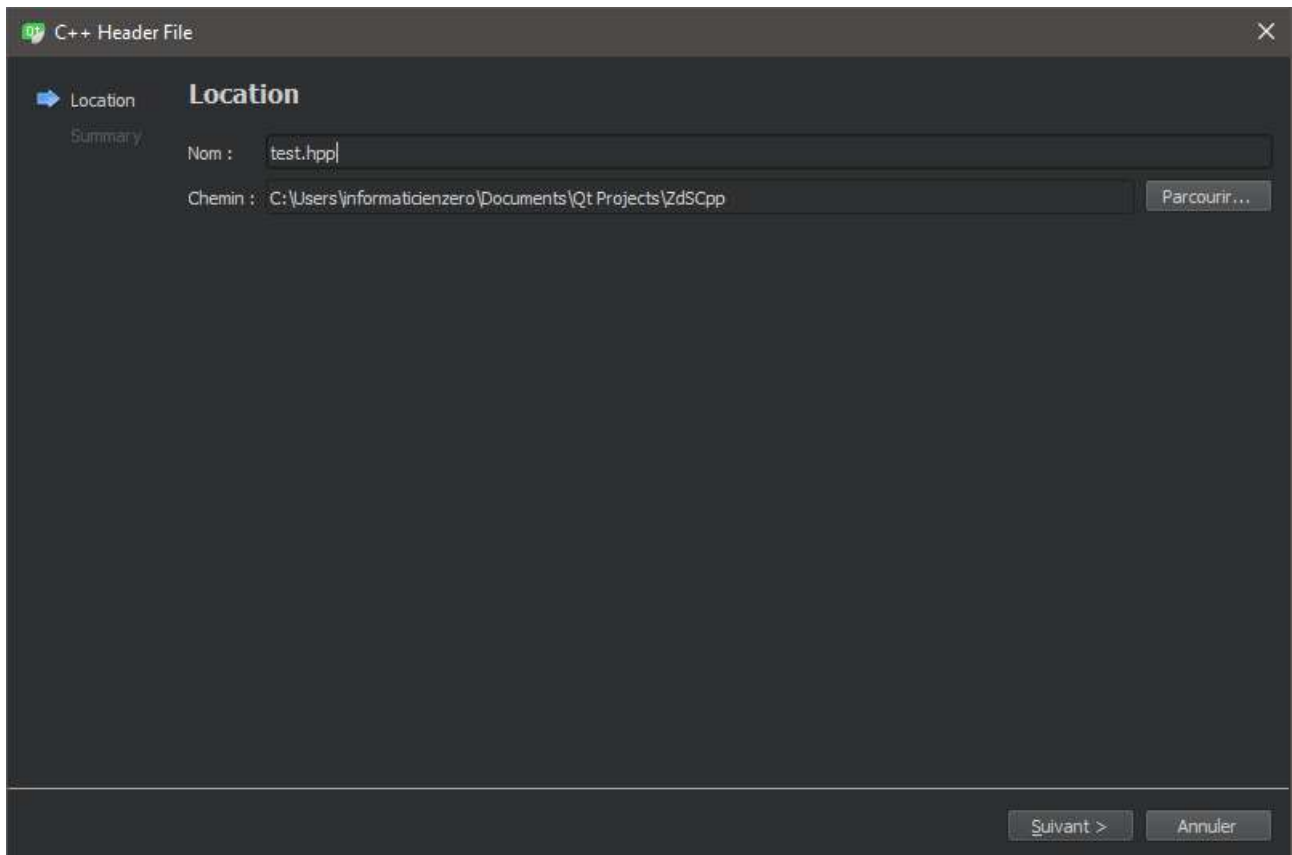


FIGURE III.11.5. – Il faut maintenant choisir un nom.

Maintenant, donnez un nom à votre fichier. Ici, pour l'exemple, je l'ai appelé **test.hpp**. Cliquez ensuite sur **Suivant** et vous arriverez sur le troisième écran.

III. On passe la deuxième!

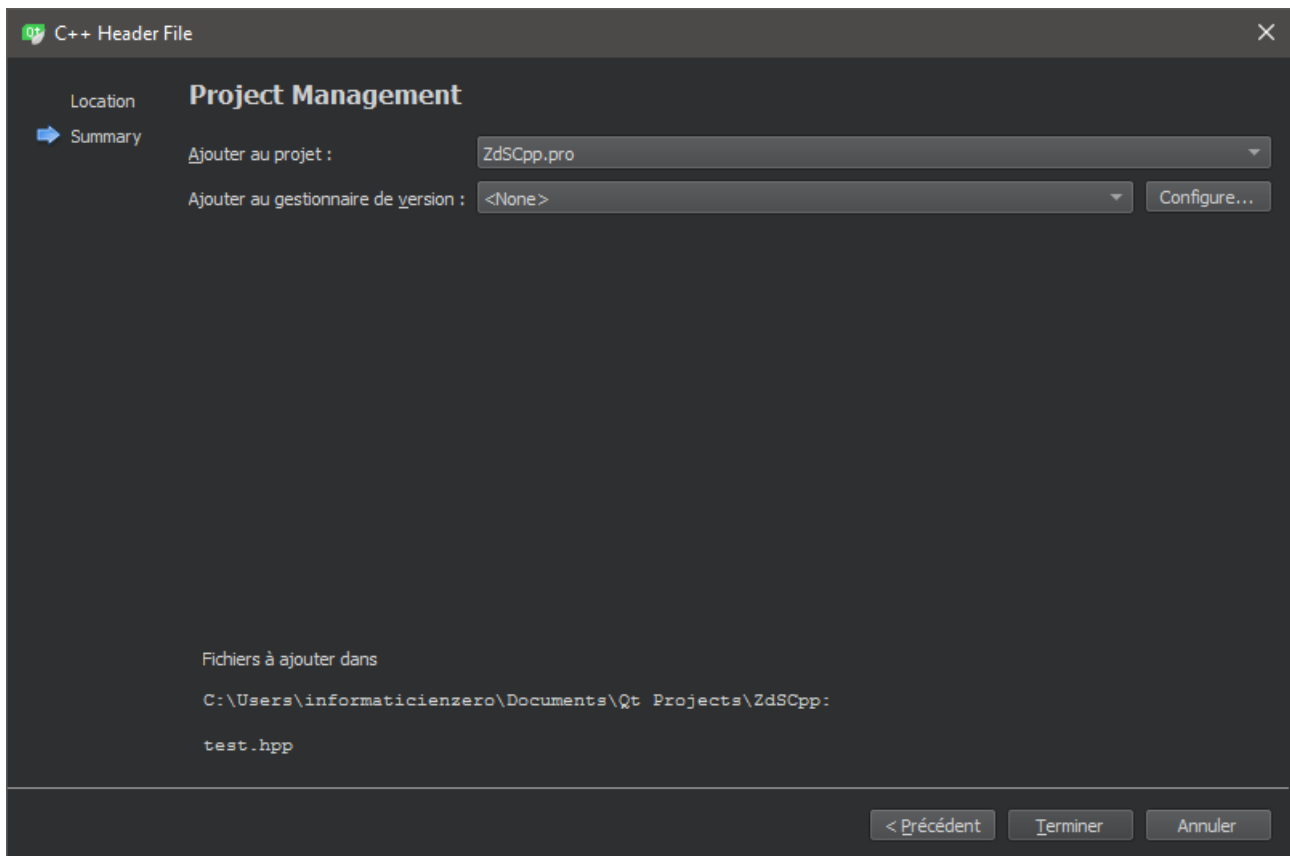


FIGURE III.11.6. – Normalement, vous n’avez rien besoin de toucher.

Tout est bon, il ne reste plus qu’à cliquer sur **Terminer** et nous en avons fini avec l’ajout de fichiers d’en-tête. On retombe sur la page principale, avec notre fichier bien présent en haut à gauche, signe qu’il a bien été ajouté au projet.

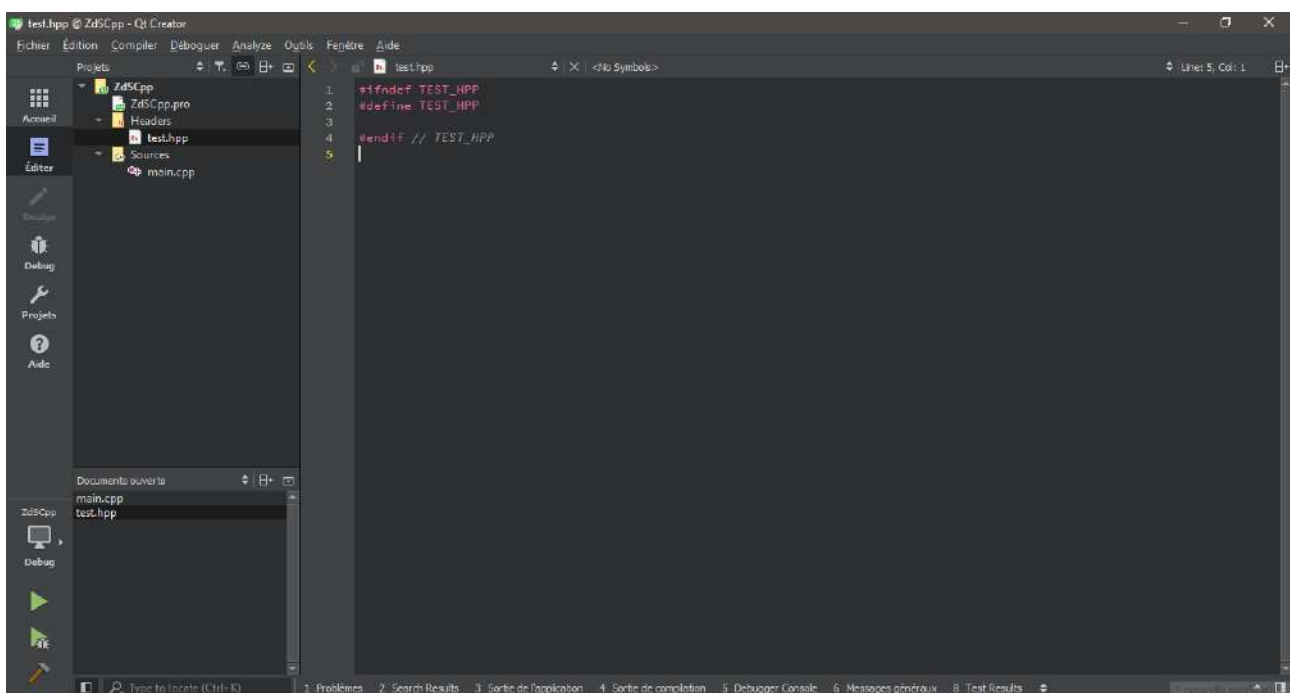


FIGURE III.11.7. – Voilà, tout est bon.

III.11.1.1.3. En ligne de commande

Rien de compliqué, il suffit de créer un simple fichier `.hpp` et c'est tout. 🍊

III.11.1.2. Le fichier source

Le fichier source **va contenir les implémentations de nos fonctions**. La création d'un fichier source est exactement identique à celle de la création d'un fichier d'en-tête, à ceci près que, cette fois, c'est un fichier source. 🍊

Un fichier source peut inclure **autant de fichiers d'en-tête que nécessaire**. Il y a bien sûr celui qui contient les prototypes et les types qu'il définit, mais il peut en inclure d'autres.

III.11.1.3. Créons un lien

Il reste une étape importante: lier les fichiers d'en-têtes et sources. Pour cela, il suffit d'inclure le fichier d'en-tête concerné dans le fichier source concerné. Cependant, à la différence des fichiers de la bibliothèque standard qui s'incluent avec des chevrons, on utilise les guillemets doubles `"`. On doit aussi écrire le nom complet, **extension comprise**. Celle-ci est `.hpp`.

Ainsi, si vous nommez votre fichier `test.hpp`, vous l'incluez dans le code en faisant comme suit.

```
1 #include "test.hpp"
```



Extension

Les fichiers d'en-tête peuvent aussi utiliser l'extension `.h`. C'est parfaitement valide. La différence n'est qu'une histoire de goût. Nous préférons `.hpp` à `.h` par analogie aux fichiers sources, qui se terminent par `.cpp` en C++ mais `.c` en C.

Voici un exemple tout bête mais fonctionnel de séparation en plusieurs fichiers. Pour le reproduire, créez les fichiers concernés dans votre projet et copiez-collez les codes ci-dessous.

III. On passe la deuxième!

test.hpp

```
1 // Parce que le compilateur a besoin de savoir ce qu'est
  std::string.
2 #include <string>
3
4 // On ne met ici que le prototype.
5 void afficher_message(std::string const & message);
```

test.cpp

```
1 // On a besoin de iostream pour utiliser std::cout,
  std::quoted et std::endl.
2 #include <iostream>
3 // On inclut le fichier d'en-tête correspondant.
4 #include "test.hpp"
5
6 // Par contre, comme 'test.hpp' inclut déjà '<string>', on n'a
  pas besoin de l'inclure de nouveau ici, dans le fichier
  source.
7
8 // On définit la fonction ici.
9 void afficher_message(std::string const & message)
10 {
11     std::cout << "J'ai reçu un message : " <<
        std::quoted(message) << std::endl;
12 }
```

main.cpp

```
1 // Pour appeler la fonction, il faut connaître son prototype,
  donc on doit inclure le fichier d'en-tête correspondant.
2 #include "test.hpp"
3
4 int main()
5 {
6     // Ici, on va utiliser la fonction.
7     afficher_message("Salut les lecteurs !");
8     return 0;
9 }
```

III. On passe la deuxième!

Pour compiler ce programme avec Visual Studio ou Qt Creator, rien de bien compliqué, on lance la compilation et c'est tout. Pour ceux qui sont en console, il faut **ajouter chaque fichier .cpp** à la liste des fichiers à compiler. Quant aux **.hpp**, **il ne faut pas les inclure**, nous verrons pourquoi plus tard.

L'exemple suivant se compile ainsi.

```
1 > g++ -std=c++17 test.cpp main.cpp -o programme.out
2 > clang++ -std=c++17 test.cpp main.cpp -o programme.out
```

III.11.2. La sécurité, c'est important

Avez-vous noté ces lignes nouvelles qui étaient automatiquement présentes lors de la création du fichier d'en-tête? Ce sont surtout celles de Qt Creator qui nous intéressent (celle de Visual Studio n'est pas du C++ standard). Je les réécris ci-dessous.

```
1 #ifndef TEST_HPP
2 #define TEST_HPP
3
4 #endif // TEST_HPP
```

C'est qu'en fait, lorsque le compilateur voit une ligne **#include**, il ne réfléchit pas et inclut le fichier en question. Sauf que, si, par erreur, ou même volontairement, on inclut plusieurs fois le même fichier, on va se retrouver avec **plusieurs fois les mêmes déclarations de fonctions, de types, etc.** Autant dire que le compilateur ne va que très moyennement apprécier.

Afin de s'en prémunir, il faut écrire ces trois lignes, qui sont des directives de préprocesseur et ce dans chaque fichier d'en-tête que vous créerez. Vous pouvez remplacer **TEST_HPP** par ce que vous voulez, à condition que **ce nom soit unique**. Le plus simple est de le remplacer par le nom du fichier en question: s'il s'appelle **autre.hpp**, alors écrivez **AUTRE_HPP**. Le code de l'en-tête ira entre la deuxième et la troisième ligne.

Ainsi, le fichier d'en-tête de l'exemple de la section précédente n'est pas correct. Il faut le corriger en utilisant ce que nous venons d'apprendre.

```
1 #ifndef TEST_HPP
2 #define TEST_HPP
3
4 #include <string>
5
6 void afficher_message(std::string const & message);
7
8 #endif
```

III.11.3. Découpons le TP!

Pour s'exercer, on va prendre le corrigé du TP précédent et on va l'organiser en plusieurs fichiers. Commençons par le commencement, séparons prototypes et implémentations. On aura alors l'esprit plus clair pour réorganiser tout ça une fois qu'on aura uniquement les prototypes sous les yeux.

☉ Séparation prototype / implémentation

On peut former des groupes de types et fonctions liées.

- Les fonctions utilitaires: pour l'instant, on n'a que `traitement_chaine`, mais elle pourrait être rejointe par d'autres lors de l'évolution du projet.
- Les types de base `Artiste`, `Album` et `Morceau` et les fonctions les manipulant.
- Les tests unitaires correspondants.
- Le type `Discographie` et ses fonctions de manipulation.
- Les types et fonctions concernant l'exécution des commandes.

On va donc créer cinq paires de fichiers pour réorganiser notre code en conséquence: `utils`, `donnees_disco`, `donnees_disco_tests`, `discographie`, `systeme_commandes`.

i

Solution

Comme d'habitude, il n'y a pas qu'une seule solution en programmation. L'organisation que je propose en est une, mais l'essentiel est juste d'avoir un code organisé de manière cohérente.

Cela donne donc les fichiers suivants.

☉ Utilitaire

☉ Données discographie

☉ Tests unitaires

☉ Discographie

☉ Commandes

Notre fichier `main.cpp` devient alors très simple.

III. On passe la deuxième!

```
1 #include "donnees_disco_tests.hpp"
2 #include "discographie.hpp"
3 #include "systeme_commandes.hpp"
4 #include "utils.hpp"
5
6 #include <stdexcept>
7
8 int main()
9 {
10     test_creation_morceau_entree_complete();
11     test_creation_morceau_entree_espaces_partout();
12     test_creation_morceau_entree_chanson_artiste();
13     test_creation_morceau_entree_chanson_uniquement();
14     test_creation_morceau_entree_vide();
15
16     Artiste const dave_brubeck { "Dave Brubeck " };
17     Artiste const secret_garden { "Secret Garden" };
18     Artiste const indochine { "Indochine" };
19
20     Album const time_out { "Time Out" };
21     Album const songs_from_a_secret_garden {
22         "Songs from a Secret Garden" };
23     Album const l_aventurier { "L'aventurier" };
24     Album const paradize { "Paradize" };
25
26     Morceau const take_five { "Take Five", dave_brubeck, time_out
27         };
28     Morceau const blue_rondo_a_la_turk { "Blue Rondo a la Turk",
29         dave_brubeck, time_out };
30     Morceau const nocturne { "Nocturne", secret_garden,
31         songs_from_a_secret_garden };
32     Morceau const aventurier { "L'aventurier", indochine,
33         l_aventurier };
34     Morceau const j_ai_demande_a_la_lune {
35         "J'ai demandé à la lune", indochine, paradize };
36
37     Discographie discographie { take_five, blue_rondo_a_la_turk,
38         nocturne, aventurier, j_ai_demande_a_la_lune };
39
40     bool continuer { true };
41     do
42     {
43         try
44         {
45             std::string entree { recuperer_commande() };
46             auto [commande, instructions] =
47                 analyser_commande(entree);
48             instructions = traitement_chaine(instructions);
```

III. On passe la deuxième!

```
41         continuer = executer_commande(discographie, commande,  
42                                     instructions);  
43     }  
44     catch (std::runtime_error const & exception)  
45     {  
46         std::cout << "Erreur : " << exception.what() <<  
47         std::endl;  
48     }  
49     } while (continuer);  
50     return 0;  
51 }
```


Et voilà! On a un code organisé en plusieurs fichiers, prêt à évoluer!

III.11.4. Les avantages du découpage en fichiers

Le découpage en fichiers permet de mieux organiser le code, certes. Mais cela ne s'arrête pas là; il y a d'autres avantages qu'on va voir maintenant.


III.11.4.1. Une structure de projet plus visible

Imaginez un projet encore plus conséquent que le T.P précédent. Tout est dans le même fichier. Complicé après d'y voir un ordre quelconque. Si, par contre, le projet est découpé intelligemment en plusieurs fichiers, vous pourrez, d'un coup d'œil, **comprendre la structure du projet** et mieux orienter vos recherches.

Le projet [Zeste de Savoir](#) , par exemple, est découpé en plein de fichiers, qui séparent les tests, l'affichage, la manipulation de la base de données, etc. Un développeur ayant des connaissances dans les technologies utilisées sait donc, d'un rapide regard, où il doit orienter ses recherches.

III.11.4.2. Une meilleure abstraction

L'avantage de séparer l'implémentation des prototypes et des définitions de type, notamment avec les fichiers d'en-tête, c'est que **l'utilisateur ne dépend plus de détails techniques** mais de fonctions générales. Si besoin est, on peut changer du tout au tout la façon d'écrire nos fonctions, pour les rendre plus sûres ou plus rapides et l'utilisateur n'y verra rien.

Vous, apprentis développeurs C++, dépendez des prototypes et abstractions de la bibliothèque standard. Par contre, en coulisse, l'implémentation n'est pas la même selon que vous utilisez Visual Studio, GCC ou Clang. Mais ça n'est pas un problème. **Nous dépendons d'abstractions, non de détails.** C'est un des [grands principes](#)  d'un programme bien conçu.

III. On passe la deuxième!

De plus, puisque les détails techniques sont cachés dans les fichiers sources, les fichiers d'en-tête sont bien plus lisibles et n'exposent que les fonctions et données publiques qui intéressent leur utilisateur.

III.11.4.3. Une meilleure modularité

Séparer en plusieurs fichiers, c'est aussi autoriser des portions de codes à être réutilisées dans d'autres endroits. Vous le savez très bien, puisque vous faites de nombreux `#include` d'en-têtes de la bibliothèque standard. Il serait plus difficile et bien plus lourd d'utiliser `std::string` ou `std::vector` s'il n'y avait pas les fichiers d'en-tête correspondants, n'est-ce pas?

Rien que dans le T.P précédent, le fait d'avoir séparé la fonction de traitement des chaînes de caractères, pour la mettre dans un fichier à part, nous autorise à la réutiliser ailleurs que ce qu'on avait originellement prévu.

III.11.5. Le cas des templates

Il me reste un cas particulier à aborder, celui des *templates*. Dans le chapitre qui leur est consacré, nous avons dit que le compilateur va instancier une fonction *template* chaque fois qu'elle est appelée avec un type réel différent. Mais pour l'instancier, le compilateur a besoin de son **implémentation complète**. Du coup, un code comme celui-ci ne peut pas fonctionner.

conteneur.hpp

```
1 #ifndef CONTENEUR_HPP
2 #define CONTENEUR_HPP
3
4 #include <iostream>
5
6 template <typename Collection>
7 void afficher(Collection const & iterable);
8
9 #endif
```

III. On passe la deuxième!

conteneur.cpp

```
1 #include "conteneur.hpp"
2
3 template <typename Collection>
4 void afficher(Collection const & iterable)
5 {
6     for (auto const & e : iterable)
7     {
8         std::cout << e << std::endl;
9     }
10 }
```

main.cpp

```
1 #include <iostream>
2 #include <vector>
3 #include "conteneur.hpp"
4
5 int main()
6 {
7     std::vector<int> const tableau { 1, 2, 3, 4 };
8     afficher(tableau);
9     return 0;
10 }
```

```
1 [Visual Studio]
2 Erreur LNK2019 symbole externe non résolu "void
  __cdecl afficher<class std::vector<int,class
  std::allocator<int> > >(class std::vector<int,class
  std::allocator<int> > const &)"
  (??$afficher@V?$vector@HV?$allocator@H@std@@@std@@@YAXABV?$vector@HV?$all
  référencé dans la fonction _main
3
4 -----
5
6 [GCC]
7 /usr/bin/ld: /tmp/cc7jhqie.o: in function `main':
8 main.cpp:(.text+0x6e): undefined reference to `void
  afficher<std::vector<int, std::allocator<int> >
  >(std::vector<int, std::allocator<int> > const&)'
```

III. On passe la deuxième!

```
9 collect2: error: ld a retourné le statut de sortie 1
10
11 -----
12
13 [Clang]
14 /usr/bin/ld: /tmp/main-27208d.o: in function `main':
15 main.cpp:(.text+0x80): undefined reference to `void
    afficher<std::vector<int, std::allocator<int> >
    >(std::vector<int, std::allocator<int> > const&)'
16 clang-7: error: linker command failed with exit code 1 (use -v to
    see invocation)
```

Le compilateur va en effet se plaindre (avec une erreur cryptique, soit dit en passant), car il essaye d'instancier le *template* `afficher` mais ne trouve nulle part son implémentation. Afin de compiler, il faut que **toutes nos fonctions *templates* soient déclarées et implémentées directement dans les fichiers d'en-tête**. C'est contraire à ce que nous venons d'apprendre dans ce chapitre, mais c'est pourtant ce qu'il faut faire.

conteneur.hpp

```
1 #ifndef CONTENEUR_HPP
2 #define CONTENEUR_HPP
3
4 #include <iostream>
5
6 template <typename Collection>
7 void afficher(Collection const & iterable)
8 {
9     for (auto const & e : iterable)
10     {
11         std::cout << e << std::endl;
12     }
13 }
14
15 #endif
```

```
1 1
2 2
3 3
4 4
```



Lecteur attentif

Il n'y a vraiment pas d'autres solutions? Parce que tu as dis toi-même que c'était bien que les détails techniques restent cachés dans les fichiers sources.

Eh bien figurez-vous que si! En fait, il y a une astuce toute simple, qui consiste à déclarer les fonctions dans le fichier d'en-tête, mais à **les implémenter dans un fichier séparé** (souvent avec l'extension `.hpp`), fichier qui sera **includé à la fin du fichier d'en-tête**. Ainsi, on cache les détails techniques dans un autre fichier tout en laissant la compilation se faire.

conteneur_impl.hpp

```
1 // Implémentation des fonctions templates.
2
3 template <typename Collection>
4 void afficher(Collection const & iterable)
5 {
6     for (auto const & e : iterable)
7     {
8         std::cout << e << std::endl;
9     }
10 }
```

conteneur.hpp

```
1 #ifndef CONTENEUR_HPP
2 #define CONTENEUR_HPP
3
4 #include <iostream>
5
6 template <typename Collection>
7 void afficher(Collection const & iterable);
8
9 // A la fin, on rajoute le fichier d'implémentation.
10 #include "conteneur_impl.hpp"
11
12 #endif
```

III. On passe la deuxième!

main.cpp

```
1 #include <iostream>
2 #include <vector>
3
4 // Le fichier d'implémentation est lui aussi inclut, du coup.
5 #include "conteneur.hpp"
6
7 int main()
8 {
9     std::vector<int> const tableau { 1, 2, 3, 4 };
10    afficher(tableau);
11    return 0;
12 }
```

```
1 1
2 2
3 3
4 4
```

III.11.5.1. En résumé

- Découper en fichiers permet de séparer les définitions des implémentations.
- Les fichiers d'en-tête contiennent les définitions et portent, dans ce cours, l'extension `.hpp`.
- Les fichiers source contiennent les implémentations et les détails techniques. Ils terminent par l'extension `.cpp`.
- Lors de la création de fichiers d'en-tête, il faut bien penser à les sécuriser avec les directives de préprocesseur abordées.
- Découper son projet en fichiers permet une meilleure modularité, une meilleure réutilisation, une structure de projet plus visible.
- Dans le cas des *templates*, il faut que l'implémentation soit écrite directement dans le fichier d'en-tête, ou bien dans un fichier séparé qui lui-même sera inclus dans le fichier d'en-tête concerné.

Vous avez maintenant un certain niveau et êtes capables de faire beaucoup de choses. Bien entendu, votre apprentissage ne s'arrête pas là et je vous invite à continuer ce cours à nos côtés. Mais prenez quelques instants pour mesurer vos nouvelles compétences et apprécier les efforts que vous avez fournis. 🍊

Contenu masqué

Contenu masqué n°60 : Séparation prototype / implémentation

```
1 #include <algorithm>
2 #include <cassert>
3 #include <cctype>
4 #include <fstream>
5 #include <iostream>
6 #include <sstream>
7 #include <string>
8 #include <tuple>
9 #include <vector>
10
11 std::string traitement_chaine(std::string const & chaine);
12
13 struct Artiste
14 {
15     std::string nom;
16 };
17
18 std::ostream & operator<<(std::ostream & sortie, Artiste const &
19     artiste);
20
21 struct Album
22 {
23     std::string nom;
24 };
25
26 std::ostream & operator<<(std::ostream & sortie, Album const &
27     album);
28
29 struct Morceau
30 {
31     std::string nom;
32     Artiste compositeur;
33     Album album;
34 };
35
36 std::ostream & operator<<(std::ostream & sortie, Morceau const &
37     morceau);
38 std::istream & operator>>(std::istream & entree, Morceau &
39     morceau);
40
41 void test_creation_morceau_entree_complete();
42 void test_creation_morceau_entree_espaces_partout();
```

III. On passe la deuxième!

```
39 void test_creation_morceau_entree_chanson_artiste();
40 void test_creation_morceau_entree_chanson_uniquement();
41 void test_creation_morceau_entree_vide();
42
43 using Discographie = std::vector<Morceau>;
44
45 void tri_morceau(Discographie& discographie);
46 void tri_album(Discographie& discographie);
47 void tri_artiste(Discographie& discographie);
48
49 void affichage_morceau(Discographie const & discographie);
50 void affichage_album(Discographie const & discographie);
51 void affichage_artiste(Discographie const & discographie);
52
53 enum class Affichage { Artiste, Album, Morceau };
54
55 void affichage(Discographie& discographie, Affichage
    type_affichage);
56
57 void enregistrement(Discographie const & discographie, std::string
    const & nom_fichier);
58 void chargement(Discographie & discographie, std::string const &
    nom_fichier);
59
60 std::string recuperer_commande();
61
62 enum class Commande { Afficher, Ajouter, Enregistrer, Charger,
    Quitter };
63
64 std::tuple<Commande, std::string> analyser_commande(std::string
    const & commande_texte);
65 bool executer_commande(Discographie & discographie, Commande
    commande, std::string const & instructions);
66
67 int main()
68 {
69     test_creation_morceau_entree_complete();
70     test_creation_morceau_entree_espaces_partout();
71     test_creation_morceau_entree_chanson_artiste();
72     test_creation_morceau_entree_chanson_uniquement();
73     test_creation_morceau_entree_vide();
74
75     Artiste const dave_brubeck { "Dave Brubeck " };
76     Artiste const secret_garden { "Secret Garden" };
77     Artiste const indochine { "Indochine" };
78
79     Album const time_out { "Time Out" };
80     Album const songs_from_a_secret_garden {
        "Songs from a Secret Garden" };
81     Album const l_aventurier { "L'aventurier" };
```

III. On passe la deuxième!

```
82 Album const paradise { "Paradize" };
83
84 Morceau const take_five { "Take Five", dave_brubeck, time_out
    };
85 Morceau const blue_rondo_a_la_turk { "Blue Rondo a la Turk",
    dave_brubeck, time_out };
86 Morceau const nocturne { "Nocturne", secret_garden,
    songs_from_a_secret_garden };
87 Morceau const aventurier { "L'aventurier", indochine,
    l_aventurier };
88 Morceau const j_ai_demande_a_la_lune {
    "J'ai demandé à la lune", indochine, paradise };
89
90 Discographie discographie { take_five, blue_rondo_a_la_turk,
    nocturne, aventurier, j_ai_demande_a_la_lune };
91
92 bool continuer { true };
93 do
94 {
95     try
96     {
97         std::string entree { recuperer_commande() };
98         auto[commande, instructions] =
99             analyser_commande(entree);
100        instructions = traitement_chaine(instructions);
101        continuer = executer_commande(discographie, commande,
102            instructions);
103    }
104    catch (std::runtime_error const & exception)
105    {
106        std::cout << "Erreur : " << exception.what() <<
107            std::endl;
108    }
109 } while (continuer);
110
111 return 0;
112 }
113
114 std::string traitement_chaine(std::string const & chaine)
115 {
116     std::string copie { chaine };
117
118     auto premier_non_espace { std::find_if_not(std::begin(copie),
119         std::end(copie), isspace) };
120     copie.erase(std::begin(copie), premier_non_espace);
121
122     std::reverse(std::begin(copie), std::end(copie));
123     premier_non_espace = std::find_if_not(std::begin(copie),
124         std::end(copie), isspace);
```


III. On passe la deuxième!

```
121     copie.erase(std::begin(copie), premier_non_espace);
122     std::reverse(std::begin(copie), std::end(copie));
123
124     return copie;
125 }
126
127 std::ostream & operator<<(std::ostream & sortie, Artiste const &
    artiste)
128 {
129     sortie << artiste.nom;
130     return sortie;
131 }
132
133 std::ostream & operator<<(std::ostream & sortie, Album const &
    album)
134 {
135     sortie << album.nom;
136     return sortie;
137 }
138
139 std::ostream & operator<<(std::ostream & sortie, Morceau const &
    morceau)
140 {
141     sortie << morceau.nom << " | " << morceau.album << " | " <<
        morceau.compositeur;
142     return sortie;
143 }
144
145 std::istream & operator>>(std::istream & entree, Morceau & morceau)
146 {
147     std::string mot {};
148     std::ostringstream flux {};
149
150     while (entree >> mot && mot != "|")
151     {
152         flux << mot << " ";
153     }
154
155     std::string nom_morceau { flux.str() };
156     if (std::empty(nom_morceau))
157     {
158         nom_morceau = "Morceau inconnu";
159     }
160     morceau.nom = traitement_chaine(nom_morceau);
161     flux.str(std::string {});
162
163     while (entree >> mot && mot != "|")
164     {
165         flux << mot << " ";
166     }
```

III. On passe la deuxième!

```
167
168     std::string nom_album { flux.str() };
169     if (std::empty(nom_album))
170     {
171         nom_album = "Album inconnu";
172     }
173     morceau.album.nom = traitement_chaine(nom_album);
174     flux.str(std::string {});
175
176     while (entree >> mot)
177     {
178         flux << mot << " ";
179     }
180
181     std::string nom_artiste { flux.str() };
182     if (std::empty(nom_artiste))
183     {
184         nom_artiste = "Artiste inconnu";
185     }
186     morceau.compositeur.nom = traitement_chaine(nom_artiste);
187     flux.str(std::string {});
188
189     return entree;
190 }
191
192 void test_creation_morceau_entree_complete()
193 {
194     std::istringstream entree { "Frica | Frica | Carla's Dreams" };
195     Morceau morceau {};
196
197     entree >> morceau;
198
199     assert(morceau.nom == "Frica" &&
200            u8"Le nom du morceau doit être Frica.");
201     assert(morceau.album.nom == "Frica" &&
202            u8"Le nom de l'album doit être Frica.");
203     assert(morceau.compositeur.nom == "Carla's Dreams" &&
204            u8"Le nom du compositeur doit être Carla's Dreams.");
205 }
206
207 void test_creation_morceau_entree_espaces_partout()
208 {
209     std::istringstream entree {
210         "Levels | Levels | Avicii" };
211     Morceau morceau {};
```

III. On passe la deuxième!

```
212     assert(morceau.album.nom == "Levels" &&  
213            u8"Le nom de l'album doit être Levels.");  
213     assert(morceau.compositeur.nom == "Avicii" &&  
214            u8"Le nom du compositeur doit être Avicii.");  
214 }  
215  
216 void test_creation_morceau_entree_chanson_artiste()  
217 {  
218     std::istringstream entree {  
219         "Subeme la radio | | Enrique Iglesias" };  
219     Morceau morceau {};  
220  
221     entree >> morceau;  
222  
223     assert(morceau.nom == "Subeme la radio" &&  
224            u8"Le nom du morceau doit être Subeme la radio.");  
224     assert(morceau.album.nom == "Album inconnu" &&  
225            u8"Le nom de l'album doit être Album inconnu.");  
225     assert(morceau.compositeur.nom == "Enrique Iglesias" &&  
226            u8"Le nom du compositeur doit être Enrique Iglesias.");  
226 }  
227  
228 void test_creation_morceau_entree_chanson_uniquement()  
229 {  
230     std::istringstream entree { "Duel of the fates | |" };  
231     Morceau morceau {};  
232  
233     entree >> morceau;  
234  
235     assert(morceau.nom == "Duel of the fates" &&  
236            u8"Le nom du morceau doit être Duel of the Fates.");  
236     assert(morceau.album.nom == "Album inconnu" &&  
237            u8"Le nom de l'album doit être Album inconnu.");  
237     assert(morceau.compositeur.nom == "Artiste inconnu" &&  
238            u8"Le nom du compositeur doit être Artiste inconnu.");  
238 }  
239  
240 void test_creation_morceau_entree_vide()  
241 {  
242     std::istringstream entree { "| |" };  
243     Morceau morceau {};  
244  
245     entree >> morceau;  
246  
247     assert(morceau.nom == "Morceau inconnu" &&  
248            u8"Le nom du morceau doit être Morceau inconnu.");  
248     assert(morceau.album.nom == "Album inconnu" &&  
249            u8"Le nom de l'album doit être Album inconnu.");  
249     assert(morceau.compositeur.nom == "Artiste inconnu" &&  
250            u8"Le nom du compositeur doit être Artiste inconnu.");
```

III. On passe la deuxième!

```
250 }
251
252 void tri_morceau(Discographie& discographie)
253 {
254     std::sort(std::begin(discographie), std::end(discographie),
255               [](Morceau const & lhs, Morceau const & rhs)
256               {
257                 return lhs.nom < rhs.nom;
258             });
259 }
260
261 void tri_album(Discographie& discographie)
262 {
263     std::sort(std::begin(discographie), std::end(discographie),
264               [](Morceau const & lhs, Morceau const & rhs)
265               {
266                 if (lhs.album.nom < rhs.album.nom)
267                     return true;
268
269                 return lhs.album.nom == rhs.album.nom && lhs.nom <
270                     rhs.nom;
271             });
272 }
273
274 void tri_artiste(Discographie& discographie)
275 {
276     std::sort(std::begin(discographie), std::end(discographie),
277               [](Morceau const & lhs, Morceau const & rhs)
278               {
279                 if (lhs.compositeur.nom < rhs.compositeur.nom)
280                     return true;
281
282                 if (lhs.compositeur.nom == rhs.compositeur.nom)
283                 {
284                     if (lhs.album.nom < rhs.album.nom)
285                         return true;
286
287                     return lhs.album.nom == rhs.album.nom &&
288                         lhs.nom < rhs.nom;
289                 }
290
291                 return false;
292             });
293 }
294
295 // Affichage de la discographie par morceau, dans l'ordre
296 // alphabétique.
297 void affichage_morceau(Discographie const & discographie)
298 {
299     for (Morceau const & morceau : discographie)
```

III. On passe la deuxième!

```
294     {
295         std::cout << "--> " << morceau << std::endl;
296     }
297 }
298
299 // Affichage de la discographie par album, dans l'ordre
    alphabétique.
300 void affichage_album(Discographie const & discographie)
301 {
302     Album album_precedent{};
303     for (Morceau const & morceau : discographie)
304     {
305         if (morceau.album.nom != album_precedent.nom)
306         {
307             std::cout << "--> " << morceau.album <<
                " | " << morceau.compositeur <<
                std::endl;
308         }
309         std::cout << "\t/--> " << morceau.nom << std::endl;
310
311         album_precedent = morceau.album;
312     }
313 }
314
315 // Affichage de la discographie par artiste, dans l'ordre
    alphabétique.
316 void affichage_artiste(Discographie const & discographie)
317 {
318     Artiste artiste_precedent{};
319     Album album_precedent{};
320
321     for (Morceau const & morceau : discographie)
322     {
323         if (morceau.compositeur.nom !=
            artiste_precedent.nom)
324         {
325             std::cout << "--> " << morceau.compositeur
                << std::endl;
326         }
327
328         if (morceau.album.nom != album_precedent.nom)
329         {
330             std::cout << "\t/--> " << morceau.album <<
                std::endl;
331         }
332
333         std::cout << "\t\t/--> " << morceau.nom <<
            std::endl;
334
335         artiste_precedent = morceau.compositeur;
```

III. On passe la deuxième!

```
336         album_precedent = morceau.album;
337     }
338 }
339
340 void affichage(Discographie& discographie, Affichage
    type_affichage)
341 {
342     if (type_affichage == Affichage::Album)
343     {
344         tri_album(discographie);
345         affichage_album(discographie);
346     }
347     else if (type_affichage == Affichage::Artiste)
348     {
349         tri_artiste(discographie);
350         affichage_artiste(discographie);
351     }
352     else if (type_affichage == Affichage::Morceau)
353     {
354         tri_morceau(discographie);
355         affichage_morceau(discographie);
356     }
357     else
358     {
359         // Par exemple si on met à jour l'énumération mais
            qu'on oublie d'ajouter la fonction
            correspondante.
360         throw
            std::runtime_error("Commande d'affichage inconnue.");
361     }
362 }
363
364 void enregistrement(Discographie const & discographie, std::string
    const & nom_fichier)
365 {
366     std::ofstream fichier { nom_fichier };
367     if (!fichier)
368     {
369         throw
            std::runtime_error("Impossible d'ouvrir le fichier en écriture.");
370     }
371
372     for (Morceau const & morceau : discographie)
373     {
374         fichier << morceau << std::endl;
375     }
376 }
377
378 void chargement(Discographie & discographie, std::string const &
    nom_fichier)
```

III. On passe la deuxième!

```
379 {
380     std::ifstream fichier { nom_fichier };
381     if (!fichier)
382     {
383         throw
384             std::runtime_error("Impossible d'ouvrir le fichier en lecture.");
385     }
386     std::string ligne {};
387     while (std::getline(fichier, ligne))
388     {
389         Morceau morceau {};
390
391         std::istringstream flux { ligne };
392         flux >> morceau;
393
394         discographie.push_back(morceau);
395     }
396 }
397
398 std::string recuperer_commande()
399 {
400     std::cout << "> ";
401
402     std::string commande {};
403     std::getline(std::cin, commande);
404     return commande;
405 }
406
407 std::tuple<Commande, std::string> analyser_commande(std::string
408     const & commande_texte)
409 {
410     std::istringstream flux { commande_texte };
411
412     std::string premier_mot {};
413     flux >> premier_mot;
414     premier_mot = traitement_chaine(premier_mot);
415
416     std::string instructions {};
417     std::getline(flux, instructions);
418     instructions = traitement_chaine(instructions);
419
420     if (premier_mot == "afficher")
421     {
422         return { Commande::Afficher, instructions };
423     }
424     else if (premier_mot == "ajouter")
425     {
426         return { Commande::Ajouter, instructions };
427     }
```

III. On passe la deuxième!

```
427     else if (premier_mot == "enregistrer")
428     {
429         return{ Commande::Enregistrer, instructions };
430     }
431     else if (premier_mot == "charger")
432     {
433         return { Commande::Charger, instructions };
434     }
435     else if (premier_mot == "quitter")
436     {
437         return { Commande::Quitter, std::string {} };
438     }
439     else
440     {
441         throw std::runtime_error("Commande invalide.");
442     }
443 }
444
445 bool executer_commande(Discographie & discographie, Commande
commande, std::string const & instructions)
446 {
447     if (commande == Commande::Afficher)
448     {
449         if (instructions == "artistes")
450         {
451             affichage(discographie, Affichage::Artiste);
452         }
453         else if (instructions == "albums")
454         {
455             affichage(discographie, Affichage::Album);
456         }
457         else if (instructions == "morceaux")
458         {
459             affichage(discographie, Affichage::Morceau);
460         }
461         else
462         {
463             throw
std::runtime_error("Commande d'affichage inconnue.");
464         }
465     }
466     else if (commande == Commande::Ajouter)
467     {
468         std::istringstream flux { instructions };
469         Morceau morceau {};
470         flux >> morceau;
471         discographie.push_back(morceau);
472     }
473     else if (commande == Commande::Charger)
474     {
```


III. On passe la deuxième!

```
475     chargement(discographie, instructions);
476 }
477 else if (commande == Commande::Enregistrer)
478 {
479     enregistrement(discographie, instructions);
480 }
481 else if (commande == Commande::Quitter)
482 {
483     return false;
484 }
485
486 return true;
487 }
```

[Retourner au texte.](#)

Contenu masqué n°61 : Utilitaire

/utils.hpp

```
1 #ifndef UTILS_HPP_INCLUDED
2 #define UTILS_HPP_INCLUDED
3
4 #include <string>
5
6 std::string traitement_chaine(std::string const & chaine);
7
8 #endif
```

III. On passe la deuxième!

/utils.cpp

```
1 #include "utils.hpp"
2
3 #include <algorithm>
4
5 std::string traitement_chaine(std::string const & chaine)
6 {
7     std::string copie { chaine };
8
9     auto premier_non_espace {
10         std::find_if_not(std::begin(copie), std::end(copie),
11             isspace) };
12     copie.erase(std::begin(copie), premier_non_espace);
13
14     std::reverse(std::begin(copie), std::end(copie));
15     premier_non_espace = std::find_if_not(std::begin(copie),
16         std::end(copie), isspace);
17     copie.erase(std::begin(copie), premier_non_espace);
18     std::reverse(std::begin(copie), std::end(copie));
19
20     return copie;
21 }
```

[Retourner au texte.](#)

Contenu masqué n°62 :
Données discographie

III. On passe la deuxième!

/donnees_disco.cpp

```
1  #ifndef DONNEES_DISCO_HPP_INCLUDED
2  #define DONNEES_DISCO_HPP_INCLUDED
3
4  #include <string>
5  #include <iostream>
6  #include <vector>
7
8  struct Artiste
9  {
10     std::string nom;
11 };
12
13 std::ostream & operator<<(std::ostream & sortie, Artiste const
    & artiste);
14
15 struct Album
16 {
17     std::string nom;
18 };
19
20 std::ostream & operator<<(std::ostream & sortie, Album const &
    album);
21
22 struct Morceau
23 {
24     std::string nom;
25     Artiste compositeur;
26     Album album;
27 };
28
29 std::ostream & operator<<(std::ostream & sortie, Morceau const
    & morceau);
30 std::istream & operator>>(std::istream & entree, Morceau &
    morceau);
31
32 using Discographie = std::vector<Morceau>;
33
34 #endif
```

III. On passe la deuxième!

/donnees_disco.cpp

```
1 #include "donnees_disco.hpp"
2
3 #include "utils.hpp"
4 #include <sstream>
5
6 std::ostream & operator<<(std::ostream & sortie, Artiste const
    & artiste)
7 {
8     sortie << artiste.nom;
9     return sortie;
10 }
11
12 std::ostream & operator<<(std::ostream & sortie, Album const &
    album)
13 {
14     sortie << album.nom;
15     return sortie;
16 }
17
18 std::ostream & operator<<(std::ostream & sortie, Morceau const
    & morceau)
19 {
20     sortie << morceau.nom << " | " << morceau.album << " | " <<
        morceau.compositeur;
21     return sortie;
22 }
23
24 std::istream & operator>>(std::istream & entree, Morceau &
    morceau)
25 {
26     std::string mot {};
27     std::ostringstream flux {};
28
29     while (entree >> mot && mot != "|")
30     {
31         flux << mot << " ";
32     }
33
34     std::string nom_morceau { flux.str() };
35     if (std::empty(nom_morceau))
36     {
37         nom_morceau = "Morceau inconnu";
38     }
39     morceau.nom = traitement_chaine(nom_morceau);
40     flux.str(std::string {});
41
42     while (entree >> mot && mot != "|")
43     {
44         flux << mot << " ";
45     }
46
47     std::string nom_album { flux.str() };
48     morceau.album = traitement_chaine(nom_album);
49     flux.str(std::string {});
50
51     while (entree >> mot && mot != "|")
52     {
53         flux << mot << " ";
54     }
55     morceau.compositeur = traitement_chaine(flux.str());
56     flux.str(std::string {});
57
58     return entree;
59 }
```

[Retourner au texte.](#)

Contenu masqué n°63 : Tests unitaires

/donnees_disco_tests.hpp

```
1  #ifndef DONNEES_DISCO_TESTS_HPP_INCLUDED
2  #define DONNEES_DISCO_TESTS_HPP_INCLUDED
3
4  void test_creation_morceau_entree_complete();
5  void test_creation_morceau_entree_espaces_partout();
6  void test_creation_morceau_entree_chanson_artiste();
7  void test_creation_morceau_entree_chanson_uniquement();
8  void test_creation_morceau_entree_vide();
9
10 #endif
```

III. On passe la deuxième!

/donnees_disco_tests.cpp

```
1 #include "donnees_disco_tests.hpp"
2
3 #include <sstream>
4 #include <cassert>
5 #include "donnees_disco.hpp"
6
7 void test_creation_morceau_entree_complete()
8 {
9     std::istringstream entree {
10         "Frica | Frlica | Carla's Dreams" };
11     Morceau morceau {};
12
13     entree >> morceau;
14
15     assert(morceau.nom == "Frica" &&
16         u8"Le nom du morceau doit être Frlica.");
17     assert(morceau.album.nom == "Frica" &&
18         u8"Le nom de l'album doit être Frlica.");
19     assert(morceau.compositeur.nom == "Carla's Dreams" &&
20         u8"Le nom du compositeur doit être Carla's Dreams.");
21 }
22
23 void test_creation_morceau_entree_espaces_partout()
24 {
25     std::istringstream entree {
26         "Levels | Levels | Avicii" };
27     Morceau morceau {};
28
29     entree >> morceau;
30
31     assert(morceau.nom == "Levels" &&
32         u8"Le nom du morceau doit être Levels.");
33     assert(morceau.album.nom == "Levels" &&
34         u8"Le nom de l'album doit être Levels.");
35     assert(morceau.compositeur.nom == "Avicii" &&
36         u8"Le nom du compositeur doit être Avicii.");
37 }
38
39 void test_creation_morceau_entree_chanson_artiste()
40 {
41     std::istringstream entree {
42         "Subeme la radio | | Enrique Iglesias" };
43     Morceau morceau {};
44
45     entree >> morceau;
46
47     assert(morceau.nom == "Subeme la radio" &&
48         u8"Le nom du morceau doit être Subeme la radio.");
49     assert(morceau.album.nom == "Album inconnu" &&
50         u8"Le nom de l'album doit être Album inconnu.");
51     assert(morceau.compositeur.nom == "Enrique Iglesias" &&
52         u8"Le nom du compositeur doit être Enrique Iglesias.");
53 }
```

Contenu masqué n°64 : Discographie

/discographie.hpp

```
1  #ifndef DISCOGRAPHIE_HPP_INCLUDED
2  #define DISCOGRAPHIE_HPP_INCLUDED
3
4  #include "donnees_disco.hpp"
5  #include <string>
6
7  void tri_morceau(Discographie& discographie);
8  void tri_album(Discographie& discographie);
9  void tri_artiste(Discographie& discographie);
10
11 void affichage_morceau(Discographie const & discographie);
12 void affichage_album(Discographie const & discographie);
13 void affichage_artiste(Discographie const & discographie);
14
15 enum class Affichage { Artiste, Album, Morceau };
16
17 void affichage(Discographie& discographie, Affichage
    type_affichage);
18
19 void enregistrement(Discographie const & discographie,
    std::string const & nom_fichier);
20 void chargement(Discographie & discographie, std::string const
    & nom_fichier);
21
22 #endif
```

III. On passe la deuxième!

/discographie.cpp

```
1  #include "discographie.hpp"
2
3  #include <algorithm>
4  #include <fstream>
5  #include <sstream>
6  #include <stdexcept>
7
8  void tri_morceau(Discographie& discographie)
9  {
10     std::sort(std::begin(discographie),
11               std::end(discographie), [](Morceau const & lhs,
12                                           Morceau const & rhs)
13     {
14         return lhs.nom < rhs.nom;
15     });
16 }
17
18 void tri_album(Discographie& discographie)
19 {
20     std::sort(std::begin(discographie),
21               std::end(discographie), [](Morceau const & lhs,
22                                           Morceau const & rhs)
23     {
24         if (lhs.album.nom < rhs.album.nom)
25             return true;
26
27         return lhs.album.nom == rhs.album.nom &&
28                lhs.nom < rhs.nom;
29     });
30 }
31
32 void tri_artiste(Discographie& discographie)
33 {
34     std::sort(std::begin(discographie),
35               std::end(discographie), [](Morceau const & lhs,
36                                           Morceau const & rhs)
37     {
38         if (lhs.compositeur.nom < rhs.compositeur.nom)
39             return true;
40
41         if (lhs.compositeur.nom ==
42             rhs.compositeur.nom)
43         {
44             if (lhs.album.nom < rhs.album.nom)
45                 return true;
46
47             return lhs.album.nom == rhs.album.nom
48                    && lhs.nom < rhs.nom;
49         }
50
51         return false;
52     });
53 }
```


Contenu masqué n°65 : Commandes

/systeme_commandes.hpp

```
1  #ifndef SYSTEME_COMMANDES_HPP_INCLUDED
2  #define SYSTEME_COMMANDES_HPP_INCLUDED
3
4  #include "donnees_disco.hpp"
5
6  #include <string>
7  #include <tuple>
8
9  std::string recuperer_commande();
10
11  enum class Commande { Afficher, Ajouter, Enregistrer, Charger,
12      Quitter };
13
14  std::tuple<Commande, std::string>
15      analyser_commande(std::string const & commande_texte);
16  bool executer_commande(Discographie & discographie, Commande
17      commande, std::string const & instructions);
18
19 #endif
```

III. On passe la deuxième!

/systeme_commandes.cpp

```
1  #include "discographie.hpp"
2  #include "systeme_commandes.hpp"
3  #include "utils.hpp"
4
5  #include <sstream>
6  #include <stdexcept>
7  #include <string>
8
9  std::string recuperer_commande()
10 {
11     std::cout << "> ";
12
13     std::string commande {};
14     std::getline(std::cin, commande);
15     return commande;
16 }
17
18 std::tuple<Commande, std::string>
19 analyser_commande(std::string const & commande_texte)
20 {
21     std::istringstream flux { commande_texte };
22
23     std::string premier_mot {};
24     flux >> premier_mot;
25     premier_mot = traitement_chaine(premier_mot);
26
27     std::string instructions {};
28     std::getline(flux, instructions);
29     instructions = traitement_chaine(instructions);
30
31     if (premier_mot == "afficher")
32     {
33         return { Commande::Afficher, instructions };
34     }
35     else if (premier_mot == "ajouter")
36     {
37         return { Commande::Ajouter, instructions };
38     }
39     else if (premier_mot == "enregistrer")
40     {
41         return { Commande::Enregistrer, instructions };
42     }
43     else if (premier_mot == "charger")
44     {
45         return { Commande::Charger, instructions };
46     }
47     else if (premier_mot == "quitter")
48     {
49         return { Commande::Quitter, std::string {} };
50     }
51     else
```

III. On passe la deuxième!

[Retourner au texte.](#)

Quatrième partie

Interlude - Être un développeur

IV. Interlude - Être un développeur

L'art de développer ne se limite pas à la maîtrise du langage utilisé. Il y a, en effet, bien d'autres aspects dans le développement que la pure programmation et c'est ce que nous allons découvrir dans cette partie. Les outils que vous y découvrirez vous donneront une grande autonomie pour mener à bien des projets seuls. Vous serez alors à même de créer des programmes mine de rien assez complexes. 🍊

IV.1. Avant-propos

Nous avons découvert de très nombreuses notions depuis le début de ce tutoriel, et avec elles nous avons touché du doigt certains aspects importants du développement informatique. Il est temps de revenir sur ce que nous avons fait et de l'analyser avec un peu plus de recul. Si nous insistons autant sur ces aspects, c'est parce qu'être un bon développeur nécessite plus que d'écrire du C++.

IV.1.1. Ce que nous avons appris

Jusqu'à présent, nous avons abordé plusieurs aspects de la programmation en C++, que ce soit les notions de base comme les conditions, les boucles, ou encore des notions plus avancées comme l'utilisation d'outils fournis par la bibliothèque standard, ainsi que la création de fonctions et de structures de données personnalisées. Vous avez ainsi déjà acquis un arsenal de programmation assez complet pour faire des programmes relativement complexes. Le T.P de la partie précédente est là pour le prouver.

Cependant, nous avons vu dans ces deux parties bien plus que de simples notions de C++. En effet, nous avons touché du doigt **certains aspects essentiels du développement**. Plutôt que de simplement apprendre à programmer en C++, nous avons découvert une certaine manière de réfléchir qui traduit les qualités d'un bon développeur: le désir d'avoir toujours un code de qualité, à travers **un travail de conception poussé** et **des bonnes pratiques**. Citons, parmi celles-ci, les **tests unitaires** pour vérifier que notre code fait ce qu'on lui demande, les **contrats** pour définir clairement ce que les différents intervenants attendent comme entrées et sorties et les **exceptions** pour remonter des problèmes à l'utilisateur.

Ce sont ces qualités qui font qu'un développeur sera non seulement capable d'utiliser son langage, mais surtout **de mener ses projets à bien**. Pour être un bon marin, il ne suffit pas de savoir utiliser la voile et le gouvernail, encore faut-il savoir naviguer, s'orienter dans la mer. De même, il ne suffit pas pour un développeur de maîtriser ses outils pour mener sa barque à bon port.

Je pense que vous avez commencé à vous rendre compte de ces aspects-là, notamment grâce à la partie II qui insiste fortement sur l'amélioration de la qualité du code. Et c'est l'une des choses les plus importantes à retenir des deux premières parties. Dans cet interlude, nous allons justement découvrir de nouvelles notions qui s'inscrivent dans cette optique, à savoir faire de vous **de bons développeurs** et pas simplement de bons programmeurs.

IV.1.2. Différents paradigmes

Nous avons souvent dit que tout problème de développement a de nombreuses solutions, l'essentiel étant que les choix que l'on fait soient guidés par la volonté de produire un code propre et robuste à travers une bonne conception.

Eh bien cette diversité fait que l'on classe les manières de programmer en différentes catégories, appelées paradigmes. Le C++ étant un langage multi-paradigmes permettant de programmer de manière très libre, il nous laisse choisir parmi plusieurs de ces manières de programmer. Nous en avons déjà croisé quelques-unes.

IV.1.2.1. Le paradigme impératif

Le [paradigme impératif](#) est le principal paradigme que l'on a utilisé. C'est aussi celui de beaucoup d'autres langages de programmation, dont notamment les premiers inventés. Dans cette approche, le programme est une suite d'instructions qui s'exécutent linéairement et le font avancer. Ce programme inclut les notions de conditions, boucles, fonctions, etc. Une grande partie des notions que l'on a abordées s'inscrit, au moins partiellement, dans ce paradigme.

Voici un exemple d'un code écrit typiquement avec le paradigme impératif, qui met chaque élément d'un tableau au carré.

```
1 #include <iostream>
2 #include <vector>
3
4 int main()
5 {
6     std::vector<int> nombres { -8, 7, 48, 366, 1, 4, 3 };
7     for (int i = 0; i < nombres.size(); ++i)
8     {
9         nombres[i] *= nombres[i];
10    }
11
12    for (auto nombre : nombres)
13    {
14        std::cout << nombre << std::endl;
15    }
16
17    return 0;
18 }
```

```
1 64
2 49
3 2304
4 133956
```

5	1
6	16
7	9

IV.1.2.2. Le paradigme fonctionnel

Le [paradigme fonctionnel](#) est aussi en partie couvert par le langage C++. C'est un paradigme dans lequel, comme son nom l'indique, **les fonctions jouent un rôle prépondérant**. D'ailleurs, dans les langages dit «fonctionnellement purs», la modification de variable peut ne pas exister et tout est réalisé avec des calculs sur résultats de fonctions! Nous avons principalement vu ce paradigme à travers les lambdas, qui sont un concept venant tout droit des langages fonctionnels. De manière plus générale, le fait de pouvoir manipuler les fonctions comme des variables vient du paradigme fonctionnel.

Le code suivant est une réécriture de l'exemple précédent, dans un style plus fonctionnel.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <vector>
4
5 int main()
6 {
7     std::vector<int> nombres { -8, 7, 48, 366, 1, 4, 3 };
8     std::for_each(std::begin(nombres), std::end(nombres), [](int &
9         nombre) -> void
10     {
11         nombre *= nombre;
12     });
13     for (auto nombre : nombres)
14     {
15         std::cout << nombre << std::endl;
16     }
17
18     return 0;
19 }
```

1	64
2	49
3	2304
4	133956
5	1
6	16
7	9

IV.1.2.3. Le paradigme générique

Le [paradigme générique](#) [↗](#) consiste principalement en l'idée que parfois, on se fiche du type réel des données que l'on manipule. Ce qui compte, c'est la manière dont elles peuvent être utilisées. Nous l'avons principalement appréhendé sous deux formes.

- Les algorithmes standards qui peuvent s'appliquer à tout un tas de collections différentes.
- Les fonctions *templates* qui permettent d'écrire du code fonctionnant avec plusieurs types, du moment qu'on peut les manipuler de cette manière.


En reprenant les exemples précédents, utilisons le paradigme générique. Voici un résultat qu'on peut obtenir.

```
1 #include <algorithm>
2 #include <iostream>
3 #include <list>
4 #include <vector>
5
6 template <typename Collection>
7 void au_carre(Collection & collection)
8 {
9     std::for_each(std::begin(collection), std::end(collection),
10         [](auto & nombre) -> void
11     {
12         nombre *= nombre;
13     });
14
15 template <typename Collection>
16 void affichage(Collection const & collection)
17 {
18     for (auto element : collection)
19     {
20         std::cout << element << std::endl;
21     }
22 }
23
24 int main()
25 {
26     std::vector<int> nombres { -8, 7, 48, 366, 1, 4, 3 };
27     au_carre(nombres);
28     affichage(nombres);
29
30     std::cout << std::endl;
31
32     std::list<int> autres { 10, 30, 60 };
33     au_carre(autres);
34     affichage(autres);
35 }
```

```
36     return 0;  
37 }
```

```
1  64  
2  49  
3  2304  
4  133956  
5  1  
6  16  
7  9  
8  
9  100  
10 900  
11 3600
```

IV.1.2.4. La programmation par contrat

Bien que ce ne soit pas aussi poussé que dans d'autres langages comme Eiffel ou le D, en C++, il est possible de faire de la [programmation par contrat](#) , comme nous l'avons vu dans le chapitre sur la gestion des erreurs. Comme nous avons pu le constater, les principes de la programmation par contrat se combinent très bien avec les autres paradigmes, donc n'hésitez pas à en user et abuser. 🍊



C++20 et PpC

La prochaine version de C++ (C++20) devrait inclure de nouveaux outils pour faire de la programmation par contrat en C++. Cela permettra de l'utiliser de manière encore plus poussée.

Maintenant que l'on a plus de recul, on voit que le C++, contrairement à d'autres langages plus restrictifs, met à notre disposition tout un panel d'outils très variés pour écrire du code de qualité. Nous pouvons prendre un peu de chaque paradigme et se servir de ses points forts. Dans la suite du cours, nous introduirons un nouveau paradigme, lui aussi très important en C++ et qu'on appelle la **programmation orientée objet**.



Remarque au passage

Ce n'est pas une critique envers d'autres langages que de dire qu'ils sont moins permissifs. C'est parfois un avantage d'utiliser un langage qui fait des choix forts. Simplement, l'un des attraits du C++ est cette permissivité.

Mais c'est aussi l'un de ses plus grands dangers. Comme on peut faire beaucoup de choses, on peut très facilement faire n'importe quoi! C'est pourquoi il faut être particulièrement



concentré sur le fait de produire de code propre et robuste. D'où le fait que nous insistions régulièrement dessus.

IV.1.3. Brancher le cerveau avant tout

Nous l'avons compris, le C++ nous offre de nombreux outils, ainsi que la liberté d'utiliser et mélanger plusieurs paradigmes. Mais tout cela n'est qu'outils et instruments. Il faut les assembler intelligemment et harmonieusement pour produire de bons programmes. Pour ça, il faut, avant de se ruer sur le clavier et de martyriser énergiquement ses pauvres touches, **réfléchir et concevoir**.

On prend habituellement un stylo et un papier², puis **on se pose des questions** sur le programme sur lequel on travaille. On réfléchit aux fonctionnalités du programme, ce qu'il rendra comme service, ce qu'il ne fera pas, ce qu'il est capable de gérer, etc. On se demande quels sont les cas d'erreurs, les entrées invalides, ce qui peut potentiellement poser problème, etc.



Les contrats

Nous avons déjà commencé à faire ça, notamment avec les contrats et la réflexion sur nos entrées et sorties.

On découpe aussi de grosses fonctionnalités **en plus petites** afin qu'elles soient plus facilement abordables. Un logiciel de traitement d'images peut ainsi être découpé en sous-tâches comme l'ouverture d'un fichier PNG, la modification d'un unique pixel, la sauvegarde du PNG modifié, etc.

C'est là que le fait de noter ses idées se révèle utile. On peut comparer plusieurs solutions à un même problème, on peut visualiser plus facilement à quoi ressemblera le programme final, on peut écrire du pseudo-code pour s'aider à mieux comprendre un algorithme complexe, etc. Et puis, personne n'ayant une mémoire infailible, noter ses idées aide à s'en rappeler. 🍊

Cependant, concevoir, anticiper, tout cela ne signifie pas que le programme est figé et qu'il n'évoluera pas d'un pouce. Au contraire, réfléchir avant de coder permet de mieux prévoir des aspects qui pourront être amenés à changer. Cela rend le programme **plus évolutif**.



Planification ou souplesse?

C'est un exercice délicat que de trouver l'équilibre entre le néant et trop de spécifications, entre une trop grande rigidité et une absence de réflexion et de conception. Ne vous inquiétez pas. Comme beaucoup de choses, tout vient en pratiquant. L'étude de cas vous donnera un exemple concret.

2. Enfin, un document numérique ça marche aussi. Question de goût.

IV.1.4. Savoir se débrouiller

Être un bon développeur, c'est être **autonome** et **se débrouiller**. Cela ne veut pas dire que les développeurs ne se parlent jamais entre eux, vivant reclus les uns des autres. Au contraire, l'existence même de ce tutoriel et de la plateforme Zeste de Savoir sont des preuves que l'échange est important.

En fait, ce qu'il faut comprendre, c'est qu'il **ne faut pas s'attendre à ce que tout arrive cuit dans le bec**. Un bon développeur ne compte pas sur les autres pour faire son travail. Il n'attend pas de réponse toute faite. Quand il est bloqué, il réfléchit d'abord et demande ensuite.

Si vous êtes vraiment bloqués sur un problème, bien entendu que vous pouvez demander un coup de pouce, notamment sur Zeste de Savoir. Mais quand cela arrivera, montrez ce que vous avez tenté, les erreurs qui vous bloquent, si vous avez déjà fait des recherches dessus, etc. Et vous verrez, vous aurez de l'aide et des réponses. 🍌

i

À venir

Dans cette partie, nous verrons notamment une technique appelée le **débugage**, qui permet de voir pas-à-pas l'exécution du code pour trouver là où une erreur se cache. Avec cette compétence, vous pourrez déjà bien avancer seuls dans la recherche de la solution.

En conclusion, laissez-moi vous lister quelques qualités, en plus de l'autonomie, qui feront de vous de meilleurs développeurs.

- **Patience**. Il en faut pour apprendre, pour comprendre, pour faire et corriger des erreurs, pour chercher des solutions, pour ne pas s'énervier quand ça ne marche pas.
- **Persévérance** et **motivation**. Développer, c'est parfois rencontrer des problèmes qui nous dépassent, qui semblent trop complexes à première vue, ou bien des notions trop abstraites. Ces qualités aideront à ne pas abandonner à la première difficulté.
- **Humilité**. Un bon développeur accepte de ne pas avoir réponse à tout et donc de demander de l'aide quand il n'y arrive vraiment pas. Il accepte les conseils et les corrections qu'on lui donne. Il accepte de ne pas se reposer sur ses acquis, se remet régulièrement à jour et en question. Cela lui évite les réponses méprisantes, arrogantes et hautaines qu'on voit parfois sur certains forums.

IV.1.4.1. En résumé

- Nous avons découvert plein de choses sur le langage C++ pendant ce cours, mais on a aussi mis en avant des aspects plus généraux du développement, comme l'écriture de tests unitaires, de contrats ou la réflexion à mener sur les entrées / sorties.
- Le C++ est un langage très riche qui nous permet de programmer de manières très différentes.
- Cette liberté est une force mais aussi un danger, c'est pourquoi une bonne conception et du code de qualité sont primordiaux.

IV. Interlude - Être un développeur

- Être un bon développeur demande aussi plusieurs qualités, dont la persévérance, l'autonomie et la patience.
- La suite de cette partie nous aidera à tendre vers cet idéal en nous apprenant de nouvelles notions pour mieux développer.

IV.2. Mais où est la doc ?

Cette chère documentation est une des plus vieilles amies du développeur. Parfois très complète, parfois vraiment courte, parfois pas présente du tout (pas de bol), elle est là pour **le guider, répondre à ses questions** et lui apprendre comment utiliser tel ou tel type, ce qu'attend telle ou telle fonction, etc. Heureusement, dans notre cas, il existe une [excellente documentation](#) pour C++, en anglais certes (sachez que la [version française](#) est une traduction automatique), mais simple à comprendre. Et elle n'aura bientôt plus de secret pour vous. 🍊

Nous allons apprendre à **lire la documentation C++** ainsi qu'à **écrire celle de nos programmes**.

IV.2.1. Lire une page de doc

IV.2.1.1. À l'arrivée

Vous êtes donc sur la documentation et tant de liens vous laissent perplexes. Comment on s'y retrouve là-dedans ? Par où chercher l'information ? Le site étant classé en plusieurs parties, tout dépend de ce que vous cherchez. Certaines vont plus nous intéresser, dans le cadre de ce cours, que d'autres.

- **Headers / Fichiers d'en-tête standard**. Cette page recense tous les fichiers d'en-tête de la bibliothèque standard. Nous en reconnaissons plusieurs, comme `<iostream>`, `<string>` ou `<vector>`. Ils sont classés par catégorie, en fonction de s'ils concernent la manipulation de chaînes de caractères, les conteneurs, etc.
- **String library / Chaînes de caractères**. Ici, nous allons retrouver tout ce qui concerne le célèbre `std::string`.
- **Containers library / Conteneurs**. Ici, vous avez un lien direct vers tous les conteneurs qui existent. Vous reconnaîtrez notamment `vector`, `array` et `list`.
- **Algorithms library / Algorithmes**. Vous en connaissez certains, mais il en reste d'autres, tous listés ici.

Je vous cite ici les autres catégories, pour information et pour satisfaire votre curiosité débordante.



- **Utilities library**. Correspond aux utilitaires pour gérer les types, la mémoire, les erreurs, les dates et temps, etc.
- **Iterators library / Itérateurs**. Vous connaissez déjà les itérateurs. Ici, on retrouve la documentation de tout ceux qu'on connaît, plus bien d'autres encore.
- **Numerics library / Numérique**. C'est tout ce qui touche à la manipulation des nombres (complexes, aléatoires, etc).

IV. Interlude - Être un développeur

- ***Input/output library* / Entrées/sorties.** On a déjà un peu manipulé, mais il reste encore beaucoup de choses.
- ***Localizations library* / Internationalisation.** Nous n'en parlerons pas, mais il y a ici des éléments permettant d'adapter son programme à d'autres langues et cultures.
- ***Filesystem library* / Système de fichiers.** Pour permettre la manipulation de dossiers et de fichiers de manière standard.
- D'autres encore, d'un niveau trop avancé pour être expliqués maintenant.

Au fur et à mesure que vous progresserez en C++ et gagnerez en autonomie, vous pourrez vous débrouiller seuls dans la documentation, car celle-ci est organisée de manière logique.

IV.2.1.2. `vector` – Retour sur le plus célèbre des conteneurs

IV.3. Présentation

Commençons en examinant la [page](#) dédiée à `vector`. Il y a une très grande quantité d'informations, mais toutes ne nous sont pas utiles. Commençons par lire à quoi ressemble un `vector`.

```
1 template<
2     class T,
3     class Allocator = std::allocator<T>
4 > class vector;
```

Vous reconnaissez la syntaxe des templates. Seulement, vous aviez l'habitude jusque là de déclarer des objets de type tableau de la manière suivante `std::vector<int> tableau`, alors qu'ici vous voyez qu'il y a deux types génériques attendus. C'est parce que **le deuxième est renseigné avec une valeur par défaut**.

- `T` est le type des éléments.
- `Allocator` est utilisé pour l'acquisition de la mémoire pour stocker les éléments.

On ne modifie le deuxième qu'en cas de besoins très particuliers, notamment quand on veut manipuler la mémoire d'une manière précise, donc pour nous, ça ne présente aucun intérêt. On laisse donc la valeur par défaut et on retrouve ainsi la syntaxe habituelle.

IV.3.0.0.1. Les fonctions membres

Sautons plusieurs détails et passons directement à la section *Member functions* / **Fonctions membres**. Ici sont listées toutes les fonctions que l'on peut appliquer à un `vector` avec la syntaxe `mon_vecteur.fonction()`. Certaines vous sont familières, comme `push_back` ou `clear`. Si vous cliquez sur l'une d'elles, vous aurez tous les détails la concernant. Examinons ces deux exemples.

Chaque page de fonction présente d'abord tous les prototypes de la fonction. Comme vous le savez déjà, avec les surcharges, il peut exister plusieurs fonctions avec le même identificateur mais des paramètres différents. C'est le cas ici pour [push_back](#). Si une fonction est apparue en même temps qu'une nouvelle norme de C++, ou si elle a été supprimée, cela est précisé à la fin de la ligne.

```
1 void push_back( const T& value ); (1)
2 void push_back( T&& value );      (2)           (since C++11)
```


Dans le cas de la fonction `push_back`, nous avons deux surcharges, dont la deuxième n'est disponible qu'à partir de C++11.



Petite explication

La syntaxe du deuxième prototype doit vous surprendre. En effet, nous ne l'avons pas encore abordée. Nous le ferons plus loin dans le cours, mais sachez que, entre autres, la deuxième version est appelée **quand vous lui donnez directement un littéral** (`push_back(5);`), alors que si vous passez une variable, ça sera le premier prototype (`int const a { 5 }; push_back(a);`). Pour faire simple, cela permet d'optimiser le code lorsqu'il est appelé avec une valeur temporaire.

IV.3.0.0.2. Les fonctions non-membres

Sur la page de `std::vector`, on trouve d'autres fonctions, dites **non-membres**. Celles-ci utilisent la syntaxe `fonction(mon_vecteur)`, en opposition à ce que nous avons vu au sous-titre précédent mais en relation avec des fonctions que vous connaissez, comme `std::size` ou `std::begin`. C'est ici que vous trouverez la plupart des opérateurs surchargés, notamment.

On trouve, par exemple, `std::swap` [↗](#), qui permet d'échanger le contenu de deux vecteurs, ou bien des opérateurs de `comparaison` [↗](#).

IV.3.0.0.3. Les paramètres

Revenons sur la page `push_back`. La sous-section **Parameters** y décrit les différents paramètres qu'une fonction attend. Rien de bien surprenant pour vous dans le cas de `push_back`, il s'agit simplement de la valeur à insérer. Ensuite, la sous-section **Return value** décrit le résultat, qui, dans notre cas, n'existe pas puisque la fonction renvoie un `void`.

IV.3.0.0.4. Les exceptions

Intéressons-nous maintenant à la section **Exceptions**. Comme vous le savez, C++ permet d'utiliser les exceptions et certaines fonctions de la bibliothèque standard le font. Cette section décrit le comportement attendu au cas où une fonction viendrait à lancer une exception. En effet, toutes ne réagissent pas de la même manière.

- Certaines ont une **garantie forte** (*strong exception guarantee*). Elles garantissent que, si une exception est levée, alors rien n'est modifié et tout reste comme avant le lancement de l'exception. C'est le cas de `push_back`: si une exception est levée, alors le tableau reste inchangé.
- D'autres ont une **garantie basique** (*basic exception guarantee*). Dans ce cas, la seule chose dont on est sûr, c'est qu'il n'y a pas de fuite de mémoire. Par contre, l'objet peut avoir été modifié.
- D'autres encore ont une **garantie pas d'exception** (*no-throw guarantee*). Aucune exception ne sera levée. Ces fonctions possèdent le mot-clef `noexcept` dans leur prototype. C'est le cas de la fonction `empty` [↗](#) par exemple.

IV. Interlude - Être un développeur

- Enfin, il en existe qui n'ont **aucune garantie** (*No exception safety*). Tout peut se passer potentiellement. Ce genre de fonction est celle qui offre le moins de garantie et de protection.

Dans le cas où la fonction est `noexcept`, la section **Exceptions** peut être absente de la page de documentation. Cela dépend des cas.



Attention avec `noexcept`

Si une fonction spécifiée `noexcept` lève quand même une exception, la fonction `std::terminate` est appelée et celle-ci provoque **un arrêt brutal du programme**.

IV.3.0.0.5. Exemple(s)

Enfin, la section *Example* / **Exemple** donne un cas d'utilisation concret de la fonction exposée. On peut même modifier et exécuter le code en cliquant sur `Run this code`.

IV.3.0.1. Les algorithmes

Continuons notre apprentissage en nous rendant sur [la page des algorithmes](#) `↗`. Nous en connaissons déjà quelques-uns, ce qui va nous faciliter la compréhension de la documentation. Notez que les algorithmes sont eux-mêmes classés en sous-catégories, comme «*Non-modifying sequence operations* / Non modifiants», «*Modifying sequence operations* / Modifiants», «*Sorting operations* / Tri», etc. Au début de chaque sous-catégorie se trouve le fichier d'en-tête à inclure pour utiliser les algorithmes sous-cités.

Prenons l'exemple de `std::sort`, que vous connaissez déjà. Si vous le cherchez dans la page, vous le trouverez dans la catégorie «*Sorting operations*», puisqu'il s'agit d'un algorithme de tri. On note en haut que le fichier d'en-tête requis est `<algorithm>`. Maintenant que nous avons ces informations, nous pouvons nous rendre sur [sa page](#) `↗`.

Une fois là-bas, on remarque qu'il existe quatre signatures différentes, chacune étant ordonnée avec un numéro. La première est celle que nous connaissons le mieux, puisque nous l'avons déjà vue. On remarque que le prototype change en fonction de si l'on compile en C++20 ou antérieur. La différence réside dans le mot-clef `constexpr`, que je détaillerai plus tard.

```
1 template <class RandomIt>
2 void sort(RandomIt first, RandomIt last);
```

Le troisième lui ressemble, à ceci près qu'on a une possibilité de personnaliser l'algorithme de comparaison. Là encore, la seule différence entre pré-C++20 et post-C++20, c'est `constexpr`.

```
1 template <class RandomIt, class Compare>  
2 void sort(RandomIt first, RandomIt last, Compare comp);
```

Les deux autres, à savoir le (2) et le (4), ont été introduits avec C++17 et permettent de traiter plusieurs instructions en parallèle plutôt que les unes à la suite des autres. C'est encore compliqué à notre niveau, donc je ne m'embêterai pas à les expliquer. De toute façon, à la fin de ce chapitre, vous aurez l'autonomie et les ressources nécessaires pour aller lire et pratiquer par vous-mêmes. 🍊

La section *Parameters* nous donne des indications supplémentaires sur les arguments attendus par `std::sort`. On y trouve notamment la forme que doit prendre la fonction transmise en paramètre de `comp`. Dans ce cas-ci, vous le saviez déjà, mais c'est dans cette section que vous trouverez la même info pour d'autres algorithmes.

Enfin, avant d'avoir un exemple en code, on nous informe que la fonction ne renvoie rien et qu'elle peut éventuellement lever une exception de type `std::bad_alloc`, nous indiquant par-là de quoi nous protéger.

IV.3.0.2. Les chaînes de caractère au grand complet

Voici la [page de documentation de `std::string`](#) . Comme déjà dit lors de l'introduction de `using`, le type `std::string` est un alias d'un type `basic_string`, instancié pour les `char`. Ne vous étonnez donc pas en voyant la documentation.

IV.3.0.2.1. Exemple de fonction membre

Vous pouvez voir, dans la section *Member functions*, tout un tas de fonctions que nous ne connaissons pas encore. Certaines sont familières, comme `push_back` ou `find`. D'autres nouvelles, comme `insert` , que nous allons étudier.

En regardant les différents prototypes proposés, on distingue pas moins de 11 signatures différentes. On peut insérer plusieurs fois un même caractère (1), on peut insérer une chaîne entière (2)(4), des morceaux de chaînes grâce aux itérateurs (8) ou à des index (5), etc. Chacune est détaillée plus bas, avec explication de chaque paramètre attendu.

Toutes ces fonctions peuvent lever des exceptions. La documentation cite `std::out_of_range` si on transmet un index plus grand que la taille de la chaîne (logique) et `std::length_error` si on tente de créer une trop grande chaîne. Dans tous les cas, on a une garantie forte.

IV.3.0.2.2. Exemple de fonction non-membre

Comme pour `std::vector`, on a ici des opérateurs de , mais d'autres encore. Ainsi, quand nous concaténons deux chaînes en faisant `"A"s + "B"s`, nous faisons appel à `operator+` .

IV.3.0.2.3. Exemple d'autres fonctions associées

Tout ce qui permet de convertir une chaîne de caractères en entiers ou réels (et vice-versa) est listé sur cette page. Ces fonctions ne font pas partie du type `std::string`, mais sont tellement liées qu'elles sont réunies au sein du même fichier d'en-tête. On retrouve ainsi [stoi](#) [↗](#), [stod](#) [↗](#) et [to_string](#) [↗](#).

IV.3.1. Exercices

Pour que vous gagniez en autonomie et que vous soyez capables de vous débrouiller seuls dans la documentation, il faut que vous vous entraîniez. Le but de cette section d'exercices est donc que vous appreniez à chercher une information dans la documentation. C'est important, car vous n'aurez pas toujours un tutoriel pour vous expliquer quoi et où chercher.

IV.3.1.1. Remplacer une chaîne de caractère par une autre

J'aimerais pouvoir remplacer un morceau d'une chaîne de caractères par un autre. Disons que je veux remplacer une portion commençant à un index précis et d'une longueur précises. Le code suivant devra être complété pour pouvoir s'exécuter.

```
1 #include <cassert>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::string phrase { "J'aimerais annuler ce texte." };
8     std::string const mot { "remplacer" };
9
10    // Le code à écrire.
11
12    assert(phrase == "J'aimerais remplacer ce texte." &&
13           "Le texte n'a pas été modifié correctement.");
14    return 0;
15 }
```

👁 Correction

IV.3.1.2. Norme d'un vecteur

Pour ceux qui ont quelques connaissances en vecteurs, vous savez comment en calculer la norme. Pour les autres, sachez que la norme d'un vecteur est «sa longueur» que l'on calcule simplement

IV. Interlude - Être un développeur

à l'aide de la formule suivante. Il me semble qu'il y a une fonction pour ça en C++, mais laquelle?

Norme d'un vecteur

Si \vec{v} est un vecteur de coordonnées $(x; y)$, alors sa norme $\|\vec{v}\|$ vaut $\sqrt{x^2 + y^2}$.

```
1 #include <cassert>
2
3 int main()
4 {
5     int const x { 5 };
6     int const y { 12 };
7     int norme { 0 };
8
9     // Le code pour résoudre l'exercice.
10
11     assert(norme == 13 &&
12            "La norme d'un vecteur v(5;12) vaut 13.");
13     return 0;
14 }
```

👁 Indice

👁 Correction

IV.3.1.3. Nombres complexes

Ah, les nombres complexes! Saviez-vous que C++ permettait d'en utiliser? Eh oui, je peux calculer la racine carrée de -1 si ça me chante. En fait non, c'est plutôt vous qui allez le faire. Que diriez-vous de faire ce calcul puis d'afficher la partie réelle puis la partie imaginaire du résultat?

👁 Indice

👁 Correction

IV.3.1.4. Transformations

Vous connaissez déjà l'algorithme `std::for_each` [↗](#), qui applique une opération sur chaque élément d'un conteneur. Mais imaginez que nous ne voulons pas modifier le tableau lui-même mais une copie. Est-ce possible? À vous de trouver l'algorithme adapté. 🍏

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string const phrase {
7         "Voici une phrase longue que je veux en majuscules." };
8     std::string sortie { "" };
9
10    // L'algorithme à utiliser.
11
12    std::cout << sortie << std::endl;
13    return 0;
14 }
```

👁 Correction

IV.3.2. Documenter son code avec Doxygen

Vous commencez maintenant à vous débrouiller avec la documentation officielle de C++. C'est une très bonne chose, car savoir lire et comprendre une documentation est un gros atout en informatique, vu que la très grande majorité des langages et des programmes viennent avec la leur. Mais les vôtres alors? C'est un peu vide n'est-ce-pas? La suite logique serait donc d'apprendre à écrire la documentation de notre code.

IV.3.2.1. Installation des outils

Cela va se faire en utilisant un outil appelé [Doxygen](#) [↗](#). Pour l'installer sur Windows, rendez-vous sur la page des [téléchargements](#) [↗](#) et prenez la dernière version disponible. Pour les utilisateurs de GNU/Linux, regarder dans vos dépôts. Le paquet devrait s'appeler `doxygen`, accompagné ou non de `doxygen-doxywizard`³. Une fois ceci fait, nous allons utiliser le **Doxygen Wizard**, l'interface graphique pour lancer la génération de la documentation.

3. Référez-vous à la documentation de votre distribution favorite pour savoir exactement quoi installer.

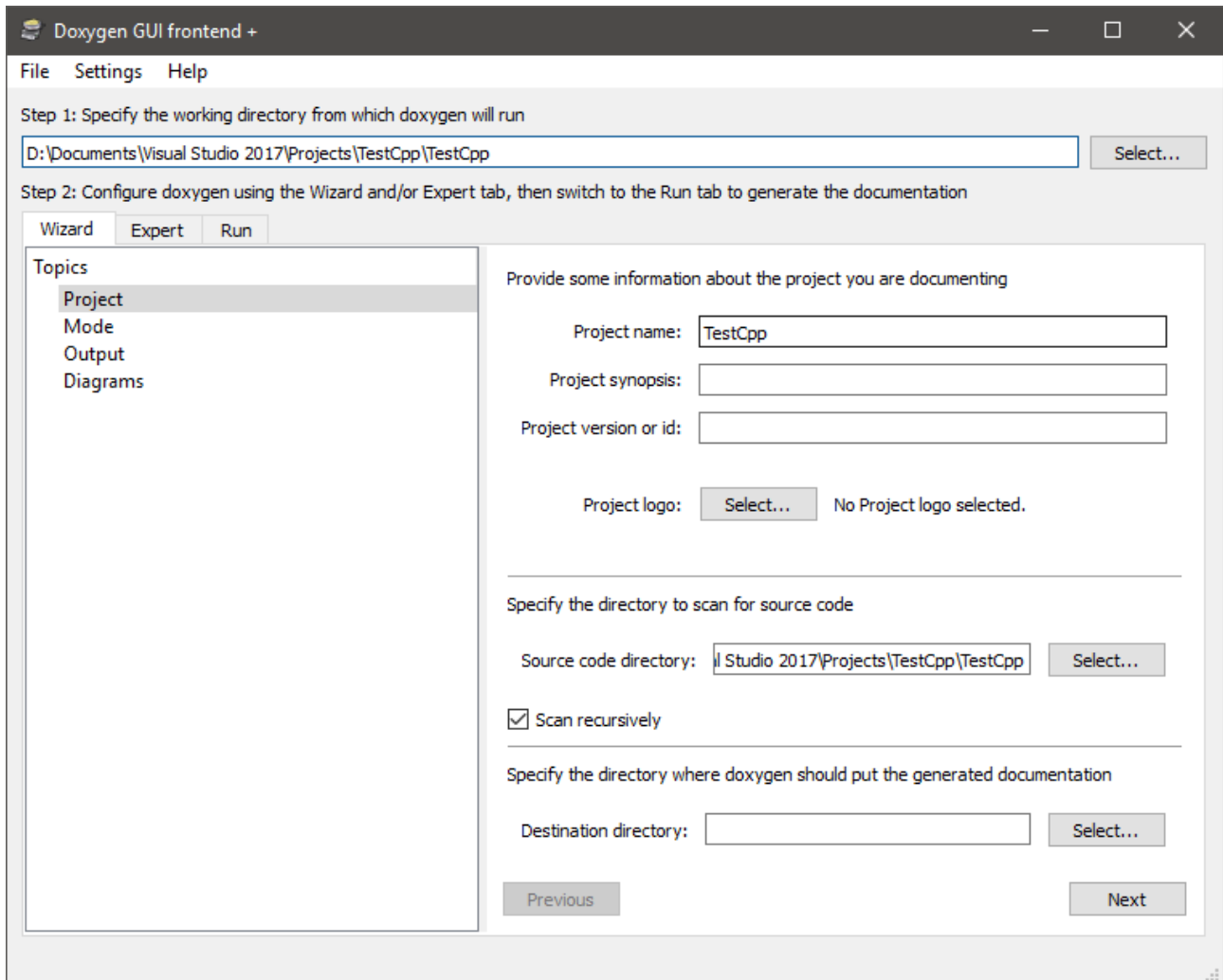


FIGURE IV.3.1. – Voici le « magicien Doxygen », si l'on traduit son nom en français.

La première étape est de **définir le chemin de sortie**, là où la documentation sera écrite. Personnellement, je la mets dans le même dossier que le code, mais vous êtes libres de faire comme vous le souhaitez.

Ensuite, on s'assure qu'on est bien sur l'onglet « *Wizard* » où l'on va pouvoir configurer plusieurs options. Les trois premiers champs sont facultatifs, mais permettent de définir le nom du projet, son but et son numéro de version, dans le cas où il y aura plusieurs versions du programme et donc de la documentation disponibles. Vous pouvez vous amuser à mettre un logo aussi.

Il faut ensuite préciser **où se situe le code source exactement**. Dans mon cas, il est dans le même dossier qu'indiqué plus haut, mais dans d'autres, il peut être rangé dans un sous-dossier. Mettez le chemin du dossier qui contient votre code source.

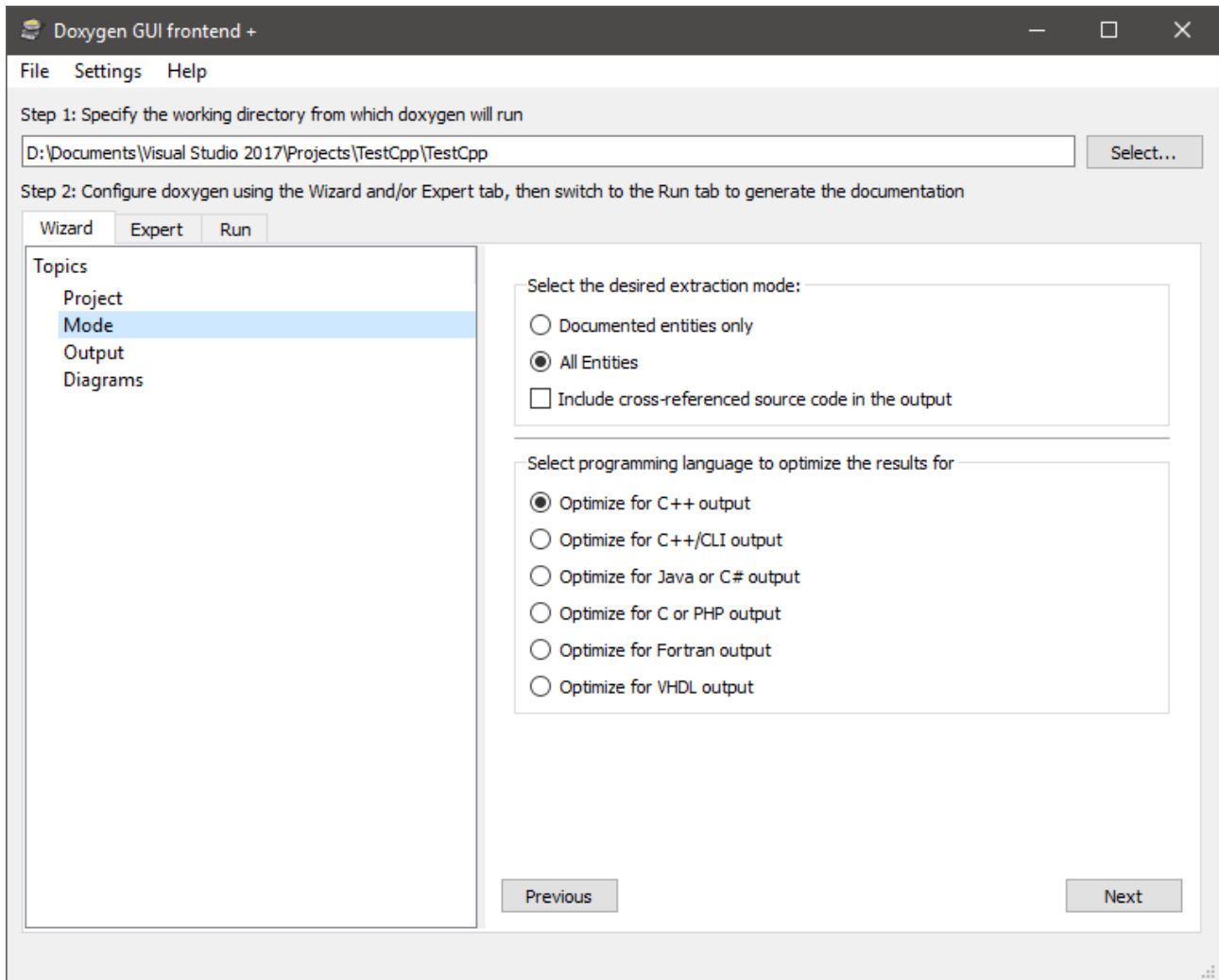


FIGURE IV.3.2. – Étape « Mode »

La deuxième étape est celle nommée « *Mode* », où l'on va indiquer que c'est du C++ (sélectionner *Optimize for C++ output*) et qu'on veut tout extraire (sélectionner *All Entities*).

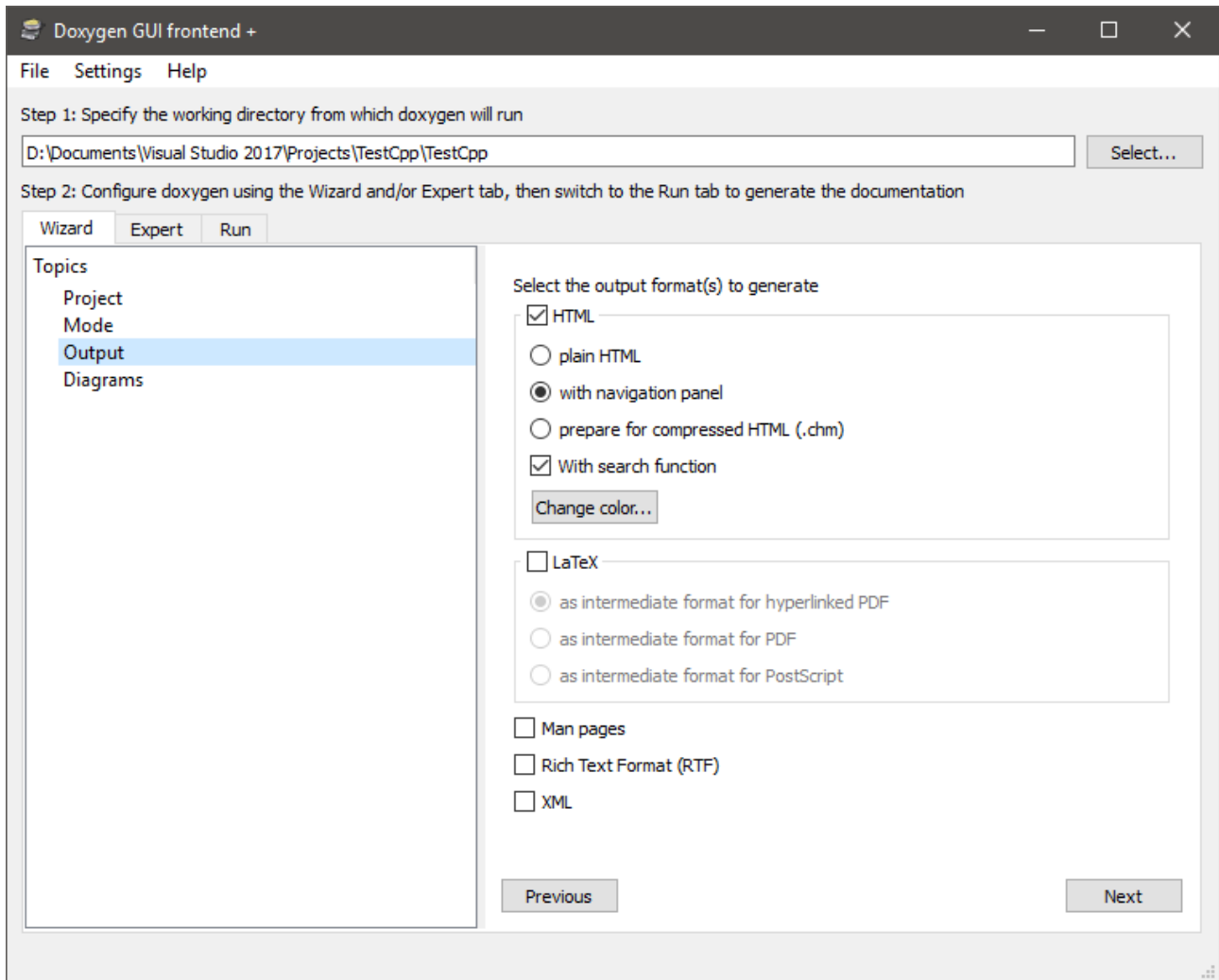


FIGURE IV.3.3. – Étape « Output »

Dans la troisième étape, « *Output* », ne cochez que **HTML** et sélectionnez *with navigation panel*, parce que les autres options de sortie ne nous intéressent pas.

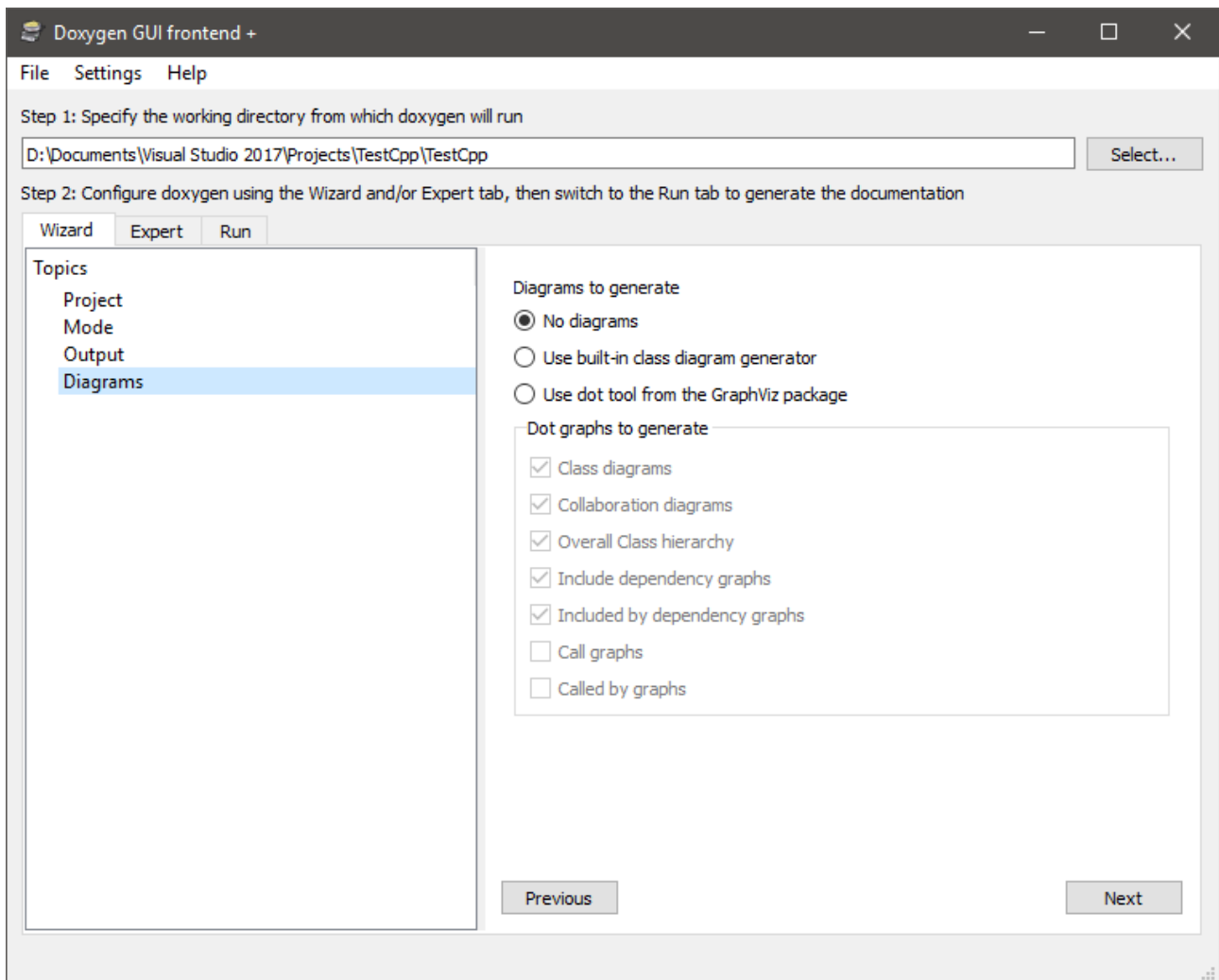


FIGURE IV.3.4. – Étape « Diagrams »

Enfin, dernière étape, «*Diagrams*», sélectionnez *No diagrams* et tout est bon. On peut passer à l'onglet «*Run*», d'où nous allons lancer la génération de la documentation en cliquant sur *Run doxygen*. Une fois cette étape faite, un dossier **html** sera présent.

i

Sauvegarder

En quittant l'application *DoxyWizard*, on vous demandera, si vous souhaitez sauvegarder cette configuration quelque part. Quand vous relancerez l'application, il suffira de recharger cette configuration pour gagner du temps.

Par défaut, la documentation est générée en anglais, mais vous pouvez mettre la langue de votre choix en cliquant sur **Expert**, puis **Project**. Là, mettez l'option **OUTPUT_LANGUAGE** à la valeur que vous décidez, **French** pour le français.

IV.3.2.2. Écrire la documentation

Mais concrètement, comment on écrit la documentation? En utilisant des **balises spéciales**, que Doxygen comprendra et qu'il traitera de façon appropriée. Ainsi, on a des attributs spéciaux

IV. Interlude - Être un développeur

pour décrire les arguments d'une fonction, sa valeur de retour, la présence de *bugs*, etc. La liste [est longue](#) ⁴, donc je n'en décrirai que certains. Libre à vous d'explorer la documentation de Doxygen pour découvrir les autres.

À noter également qu'on peut utiliser le Markdown⁴ dans les blocs de commentaire, pour entourer un morceau de code, mettre en gras, etc.

Comment fait-on? Plusieurs façons de faire sont possibles; j'ai fait le choix de vous en montrer deux. D'abord, soit la forme `/** Commentaire */`, soit la forme `/// Commentaire`. On retrouve ici un parallèle avec les commentaires `/* */` et `//`. C'est une question de choix. Personnellement, quand le commentaire fait plus d'une ligne, j'utilise la forme longue `/** */`, sinon les trois slashes `///`.

IV.3.2.2.1. Description d'une fonction

Et si nous reprenions notre fonction d'entrée sécurisée pour illustrer cette partie? Je vous remets ci-dessous la dernière version, utilisant les *templates*.

```
1  template <typename T, typename Predicat>
2  void entree_securisee(T & variable, Predicat predicat)
3  {
4      while (!(std::cin >> jour) || !predicat(variable))
5      {
6          if (std::cin.eof())
7          {
8              throw std::runtime_error("Le flux a été fermé !");
9          }
10         else if (std::cin.fail())
11         {
12             std::cout << "Entrée invalide. Recommence." <<
13                 std::endl;
14             std::cin.clear();
15
16             std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
17                 '\n');
18         }
19         else
20         {
21             std::cout << "Le prédicat n'est pas respecté !" <<
22                 std::endl;
23         }
24     }
25 }
```

Une des premières choses que nous voulons faire, c'est **d'expliquer ce que fait la fonction**, la raison de son existence. Pour cela, Doxygen fournit deux attributs.

4. Comme sur Zeste de Savoir.

IV. Interlude - Être un développeur

- `@brief` sert à **décrire brièvement** la fonction. En une ou deux phrases, on décrit la fonction et ce qu'elle fait.
- `@details` sert à **décrire en détails** la fonction. Ici, on entre dans les détails, en expliquant ce qu'elle fait, ce qu'elle ne fait pas, les exceptions qu'elle peut lever potentiellement, etc.

Appliquons donc ce que nous venons d'apprendre à notre code. J'ai écrit un certain texte, mais vous pouvez très bien mettre le vôtre.

```
1  /**
2  * @brief Saisie sécurisée avec `std::cin`.
3  * @details Vérifie que la variable demandée est du bon type et
4  *         respecte un prédicat donné.
5  * En cas de fermeture du flux d'entrée, une exception
6  *         `std::runtime_error` est levée.
7  * En cas de type invalide, le flux d'entrée est vidé.
8  * En cas de prédicat non respecté, un message d'erreur est affiché
9  *         sur la sortie standard.
10 */
11 template <typename T, typename Predicat>
12 void entree_securisee(T & variable, Predicat predicat)
13 {
14     // Le code qui ne change pas.
15 }
```

Si vous naviguez dans les fichiers HTML créés, vous allez trouver, dans la documentation du fichier C++ où se trouve la fonction (`main.cpp` par exemple), un encart présentant la fonction avec sa signature, une description courte puis une plus détaillée.

IV.3.2.2.2. Les paramètres

Maintenant, attaquons un autre morceau très important, à savoir **la description des paramètres**. Il est important que l'utilisateur sache ce que signifie chaque paramètre et s'il y a des préconditions à respecter. Ça tombe bien, on a des attributs spécifiques pour ça.

- `@param[type] paramètre` permet de **décrire un paramètre** de manière générale. Le type entre crochet peut prendre trois valeurs: `in` si le paramètre est une valeur lue mais non modifiée par la fonction, `out` si le paramètre est modifié par la fonction et `in,out` dans le cas où le paramètre est lu mais également modifié.
- `@pre` décrit **une précondition par ligne**. Si plusieurs attributs `@pre` sont présents les uns après les autres, ils sont regroupés dans un même bloc à la sortie.
- `@post` décrit **une postcondition par ligne**. Si plusieurs attributs `@post` sont présents les uns après les autres, ils sont regroupés dans un même bloc à la sortie.
- Parce que les *templates* n'échappent pas à la règle, `@tparam paramètre` permet de faire la même opération pour nos chers patrons.

Ceci appliqué à notre code, on obtient quelque chose comme ci-dessous. Pareil, le texte que vous avez mis peut varier.

```
1  /**
2  * @brief Saisie sécurisée avec `std::cin`.
3  * @details Vérifie que la variable demandée est du bon type et
4  *         respecte un prédicat donné.
5  * En cas de fermeture du flux d'entrée, une exception
6  * `std::runtime_error` est levée.
7  * En cas de type invalide, le flux d'entrée est vidé.
8  * En cas de prédicat non respecté, un message d'erreur est affichée
9  * sur la sortie standard.
10 *
11 * @tparam T Le type de la valeur à récupérer.
12 * @tparam Predicat Un prédicat à passer pour personnaliser la
13 *                 vérification.
14 *
15 * @param[out] variable Une référence sur la variable de type `T` à
16 *                    récupérer.
17 * @param[in] predicat Un prédicat que l'entrée saisie doit
18 *                    respecter.
19 *
20 * @pre Le type `T` doit être utilisable avec `std::cin`.
21 * @pre Le prédicat doit être de la forme `bool predicat(T
22 *      variable)`.
23 * @post La variable sera initialisée correctement avec la valeur
24 *       entrée.
25 */
26 template <typename T, typename Predicat>
27 void entree_securisee(T & variable, Predicat predicat)
28 {
29     // Le code qui ne change pas.
30 }
```

IV.3.2.2.3. Les exceptions

Notre fonction peut lever des exceptions, que ce soit parce que nous en lançons nous-mêmes explicitement, ou bien parce qu'une des fonctions appelées lance une exception en interne. Il est donc de bon ton de décrire ce qui peut potentiellement arriver. Bien sûr, être totalement exhaustif est impossible. Mais **décrire les exceptions que nous lançons explicitement** est une bonne habitude à prendre. Cela se fait très simplement avec l'attribut `@exception` `exception_lancée`.

```
1  /**
2  * @brief Saisie sécurisée avec `std::cin`.
3  * @details Vérifie que la variable demandée est du bon type et
4  *         respecte un prédicat donné.
```

```
4 * En cas de fermeture du flux d'entrée, une exception
   `std::runtime_error` est levée.
5 * En cas de type invalide, le flux d'entrée est vidé.
6 * En cas de prédicat non respecté, un message d'erreur est affiché
   sur la sortie standard.
7 *
8 * @tparam T Le type de la valeur à récupérer.
9 * @tparam Predicat Un prédicat à passer pour personnaliser la
   vérification.
10 *
11 * @param[out] variable Une référence sur la variable de type `T` à
   récupérer.
12 * @param[in] predicat Un prédicat que l'entrée saisie doit
   respecter.
13 *
14 * @pre Le type `T` doit être utilisable avec `std::cin`.
15 * @pre Le prédicat doit être de la forme `bool predicat(T
   variable)`.
16 * @post La variable sera initialisée correctement avec la valeur
   entrée.
17 *
18 * @exception std::runtime_error Si le flux d'entrée est fermé.
19 */
20 template <typename T, typename Predicat>
21 void entree_securisee(T & variable, Predicat predicat)
22 {
23     // Le code qui ne change pas.
24 }
```

IV.3.2.2.4. Le type de retour

Nous n'en avons pas dans notre cas, car la fonction `entree_securisee` ne retourne rien. Certains aiment utiliser l'attribut `@returns` même dans ce cas, d'autres jugent que c'est inutile. Je vous laisse choisir. Voici néanmoins un exemple d'utilisation de cet attribut, avec une fonction que nous avons écrite dans le chapitre consacré (et légèrement modifiée).

```
1 /** @brief Détermine si une expression est correctement
   parenthésée.
2 * @details La fonction regarde si on a bien le même nombre de
   parenthèses
3 * ouvrantes et fermantes et si celles-ci sont bien ordonnées.
4 *
5 * @param[in] expression Une chaîne de caractères à analyser.
6 *
7 * @returns `true` si l'expression est bien parenthésée, `false`
   sinon.
```

```

8  */
9  bool parentheses(std::string const & expression)
10 {
11     int ouvrantes { 0 };
12     int fermantes { 0 };
13
14     for (auto const & caractere : expression)
15     {
16         if (caractere == '(')
17         {
18             ++ouvrantes;
19         }
20         else
21         {
22             ++fermantes;
23         }
24
25         if (fermantes > ouvrantes)
26         {
27             return false;
28         }
29     }
30
31     return ouvrantes == fermantes;
32 }

```

IV.3.2.2.5. D'autres attributs utiles

Terminons cet aperçu des attributs de documentation d'une fonction en en présentant quelques-uns qui permettent de signaler des informations supplémentaires, ne rentrant pas dans le cadre des attributs vus plus haut. Il arrive, par exemple, qu'on veuille signaler la présence d'un *bug* encore non corrigé ou rarissime, d'une modification ou amélioration prévue mais pas encore faite, d'un avertissement concernant un aspect particulier du code, etc.

- `@bug` sert à **décrire la présence d'un *bug*** dans le code. Cet attribut aurait pu, par exemple, être utilisé dans les anciens T.P où nous ne pensions pas à prendre en compte la fermeture potentielle du flux. On aurait pu ajouter un commentaire du type `@bug Si le flux est fermé, la fonction rentre dans une boucle infinie.`
- `@todo` sert à **décrire une action à faire plus tard**. On pourrait s'en servir pour se rappeler, dans notre T.P, de mettre à jour le code pour utiliser un flux d'entrée générique à la place de `std::cin`.
- `@note` permet d'**ajouter des informations génériques**. Si votre code implémente une fonction mathématique, vous aurez peut-être envie de l'écrire clairement. Ce bloc est tout à fait adapté à ça.
- `@warning` permet d'**ajouter des avertissements**. On peut vouloir prévenir, par exemple, qu'une fonction ne doit pas être utilisée en même temps qu'une autre, où bien qu'un paramètre doit être d'abord correctement initialisé sous peine de voir le programme planter.

IV.3.2.2.6. Les structures et les énumérations

Bien évidemment, il ne faut pas oublier de commenter nos types personnalisés. Non seulement le type lui-même, mais chacun des champs. Pour cela, nous disposons des attributs `@struct` et `@enum`. Chacun fonctionne sur le même principe.

- D’abord, on écrit **l’attribut puis le nom du type**.
- Ensuite, une **description brève**.
- Enfin, une **description plus détaillée**.

De même, chaque champ peut faire l’objet de description courte, détaillée, d’avertissements, etc. En général, seul une description succincte est nécessaire. Examinez par vous-mêmes un exemple de structure et d’énumération documentées.

```
1  /** @enum Couleur
2  * @brief Liste des couleurs possibles.
3  * @details La liste des trois couleurs applicables à l'écran de
4  *         fond du programme.
5  */
6  enum class Couleur
7  {
8      /// @brief Un bleu profond.
9      Bleu,
10     /// @brief Un rouge vif.
11     Rouge,
12     /// @brief Un vert magnifique.
13     Vert
14 };
15 /** @struct InformationsPersonnelles
16 * @brief Les informations personnelles d'un membre.
17 * @details Permet de stocker, sous un même type, le nom/prénom
18 *         ainsi que le genre et l'âge d'un membre.
19 * Il n'y a pas de limite à la taille du nom ou du prénom du membre.
20 */
21 struct InformationsPersonnelles
22 {
23     /// @brief Le prénom du membre.
24     std::string prenom;
25     /// @brief Le nom de famille du membre.
26     std::string nom;
27     /// @brief Femme ou homme ?
28     std::string sexe;
29     /// @brief L'âge du membre.
30     int age;
31 };
```


IV.3.3. Quelques bonnes pratiques

Vous savez maintenant documenter votre code et c'est une excellente chose. Une bonne documentation fait gagner du temps et est la meilleure alliée d'un développeur découvrant un nouveau code. Mais doit-on tout documenter? Absolument tout? **Y a t-il des bonnes pratiques?**

IV.3.3.1. Ce qu'il faut documenter

Dans l'idéal, **toutes vos fonctions devraient être documentées**. Utilisez Doxygen pour détailler vos fonctions, leur utilisation, ce qu'elles attendent, les exceptions qu'elles peuvent lancer, etc. C'est assez fastidieux, je vous l'accorde. Mais ne pas le faire est une très mauvaise pratique. Qui n'a jamais pesté contre un appareil qu'il a dû utiliser sans manuel ni instructions? Si quelqu'un vient à utiliser votre code, il n'aimera pas non plus avoir à deviner ce que celui-ci fait, les exceptions qu'il lance, etc.

?

Mais si je suis seul, c'est pas un peu inutile?

Cela vous servira même si vous êtes seul. Ne croyez pas que vous vous souviendrez de ce que font vos 4000 lignes de code quand vous rouvrirez le projet après plusieurs mois d'absence s'il n'y a pas de documentation. Ce qui vous semble clair maintenant ne le sera peut-être pas plus tard. Épargnez-vous des problèmes, documentez. Pour ne pas vous démotiver face à l'immensité de cette tâche, commencez tôt et faites en petit à petit. Ainsi, tandis que votre base de code augmentera, elle sera bien documentée et facile à comprendre.

IV.3.3.2. Les fichiers d'en-tête ou source ?

Nous n'aimons pas la recopie. C'est une source d'erreurs et d'incompréhensions. Imaginez que la documentation d'un prototype dans un fichier d'en-tête soit légèrement différente de celle de son implémentation dans le fichier source. Qui a raison? Qui est à jour? Un vrai nœud à problèmes que de maintenir deux fois la même documentation à jour.

Heureusement, Doxygen est assez intelligent et affiche la documentation du fichier d'en-tête même si on demande à voir celle du fichier source. Il comprend qu'on a affaire aux mêmes fonctions, avec d'un côté le prototype et de l'autre l'implémentation.

IV.3.3.2.1. Exemple

Parfois, certaines fonctions sont assez compliquées à utiliser et avoir un exemple fonctionnel sous les yeux nous aide vraiment. Vous-mêmes l'avez certainement remarqué en vous plongeant dans la documentation C++. **Un exemple, ça aide**. Voilà pourquoi j'ai choisi de séparer cette section du reste de la présentation de Doxygen, pour que vous compreniez bien que c'est important.

Une façon de faire consiste à utiliser l'attribut `@note`, pour bien différencier des autres sections, puis à écrire un petit exemple entre les attributs `@code` et `@endcode`.

```
1  /** @brief Détermine si une expression est correctement
2      * @details La fonction regarde si on a bien le même nombre de
3      * ouvrantes et fermantes et si celles-ci sont bien ordonnées.
4      *
5      * @param[in] expression Une chaîne de caractères à analyser.
6      *
7      * @returns `true` si l'expression est bien parenthésée, `false`
8      * sinon.
9      *
10     * @note Voici un exemple d'utilisation.
11     * @code
12     * std::string const test { "((()))" };
13     * parentheses(test);
14     * @endcode
15     */
16     bool parentheses(std::string const & expression)
17     {
18         // Implémentation.
```

IV.3.3.3. Commentaire vs documentation

La découverte et l'utilisation de Doxygen signifie t-elle la fin des commentaires comme nous avons appris dès le chapitre introductif à C++? Bien sûr que non. En fait, les deux ne sont pas opposés, simplement, **les commentaires ne servent pas les mêmes buts**.

La documentation a pour but d'être, si possible, **exhaustive** et de **décrire le quoi, le comment**. On s'en sert pour expliquer ce que fait une fonction, les paramètres qu'elle attend, ce qu'elle renvoie, les exceptions qu'elle peut lever, ses contrats, etc. On s'en sert aussi pour illustrer le fonctionnement du code par des exemples, comme vu plus haut. **Si la documentation est bien faite, un développeur n'aura jamais besoin d'aller lire le code source pour comprendre**.

Les commentaires eux, sont plutôt à **usage interne** et **expliquent le pourquoi**. On s'en sert pour **expliciter ce que le code ne dit pas**, pour décrire les raisons qui ont poussé à faire ou ne pas faire telle ou telle chose. Ils seront donc moins présents certes, mais pas moins utiles.

IV.3.3.3.1. De bons commentaires...

Par exemple, on peut vouloir expliquer le choix d'une structure de données qui sort de l'ordinaire. Vous rappelez-vous de `std::map` et `std::unordered_map`? La deuxième est souvent choisie pour des raisons de performances. Ainsi, vous voudrez peut-être expliquer pourquoi vous avez choisi la première avec un bref commentaire.

IV. Interlude - Être un développeur

```
1 // Utilisation de `std::map` car les clés doivent rester triées
  dans l'ordre.
2 using Data = std::map<std::string, int>;
```

On peut aussi s'en servir pour justifier un code un peu sale ou exotique.

```
1 int const x { 0 };
2 // Le +1 est nécessaire car fonction_externe est bugué quand x vaut
  0.
3 int resultat { fonction_externe(x) + 1 };
```

De la même manière, si vous voulez tester si un nombre N est premier [↗](#) ou pas, vous pouvez appliquer une astuce consistant à ne tester que de 2 à \sqrt{N} . Cela peut faire l'objet d'un commentaire explicatif.

```
1 int const N { 24 };
2 bool est_premier { true };
3
4 // On ne teste que jusqu'à racine de N pour gagner du temps.
5 for (int i { 2 }; i < std::sqrt(N); ++i)
6 {
7     if (N % i == 0)
8     {
9         est_premier = false;
10        break;
11    }
12 }
```

Ces commentaires sont utiles, car seulement en lisant le code, vous n'auriez pas compris **pourquoi** il a été écrit de cette façon.

IV.3.3.3.2. ...et de mauvais commentaires

Malheureusement, ce qu'on voit le plus souvent, ce sont les mauvais commentaires. Que ce soit parce qu'ils se contentent de redire ce que dit déjà le code, parce que le code est mal écrit, ou bien encore parce qu'ils ne sont pas clairs, le fait est **qu'ils n'apportent rien de bon**. Un exemple? La plupart des commentaires écrits dans ce cours répètent ce que le code dit. Bien sûr, dans votre cas, c'est nécessaire car vous êtes en plein apprentissage. Par contre, pour des développeurs chevronnés, ce genre de commentaires est inutile.

Un autre type de commentaire problématique, c'est celui qui explique le code parce que celui-ci n'est pas clair. Regardez la façon dont cela est illustré dans l'exemple suivant.

```
1 // Si la vie atteint une valeur critique (<100).
2 if (vie < 100)
3 {
4     // Du code.
5 }
```

Le code pourrait être amélioré en utilisant une constante, ce qui supprimerait le commentaire au passage.

```
1 int const niveau_vie_critique { 100 };
2
3 // ...
4
5 if (vie < niveau_vie_critique)
6 {
7     // Du code.
8 }
```

Dans le même genre, les commentaires signalant une accolade fermante, ceux signalant la date de dernière modification, ceux retraçant l'historique des modifications d'un fichier, tout ça fait partie des commentaires qui n'ont pas lieu d'être.



Ressources supplémentaires

La distinction peut vous sembler dur à comprendre et la frontière entre un bon et un mauvais commentaire peut paraître bien mince. Une littérature informatique abondante aborde ce sujet, que nous ne pouvons, dans le cadre de ce cours, parcourir exhaustivement. Si vous souhaitez aller plus loin, un chapitre du livre [Coder proprement](#) y est dédié.

IV.3.3.4. En résumé

- La documentation est une des meilleures amies du développeur.
- La documentation C++ est classée en divers thèmes, allant du langage lui-même aux algorithmes, en passant par `std::string` et les fonctions mathématiques.
- On retrouve souvent les mêmes sections, décrivant les différents prototypes, les paramètres, les exceptions possibles, ainsi que des exemples concrets.
- Doxygen nous permet d'écrire la documentation de nos programmes. C'est une bonne habitude à prendre.
- Certaines bonnes pratiques peuvent nous aider à écrire des commentaires et de la documentation de qualité.

Contenu masqué

Contenu masqué n°66 :

Correction

Puisqu'on parle de chaînes de caractères, on va donc chercher du côté de [basic_string](#) . On remarque, dans la liste des fonctions disponibles, une certaine [replace](#) . Le premier prototype est justement celui qui répond à l'énoncé. On obtient donc ce qui suit.

```
1 #include <cassert>
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     std::string phrase { "J'aimerais annuler ce texte." };
8     std::string const mot { "remplacer" };
9
10    // Le code.
11    phrase.replace(10, 7, mot);
12
13    std::cout << phrase << std::endl;
14    assert(phrase == "J'aimerais remplacer ce texte." &&
15           "Le texte n'a pas été modifié correctement.");
16    return 0;
17 }
```

[Retourner au texte.](#)

Contenu masqué n°67 :

Indice

Il faut chercher dans les [fonctions mathématiques](#) .

[Retourner au texte.](#)

Contenu masqué n°68 :

Correction

En parcourant la liste des fonctions contenues sur cette page de documentation, on note le triplet `hypot`, `hypotf` et `hypotl`. La description associée dit que ces fonctions calculent la racine carrée de la somme des carrés de deux nombres, soit $\sqrt{x^2 + y^2}$. Bien que, de par son nom, cette fonction semble plutôt destinée à calculer une hypoténuse, elle se révèle quand même adaptée. La correction est toute simple.

```
1 #include <cassert>
2 #include <cmath>
3
4 int main()
5 {
6     int const x { 5 };
7     int const y { 12 };
8     int norme { 0 };
9
10    // Le code pour résoudre l'exercice.
11    norme = std::hypot(x, y);
12
13    assert(norme == 13 &&
14           "La norme d'un vecteur v(5;12) vaut 13.");
15    return 0;
16 }
```

[Retourner au texte.](#)

Contenu masqué n°69 : Indice

Vous devez chercher du côté de la page [complex](#) .

[Retourner au texte.](#)

Contenu masqué n°70 : Correction

Alors, on arrive sur la fameuse page en question, où l'on découvre un nouveau type, `complex`. C'est un *template*, donc on va devoir lui passer un type, en l'occurrence `double`.

On voit ensuite, plus bas, la fonction `sqrt` , qui répond à la première partie de l'exercice. C'est une fonction non-membre, donc qui s'utilise en se préfixant de `std::`.

Plus haut, dans les fonctions membres, on voit `real` et `imag` , qui donnent respectivement la partie réelle et la partie imaginaire de notre résultat complexe. On a donc tout ce qu'il nous faut.

```
1 #include <complex>
2 #include <iostream>
3
4 int main()
5 {
6     auto resultat = std::sqrt(std::complex<double>{ -1, 0 });
```

```
7     std::cout << "La racine carrée de -1 donne (" <<
      resultat.real() << ";" << resultat.imag() << "i)." <<
      std::endl;
8     return 0;
9 }
```

[Retourner au texte.](#)

Contenu masqué n°71 : Correction

Qui dit `algorithm` dit bien souvent le fichier correspondant `<algorithm>` [↗](#). C'est effectivement dedans que nous allons trouver ce qu'il nous faut et l'algorithme en question se nomme `std::transform` [↗](#). Avez-vous écrit un code comme celui-ci?

```
1 std::transform(std::begin(phrase), std::end(phrase),
2               std::begin(sortie), [](char c) -> char
3               {
4                   return std::toupper(c);
5               });
```

Si oui, alors vous n'avez pas bien lu la documentation. En effet, lancez ce programme et il plantera. Pourquoi? Parce que la chaîne de sortie est vide, donc on ne peut pas modifier les éléments d'une chaîne vide puisqu'il n'y en a pas. La documentation apporte une précision.

`std::transform` applies the given function to a range and stores the result in another range, beginning at `d_first`.

Documentation de `std::transform`.

Il faut que le conteneur de sortie ait la même taille que celui d'entrée. Ou, tout du moins, en théorie. Si vous lisez le code d'exemple situé en fin de page, vous allez remarquer la présence d'une fonction inconnue jusque-là: `std::back_inserter`. En lisant [sa documentation](#) [↗](#), on apprend qu'elle crée un `std::back_inserter_iterator` [↗](#).

Qu'à cela ne tienne, allons lire la documentation de ce nouveau genre d'itérateur. Celle-ci nous donne une information très intéressante.

`std::back_inserter_iterator` is an `LegacyOutputIterator` that appends to a container for which it was constructed. The container's `push_back()` member function is called whenever the iterator (whether dereferenced or not) is assigned to.

Documentation de `std::back_inserter_iterator`.

Traduit, cela signifie qu'à chaque fois que l'itérateur est modifié, la fonction `push_back` du conteneur en question est appelée. C'est exactement ce qu'il nous faut pour notre conteneur de sortie vide. Voici donc le code final.

```
1 #include <algorithm>
2 #include <cctype>
3 #include <iostream>
4 #include <string>
5
6 int main()
7 {
8     std::string const phrase {
9         "Voici une phrase longue que je veux en majuscules." };
10    std::string sortie { "" };
11    std::transform(std::begin(phrase), std::end(phrase),
12                  std::back_inserter(sortie), [](char c) -> char
13    {
14        return std::toupper(c);
15    });
16    std::cout << sortie << std::endl;
17    return 0;
18 }
```

Cet exercice vous a semblé plus difficile? C'était le cas. Quand on ne connaît pas quelque chose, bien souvent, une seule page de documentation n'est pas suffisant. Il faut parfois faire des recherches complémentaires. Heureusement pour vous, la documentation C++ présente sur ce site est accompagnée d'exemples qui vous aideront à bien saisir les difficultés et particularités potentielles. 🍊

[Retourner au texte.](#)

IV.4. Compilation en cours...

Depuis le début de ce tutoriel, nous avons compilé et recompile de nombreuses fois nos programmes. Vous savez, depuis l'introduction, que la compilation transforme notre code source en un programme exécutable qui sera compris par l'ordinateur. Mais, concrètement, que se passe-t-il pendant la compilation? Quelles en sont les différentes étapes?

Le but de ce chapitre va être de **présenter en détails la compilation d'un code C++**.

IV.4.1. Le préprocesseur

La toute première étape est celle du **préprocesseur**. Ce nom vous est familier? Normal, on fait déjà appel à lui pour chaque ligne qui commence par `#`.

IV.4.1.1. Inclure des fichiers

Dans le cas de la directive `#include`, il s'agit d'inclure tout le fichier demandé. Il est possible de voir à quoi ressemble notre code après son passage.

- Avec **Visual Studio**, il faut faire un clic-droit sur le nom du projet et cliquer sur **Propriétés**. Là, on va dans **Propriétés de configuration -> C/C++ -> Préprocesseur**, puis on met le champ **Prétraiter dans un fichier** à **Oui (/P)**. Attention cependant, **cette option bloque la suite de la compilation**. Donc désactivez la sortie du préprocesseur une fois que vous avez fini d'explorer cette option.
- Avec **QtCreator**, il faut modifier le fichier **.pro** en lui ajoutant la ligne `QMAKE_CXXFLAGS += -save-temps`. Le fichier obtenu après traitement a une extension en **.ii**.
- Pour ceux qui travaillent en ligne de commande, pour GCC c'est `g++ -E fichier.cpp -o fichier.i` et pour Clang, `clang++ -E fichier.cpp -o fichier.i`.

Le résultat n'est pas le même en fonction du compilateur, mais **le fichier devient conséquent** après le passage du préprocesseur. Pour un simple `#include <iostream>`, j'obtiens plus de 17.800 lignes pour Clang, plus de 18.000 lignes avec GCC et même plus de 50.000 lignes avec Visual Studio (dont pas mal vides, certes)!

IV.4.1.2. Conditions

Mais le préprocesseur ne se contente pas d'inclure bêtement ce qu'on lui demande et c'est tout. Il est capable d'exécuter d'autres instructions, notamment celles permettant d'éviter les inclusions multiples. Ce sont des **instructions conditionnelles** pour le préprocesseur. Je vous remets un exemple ci-dessous.

```
1 #ifndef TEST_HPP
2 #define TEST_HPP
3
4 #endif
```

La première ligne teste **si une constante de préprocesseur `TEST_HPP` existe**. Si ce n'est pas le cas, alors on exécute le code qui suit. Justement, la ligne d'après définit la constante `TEST_HPP`. Ainsi, si l'on tente d'inclure de nouveau le fichier, la première condition devient fausse et rien n'est inclus. Enfin, l'instruction `#endif` permet de fermer la portée du `#ifndef`, tout comme les accolades fermantes pour `if`.

Ce mécanisme de conditions est très utilisé au sein des fichiers d'en-tête de la bibliothèque standard. En effet, cela permet d'**adapter le code en fonction de la version du compilateur ou de la norme C++**, entre autres. Par exemple, la bibliothèque standard fournie avec Visual Studio utilise le code suivant pour ne définir la fonction `std::for_each_n` [↗](#) que si l'on compile en C++17.

```
1 #if _HAS_CXX17
2
3 // Le code de la fonction std::for_each_n, disponible uniquement à
  partir de C++17.
4
5 #endif /* _HAS_CXX17 */
```

IV.4.1.3. *Debug* ou *release* ?

Le préprocesseur est également utilisé pour **modifier la bibliothèque standard selon qu'on compile en *debug* ou en *release***. Il s'agit de modes de compilation qui sont utilisés en fonction du moment.

IV.4.1.3.1. *Debug*— Quand on code

Le mode ***debug*** inclut, dans le code de la bibliothèque standard, de **nombreuses vérifications et autres tests**. C'est le mode dans lequel est compilé le code quand on est encore en plein développement, qu'on cherche à corriger des *bugs* ou qu'on fait des tests. Depuis le début du cours, nous ne compilons qu'en mode *debug*. C'est pour cette raison que, quand nous utilisons un indice trop grand, le programme plante avec un message d'erreur.

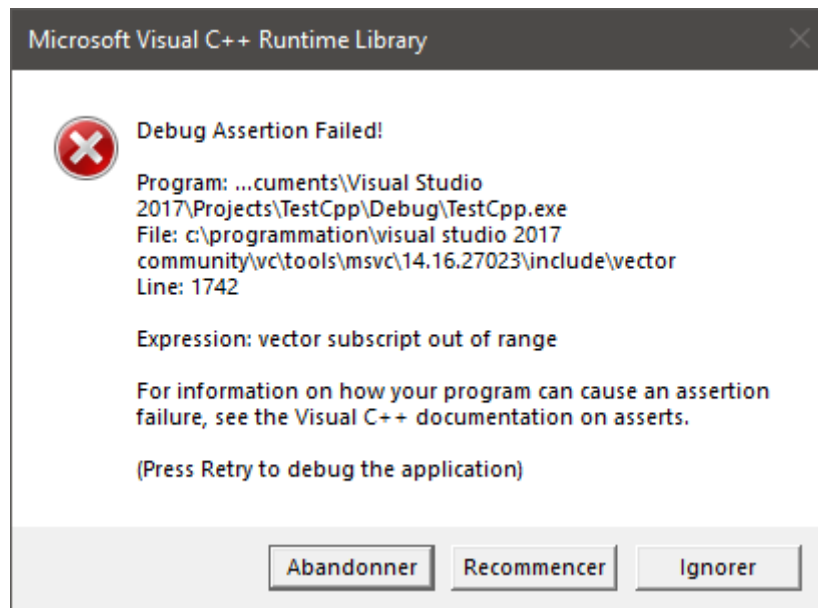


FIGURE IV.4.1. – Erreur affichée en mode debug.

Si on regarde le code de `std::vector` tel qu'il est fourni avec Visual Studio, on trouve les lignes suivantes. On retrouve le message qu'on voit dans la fenêtre ci-dessus.

```
1 #if _ITERATOR_DEBUG_LEVEL != 0
2     _STL_VERIFY(_Pos < size(),
3     "vector subscript out of range");
3 #endif /* _ITERATOR_DEBUG_LEVEL != 0 */
```

La macro `_ITERATOR_DEBUG_LEVEL` est elle-même définie uniquement si une macro `_HAS_ITERATOR_DEBUGGING` est définie comme valant autre chose que `0`. Cette dernière dépend de l'existence de la macro `_DEBUG`, qui est uniquement définie quand on compile en mode *debug*.

IV.4.1.3.2. Release—Quand on distribue

Le mode *release*, lui, se débarrasse de toutes ces vérifications, optimise le code pour le rendre plus rapide à exécuter et réduit la taille de l'exécutable qu'on obtient en sortie. On compile avec ce mode quand le programme est prêt et qu'on veut le distribuer à d'autres personnes. À ce stade, on juge que le programme est suffisamment bon et correctement testé. Il est donc inutile de continuer à inclure tous les tests et autres vérifications, qui rendent le code plus gros et plus lent.

Typiquement, les assertions sont désactivées en *release*. Bien souvent, `assert` n'est pas implémentée en tant que fonction, comme nous en avons l'habitude, mais à l'aide du préprocesseur, ce qui permet de la supprimer du code, ou de la rendre inopérante, quand on passe en *release*.

IV.4.2. La compilation

Maintenant que le préprocesseur a fini son travail, place à la compilation à proprement parler. Chaque fichier obtenu après le passage du préprocesseur, qu'on appelle **unité de compilation**, est traité individuellement. Le but de cette phase est de **transformer le C++ en code exécutable** par l'ordinateur.

C'est lors de cette étape que **beaucoup de vérifications ont lieu** et donc d'erreurs. Par exemple, dans le cas où le compilateur ne trouve pas la déclaration d'une fonction que vous appelez, il décidera de faire s'arrêter la compilation. Vous aurez alors un message d'erreur vous indiquant ce qui n'a pas fonctionné.

IV.4.2.1. Les templates

Comme dit dans le chapitre correspondant, le compilateur va **instancier chaque *template*** qu'il trouve avec les bons types. C'est pendant cette étape que des erreurs incompréhensibles de *templates* peuvent arriver, par exemple quand on tente d'appliquer `std::sort` sur une `std::list`.

Prenez l'exemple suivant d'un code C++ très simple.

```
1  template <typename T>
2  void fonction(T const & t)
3  {
4      T const resultat { t * 2 };
5  }
6
7  int main()
8  {
9      fonction(42);
10     fonction(3.1415);
11     return 0;
12 }
```

Avec Clang, il est possible d'utiliser les options `-Xclang -ast-print -fsyntax-only` pour afficher le code après l'instanciation des templates. Avec GCC, c'est la commande `-fdump-tree-original` qui génère des fichiers `.original` qu'on peut afficher.

On voit dans le résultat suivant, obtenu avec Clang, que le *template* est instancié deux fois, avec `int` et avec `double`.

```
1  template <typename T = int> void fonction(const int &t) {
2      const int resultat{t * 2};
3  }
4
5  template <typename T = double> void fonction(const double &t) {
```

```
6      const double resultat{t * 2};
7  }
8
9  template <typename T> void fonction(const T &t) {
10      const T resultat{t * 2};
11  }
12
13  int main() {
14      fonction(42);
15      fonction(3.1415);
16      return 0;
17  }
```

IV.4.2.2. constexpr

Nous avons dit, dans le chapitre sur la documentation, que nous reviendrons sur ce mot-clé en temps voulu. C'est maintenant le moment d'en parler, car il a un rapport avec la compilation. En effet, il indique au compilateur qu'une **variable ou une fonction peut déjà être évaluée pendant la compilation** et non plus seulement à l'exécution du programme. L'avantage, c'est que c'est le compilateur qui effectue le travail une fois pour toutes, ce qui rend **l'exécution du programme plus rapide**.

IV.4.2.2.1. Des constantes

En déclarant des constantes avec le mot-clé `constexpr`, on les rend utilisables à la compilation. Typiquement, dans le cas où l'on fixe d'avance la hauteur et la largeur d'une matrice, utiliser `std::array` est tout à fait possible.

```
1  constexpr std::size_t hauteur { 5 };
2  constexpr std::size_t largeur { 4 };
3
4  std::array<int, hauteur * largeur> matrice {};
```



`constexpr` et `const`

Implicitement, utiliser `constexpr` sur une variable revient à la déclarer `const`, ce qui est logique. Personnellement, dans le but d'être explicite, je précise également `const`, mais ce n'est pas une obligation, simplement une question de goût.

Par contre, pour que l'instruction soit valide et que le programme puisse compiler, il faut qu'elle soit **évaluable à la compilation**. Cela signifie que toute l'expression doit être comprise et utilisable par le compilateur pendant la compilation et que rien ne doit dépendre de variables non-constantes, d'entrées/sorties, etc.

```

1  int main()
2  {
3      // Valide parce que 5 est une littérale évaluable à la
        compilation.
4      constexpr int const x { 5 };
5      // Valide parce que x est constexpr.
6      constexpr int const y { 2 * x };
7
8      // Invalide car b n'est pas une constante.
9      int b { 10 };
10     constexpr int const non { b - 5 };
11
12     return 0;
13 }

```

IV.4.2.2.2. Des fonctions

Grâce au mot-clé `constexpr`, nous sommes capables d'écrire des fonctions qui s'exécuteront **pendant la compilation**, ce dont nous étions pas capables jusque-là, tout du moins sans utiliser les *templates*. En plus, la syntaxe est très simple, puisqu'il suffit d'ajouter `constexpr` au prototype de la fonction.

```

1  constexpr int add(int a, int b)
2  {
3      return a + b;
4  }

```

Cela a néanmoins un effet un peu particulier.

- Si la fonction est appelée avec des arguments `constexpr`, comme des littéraux ou des variables `constexpr`, alors **elle est exécutée à la compilation**.
- Dans le cas contraire, avec des arguments non-`constexpr`, elle se comporte normalement et est exécutée **pendant l'exécution du programme**.

Prenons un exemple en calculant la [factorielle](#) d'un nombre entier. C'est une opération mathématique simple, qui se définit comme le produit de tous les entiers de 1 à n . La formule est la suivante.

$$n! = \prod_{i=1}^n i = 1 \times 2 \times \dots \times n$$

Ainsi, $2!$ vaut 2, mais $10!$ vaut déjà 3628800! C'est donc une opération qu'il convient de faire calculer au compilateur si cela est possible. En plus, son implémentation est toute simple, comme vous le montre le code suivant. Notez que dans le premier cas, la fonction est appelée pendant la compilation, alors que dans le deuxième cas, pendant l'exécution.

```
1 #include <array>
2 #include <iostream>
3
4 constexpr int factorielle(int n)
5 {
6     int resultat { 1 };
7     for (int i { 1 }; i <= n; ++i)
8     {
9         resultat *= i;
10    }
11    return resultat;
12 }
13
14 int main()
15 {
16     // Exécuté à la compilation.
17     std::array<int, factorielle(3)> tableau {};
18
19     // Exécuté quand le programme est lancé.
20     int entier { 0 };
21     std::cout << "Donne un entier entre 1 et 10 : ";
22     std::cin >> entier;
23     std::cout << "Sa factorielle vaut " << factorielle(entier) <<
24         "." << std::endl;
25
26     return 0;
27 }
```

?

Alors pourquoi ne pas déclarer toutes nos fonctions `constexpr`? Comme ça on gagne en performances.

Parce que `constexpr` est soumis à des restrictions. En C++11, il ne pouvait pas y avoir de boucles ou de `if / else`. Depuis C++14, ces restrictions ont été levées, ce qui permet l'implémentation de la factorielle telle que vous la voyez plus haut.

Et sans parler de ces considérations, une fonction ne peut être `constexpr` que si toutes ses instructions sont évaluables pendant la compilation. Cela nous interdit donc d'utiliser les entrées/sorties ou des fonctions qui ne sont pas `constexpr`. Voilà pourquoi toutes nos fonctions ne peuvent pas être exécutées pendant la compilation.

IV.4.2.3. La compilation à proprement parler

Maintenant, il s'agit de transformer tout ce code C++ en quelque chose de compréhensible par l'ordinateur. Lui ne comprend que les 0 et les 1. Il n'a que faire de nos noms de variables, de fonctions ou de type. La transformation du C++ en code machine se fait en plusieurs étapes. On peut en citer quelques-unes.

- **L'analyse lexicale**, où il déduit que `auto` ou `int` sont des mots-clés, `main` ou `x` sont des identificateurs, `=` ou `+` sont des opérateurs, etc.
- **L'analyse syntaxique**, où le compilateur regarde si la syntaxe fait sens et ce qu'elle signifie. Ainsi, écrire `auto = 5` sera rejeté pendant la compilation car cette syntaxe n'a aucun sens.
- **Les optimisations**, notamment en mode *release*. Cette étape supprime du code inutile, modifie certaines portions pour qu'elles s'exécutent plus vite, etc.
- **Génération du code pour la plateforme cible**. Le fichier objet ne sera pas pareil selon qu'on compile sous Windows ou GNU/Linux, donc le compilateur fait des adaptations pour la machine cible.



Note

Je ne détaille pas plus la compilation du C++ à proprement parler car c'est **affreusement compliqué**. C++ a la réputation d'être un langage dont l'écriture du compilateur est très compliquée et cette réputation n'est pas volée.

Le compilateur va au final générer des **fichiers objets**, un pour chaque fichier `.cpp`, chaque unité de compilation, qu'il trouve, comprenant son code propre et tous les fichiers d'en-têtes inclus. Si aucune modification n'est détectée depuis la dernière compilation, alors le compilateur réutilise le fichier objet précédemment créé, ce qui accélère la compilation d'un projet.

Un fichier objet, reconnaissable à son extension en `.o` ou `.obj`, n'est rien d'autre que **du code machine** mais seul, **sans aucun lien avec le reste du monde**. Tel quel, **il n'est pas exécutable**, car il lui manque des informations. Pour cela, il faut utiliser le **linker**, ce que nous allons voir dans la partie suivante.

Dans le cas de Visual Studio, ils sont présents dans le dossier de sortie. Dans le cas de GCC et Clang, il faut utiliser l'option `-c` lors de la compilation.

```
1 > g++ -std=c++17 -c test.cpp -o test.o
2 > clang++ -std=c++17 -c test.cpp -o test.o
```

IV.4.2.4. Une étape intermédiaire cachée

Beaucoup de compilateurs génèrent maintenant directement des fichiers objets, rendant invisible une étape intermédiaire, la **transformation de C++ en Assembleur** [↗](#), un langage de bas-niveau très proche du langage machine. Ce langage était ensuite passé à un **assembleur** ⁵, qui s'occupait de transformer ce code en fichier objet.

Il est cependant possible d'obtenir le code Assembleur généré avec la bonne option.

- Avec Visual Studio, il faut aller dans les propriétés du projet, puis dans `[C/C++] -> Fichiers de sortie`. Là, il faut mettre l'option `[Sortie de l'assembleur]` à **Assembleur, code machine et source (/FAcs)**.

5. Oui, il s'agit du même nom pour le langage et pour l'outil qui compile ce langage. On les différencie toutefois en employant un «A» majuscule pour parler du langage.

IV. Interlude - Être un développeur

- Avec Qt Creator, si vous avez déjà ajouté la ligne `QMAKE_CXXFLAGS += -save-temps` à votre fichier `.pro`, vous n'avez rien d'autre à faire. Le fichier assembleur est celui avec une extension `.s`.
- Avec GCC et Clang, il faut compiler avec l'option `-S`.

Attention, il s'agit d'un langage très... brut de décoffrage. Si vous ne me croyez pas, jetez un œil par vous-mêmes, vous allez être surpris. 🍊

IV.4.2.5. Influer sur la compilation

Il est possible de passer d'autres informations au compilateur, notamment pour lui demander d'effectuer certaines opérations ou des tests supplémentaires. Cette possibilité est importante puisque plus le compilateur fait de vérifications, **plus il peut nous signaler d'éventuels problèmes** dans notre code. C'est ce qu'on appelle les *warnings*. Loin de nous embêter, le compilateur est donc un précieux allié.

IV.4.2.5.1. Un mot sur les outils

Comme le compilateur fourni avec Visual Studio est différent de GCC ou de Clang, les options de compilation et les *warnings* sont aussi différents et ne s'activent pas forcément de la même façon.

Avec Visual Studio, allez dans les propriétés du projet, puis `C/C++` + `-> Général` et là, mettez le champ `Niveau d'avertissement` à la valeur `Niveau 4 (/W4)`. Cela augmente le nombre de vérifications effectuées et donc de *warnings* générés. Je ne conseille pas `(/Wall)` parce que cette option est si sévère qu'elle va même générer des avertissements dans le code de la bibliothèque standard.

Avec GCC ou Clang, il faut **rajouter des options** supplémentaires, qu'on appelle couramment des *flags*. Il en existe énormément, tant pour [GCC](#) que pour [Clang](#). Mais les deux plus utilisés sont `-Wall` et `-Wextra`, qui activent en sous-main de nombreux *warnings*, ceux revenant le plus fréquemment.

```
1 > g++ -std=c++17 -Wall -Wextra test.cpp -o test.out
2 > clang++ -std=c++17 -Wall -Wextra test.cpp -o test.out
```



Un mot sur les *warnings*

Je vous conseille de toujours activer les *warnings* suggérés ci-dessus. Plus le compilateur travaille pour vous, plus vous pouvez corriger tôt d'éventuels problèmes et améliorer ainsi la qualité de votre code.

IV.4.2.5.2. Conversion entre types

Dès le début de ce cours, nous avons vu des types simples, permettant de stocker des caractères (`char`), des entiers (`int`, `std::size_t`) et des réels (`double`). Ils sont bien pratiques, mais si on commence à les mélanger, on peut avoir des problèmes.

En effet, comme déjà signalé dans le chapitre sur les erreurs, tous n'ont pas la même taille. Un `int`, par exemple, ne peut pas stocker des nombres aussi grands que `double` peut. Ainsi, convertir un `double` en `int` entraîne **une perte d'information**. C'est un peu comme vouloir transvaser de l'eau depuis un seau dans un gobelet.

Le code suivant illustre bien le principe, puisque en passant d'un `double` à un `int`, on perd comme information la partie décimale du nombre.

```
1 #include <iostream>
2
3 double fonction()
4 {
5     return 3.141582;
6 }
7
8 int main()
9 {
10     int const x { fonction() };
11     std::cout << "Voici un nombre : " << x << std::endl;
12
13     return 0;
14 }
```

Le compilateur ne va d'ailleurs pas se gêner pour vous le dire.

```
1 [Visual Studio]
2 Erreur C2397 la conversion de 'double' en 'int' requiert une
   conversion restrictive
3
4 -----
5
6 [GCC]
7 test.cpp: In function 'int main()':
8 test.cpp:10:27: warning: narrowing conversion of 'fonction()' from
   'double' to 'int' inside { } [-Wnarrowing]
9     int const x { fonction() };
10                  ~~~~~^~
11 -----
12
13 [Clang]
```

```
14 test.cpp:10:19: error: type 'double' cannot be narrowed to 'int' in
    initializer list [-Wc++11-narrowing]
15     int const x { fonction() };
16                 ^~~~~~
17 test.cpp:10:19: note: insert an explicit cast to silence this issue
18     int const x { fonction() };
19                 ^~~~~~
20                 static_cast<int>( )
```

Le compilateur nous prévient que c'est une erreur et qu'il faut une **conversion explicite**, ou en anglais *cast*, pour résoudre le problème. Clang est le plus sympa puisqu'il propose une solution, qui consiste à utiliser le mot-clé `static_cast`. Ce dernier permet de dire explicitement au compilateur qu'on veut convertir un type vers un autre, en l'occurrence ici un `double` vers un `int`. Il s'utilise comme suit.

```
1 static_cast<le type qu'on veut en résultat>(une variable ou une
   littérale)
```

Une fois la conversion explicite appliquée, le code fonctionne sans problème.

```
1 #include <iostream>
2
3 double fonction()
4 {
5     return 3.141582;
6 }
7
8 int main()
9 {
10     int const x { static_cast<int>(fonction()) };
11     std::cout << "Voici un nombre : " << x << std::endl;
12
13     return 0;
14 }
```

Sans les avertissements du compilateur, l'erreur serait sans doute passée inaperçue. Si l'on comptait faire des calculs avec nos valeurs réelles, ceux-ci auraient été faussés puisque les parties décimales auraient disparues. Avec l'avertissement, nous pouvons soit corriger notre erreur, soit mettre le *cast* pour montrer que notre conversion est voulue et volontaire.

IV.4.2.5.3. Oubli d'initialiser

Imaginez que vous oubliez d'écrire les accolades `{}` après avoir déclaré une variable. C'est moyennement cool, puisque du coup, **on ne lui donne pas de valeur par défaut**. Elle peut

IV. Interlude - Être un développeur

donc valoir tout et n'importe quoi. Le message du code suivant peut, ou non, s'afficher. C'est impossible à savoir.

```
1 #include <iostream>
2
3 int main()
4 {
5     int x;
6
7     if (x > 5)
8     {
9         std::cout << "J'ai l'impression d'oublier quelque chose..."
10            << std::endl;
11     }
12
13     return 0;
14 }
```

Heureusement, le compilateur ne manque pas de le souligner.

```
1 [Visual Studio]
2 Erreur C4700 variable locale 'x' non initialisée utilisée
3
4 -----
5
6 [GCC]
7 test.cpp: In function 'int main()':
8 test.cpp:7:5: warning: 'x' is used uninitialized in this function
9     [-Wuninitialized]
10     if (x > 5)
11         ^~
12
13 -----
14 [Clang]
15 test.cpp:7:9: warning: variable 'x' is uninitialized when used here
16     [-Wuninitialized]
17     if (x > 5)
18         ^
19
20 test.cpp:5:10: note: initialize the variable 'x' to silence this
21     warning
22     int x;
23         ^
24     = 0
```

IV.4.2.5.4. Paramètres inutilisés

Imaginez que vous déclarez une fonction attendant deux paramètres, mais que vous n'en utilisez qu'un seul. C'est peut-être volontaire, parce que vous n'avez pas encore fini de l'implémenter, mais c'est peut-être une erreur, soit que vous avez oublié une portion de code à écrire, soit oublié de supprimer un paramètre désormais inutile.

```
1 #include <iostream>
2 #include <string>
3
4 void fonction(std::string const & chaine, bool en_majuscule)
5 {
6     std::cout << chaine << std::endl;
7 }
8
9 int main()
10 {
11     std::string const une_phrase { "Du texte." };
12     fonction(une_phrase, true);
13
14     return 0;
15 }
```

Votre compilateur se demande quelle est votre intention et vous affiche donc un message.

```
1 [Visual Studio]
2 Warning C4100: 'en_majuscule' : paramètre formel non référencé
3
4 -----
5
6 [GCC]
7 test.cpp: In function 'void fonction(const string&, bool)':
8 test.cpp:4:48: warning: unused parameter 'en_majuscule'
9     [-Wunused-parameter]
10 void fonction(std::string const & chaine, bool en_majuscule)
11                                             ^~~~~~
12 -----
13 [Clang]
14 test.cpp:4:48: warning: unused parameter 'en_majuscule'
15     [-Wunused-parameter]
16 void fonction(std::string const & chaine, bool en_majuscule)
17                                             ^
17 1 warning generated.
```

IV.4.2.5.5. Release et debug

Les options de compilation servent aussi à indiquer au compilateur le mode choisi. Il est ainsi capable d'optimiser le programme pour le rendre plus petit et plus rapide. Une partie de ce travail est faite quand le préprocesseur passe, mais d'autres améliorations et optimisations sont effectuées par le compilateur.

Avec Visual Studio, il faut simplement changer la valeur du cadre de l'image ci-dessous, qui se trouve près du triangle vert **Débogueur Windows local**.

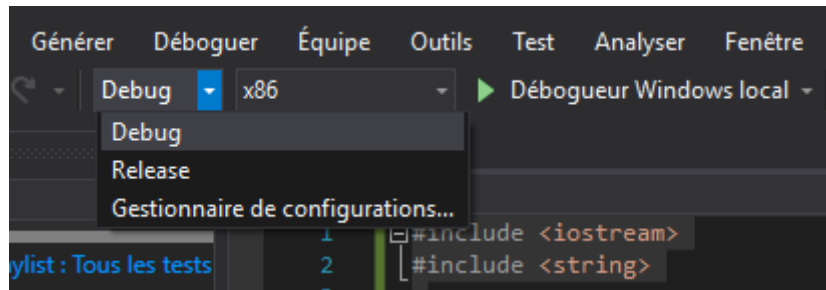


FIGURE IV.4.2. – Choix du mode de compilation avec Visual Studio 2017.

Avec GCC et Clang, c'est l'**option de compilation -O2** qu'il faut passer. Celle-ci cherche à optimiser le programme pour rendre son exécution plus rapide.

```
1 g++ -std=c++17 -Wall -Wextra -O2 test.cpp -o test.out
```

IV.4.3. Le linker



Un mot sur les outils

Les propos ci-dessous sont illustrés sous GNU/Linux, mais le principe et la plupart des explications sont valables peu importe le système d'exploitation.

Maintenant que nous avons des fichiers objets, il faut **les lier** pour obtenir notre exécutable final. C'est justement le travail accompli pendant l'étape du *linkage*. Et pour illustrer mes explications, nous allons travailler avec le code ci-dessous, divisé en trois fichiers.

D'abord, le fichier d'en-tête **test.hpp**, qui ne contient qu'un simple prototype.

```
1 #ifndef TEST_HPP
2 #define TEST_HPP
3
4 int fonction();
5
```

IV. Interlude - Être un développeur

```
6 #endif
```

Ensuite, le fichier `test.cpp`, qui est l'implémentation du prototype déclaré précédemment.

```
1 #include "test.hpp"
2
3 int fonction()
4 {
5     return 38;
6 }
```

Et enfin le fichier `main.cpp`, le point d'entrée.

```
1 #include <iostream>
2 #include "test.hpp"
3
4 int main()
5 {
6     int const x { 5 };
7     std::cout << "Voici un calcul : " << x + fonction() <<
8         std::endl;
9     return 0;
10 }
```

On génère donc le code. Que ce soit avec GCC, Clang ou en utilisant Visual Studio, on obtient deux fichiers objets, un par fichier source.

```
1 > g++ -std=c++17 -c test.cpp main.cpp
2 > clang++ -std=c++17 -c test.cpp main.cpp
```

IV.4.3.1. Une question de symboles

Un fichier objet contient, entre autres, des **symboles**. Qu'est-ce que c'est? Un symbole correspond **au nom d'une fonction ou d'une variable**. Les noms de toutes les fonctions et variables utilisées dans chaque fichier source sont donc présents dans les fichiers objets correspondants. On peut le voir en utilisant la commande `objdump -t fichier.o -C`.

```

1 > objdump -t test.o -C
2
3 test.o:      format de fichier elf64-x86-64
4
5 SYMBOL TABLE:
6 0000000000000000 l      df *ABS*  0000000000000000 test.cpp
7 0000000000000000 l      d  .text  0000000000000000 .text
8 0000000000000000 l      d  .data  0000000000000000 .data
9 0000000000000000 l      d  .bss   0000000000000000 .bss
10 0000000000000000 l      d  .note.GNU-stack      0000000000000000
    .note.GNU-stack
11 0000000000000000 l      d  .eh_frame      0000000000000000 .eh_frame
12 0000000000000000 l      d  .comment      0000000000000000 .comment
13 0000000000000000 g      F  .text  000000000000000b fonction()

```

De nombreuses informations sont affichées, qui nécessiteraient beaucoup d'explication. Notons simplement qu'on retrouve notre identificateur `fonction` sur la dernière ligne. La lettre **F** indique qu'il s'agit d'une fonction. Devant, on note la présence de `.text`, la **section** dans laquelle `fonction` se trouve.

Sans rentrer dans les détails ni transformer ce cours de C++ en cours système, il faut savoir qu'un programme est découpé en plusieurs sections. **Une section regroupe certaines informations ensemble.** Citons la section `.text` qui contient le code et `.rodata`, qui contient toutes les données non-modifiables, comme les littérales chaînes de caractères.

Dans le cas du fichier objet `test.o`, comme `fonction` désigne une fonction, donc du code exécutable, il est normal que sa section soit `.text`. Et si on faisait la même analyse sur `main.o`?

```

1 objdump -t main.o -C
2
3 main.o:      format de fichier elf64-x86-64
4
5 SYMBOL TABLE:
6 0000000000000000 l      df *ABS*  0000000000000000 main.cpp
7 0000000000000000 l      d  .text  0000000000000000 .text
8 0000000000000000 l      d  .data  0000000000000000 .data
9 0000000000000000 l      d  .bss   0000000000000000 .bss
10 0000000000000000 l      0 .bss   0000000000000001 std::__ioinit
11 0000000000000000 l      d  .rodata      0000000000000000 .rodata
12 0000000000000059 l      F  .text  0000000000000049
    __static_initialization_and_destruction_0(int, int)
13 00000000000000a2 l      F  .text  0000000000000015
    _GLOBAL__sub_I_main
14 0000000000000000 l      d  .init_array      0000000000000000
    .init_array

```


15	000000000000000000000000	l	d	.note.GNU-stack	000000000000000000000000
				.note.GNU-stack	
16	000000000000000000000000	l	d	.eh_frame	000000000000000000000000 .eh_frame
17	000000000000000000000000	l	d	.comment	000000000000000000000000 .comment
18	000000000000000000000000	g	F	.text	00000000000000000059 main
19	000000000000000000000000		*UND*	000000000000000000000000	std::cout
20	000000000000000000000000		*UND*	000000000000000000000000	
				_GLOBAL_OFFSET_TABLE_	
21	000000000000000000000000		*UND*	000000000000000000000000	
				std::basic_ostream<char, std::char_traits<char> >&	
				std::operator<< <std::char_traits<char>	
				>(std::basic_ostream<char, std::char_traits<char> >&, char	
				const*)	
22	000000000000000000000000		*UND*	000000000000000000000000	fonction()
23	000000000000000000000000		*UND*	000000000000000000000000	
				std::ostream::operator<<(int)	
24	000000000000000000000000		*UND*	000000000000000000000000	
				std::basic_ostream<char, std::char_traits<char> >&	
				std::endl<char, std::char_traits<char>	
				>(std::basic_ostream<char, std::char_traits<char> >&)	
25	000000000000000000000000		*UND*	000000000000000000000000	
				std::ostream::operator<<(std::ostream& (*)(std::ostream&))	
26	000000000000000000000000		*UND*	000000000000000000000000	
				std::ios_base::Init::Init()	
27	000000000000000000000000		*UND*	000000000000000000000000	.hidden
				__dso_handle	
28	000000000000000000000000		*UND*	000000000000000000000000	
				std::ios_base::Init::~Init()	
29	000000000000000000000000		*UND*	000000000000000000000000	__cxa_atexit

Certaines choses ne nous surprennent pas, comme `main` désignée comme étant une fonction située dans `.text`. Par contre, d'autres lignes n'ont que pour seule information `*UND*`. On trouve plusieurs choses familières, comme `std::cout` ou encore `std::ostream::operator<<(int)`.

En fait, toutes les lignes contenant `*UND*` désignent des symboles n'ayant **aucune section liée**. Ce sont des symboles qui ne sont pas définis dans notre fichier objet. Il faut donc **lier celui-ci à d'autres fichiers objets**, notamment ceux qui contiennent ce qui nous manque.

Dans le cas de la bibliothèque standard, c'est fait automatiquement par le compilateur sans avoir besoin de rien faire. Mais pour résoudre le symbole `fonction()`, il faut **lier nos deux fichiers objets**. C'est très simple, la commande suivante vous le montre. Notez que l'option `-std=c++17` est absente, car ici on ne compile rien, **on *linke*, ou assemble, deux fichiers objets**.

```

1 g++ test.o main.o -o mon_super_programme.out
2 clang++ test.o main.o -o mon_super_programme.out

```

Maintenant, on obtient un exécutable parfaitement fonctionnel. La compilation est terminée.

IV. Interlude - Être un développeur

```
1 ./super_programme.out
2 Voici un calcul : 43
```

Au contraire, si on tente de compiler seulement `main.o`, le *linkage* échouera avec une erreur comme celle ci-dessous.

```
1 > g++ main.o -o super_programme.out
2 main.o : Dans la fonction « main » :
3 main.cpp:(.text+0x27) : référence indéfinie vers « fonction() »
4 collect2: error: ld returned 1 exit status
5
6 > clang++ main.o -o super_programme.out
7 main.o : Dans la fonction « main » :
8 main.cpp:(.text+0x27) : référence indéfinie vers « fonction() »
9 clang: error: linker command failed with exit code 1 (use -v to see
   invocation)
```

IV.4.4. Schéma récapitulatif

Le processus de compilation d'un programme C++ peut sembler vraiment complexe quand on démarre. C'est vrai qu'il s'agit d'un domaine avancé, mais avoir une idée globale du processus et de ce qui se passe sous le capot est important. Voici donc un schéma, car une image vaut mieux que mille mots. 🍊

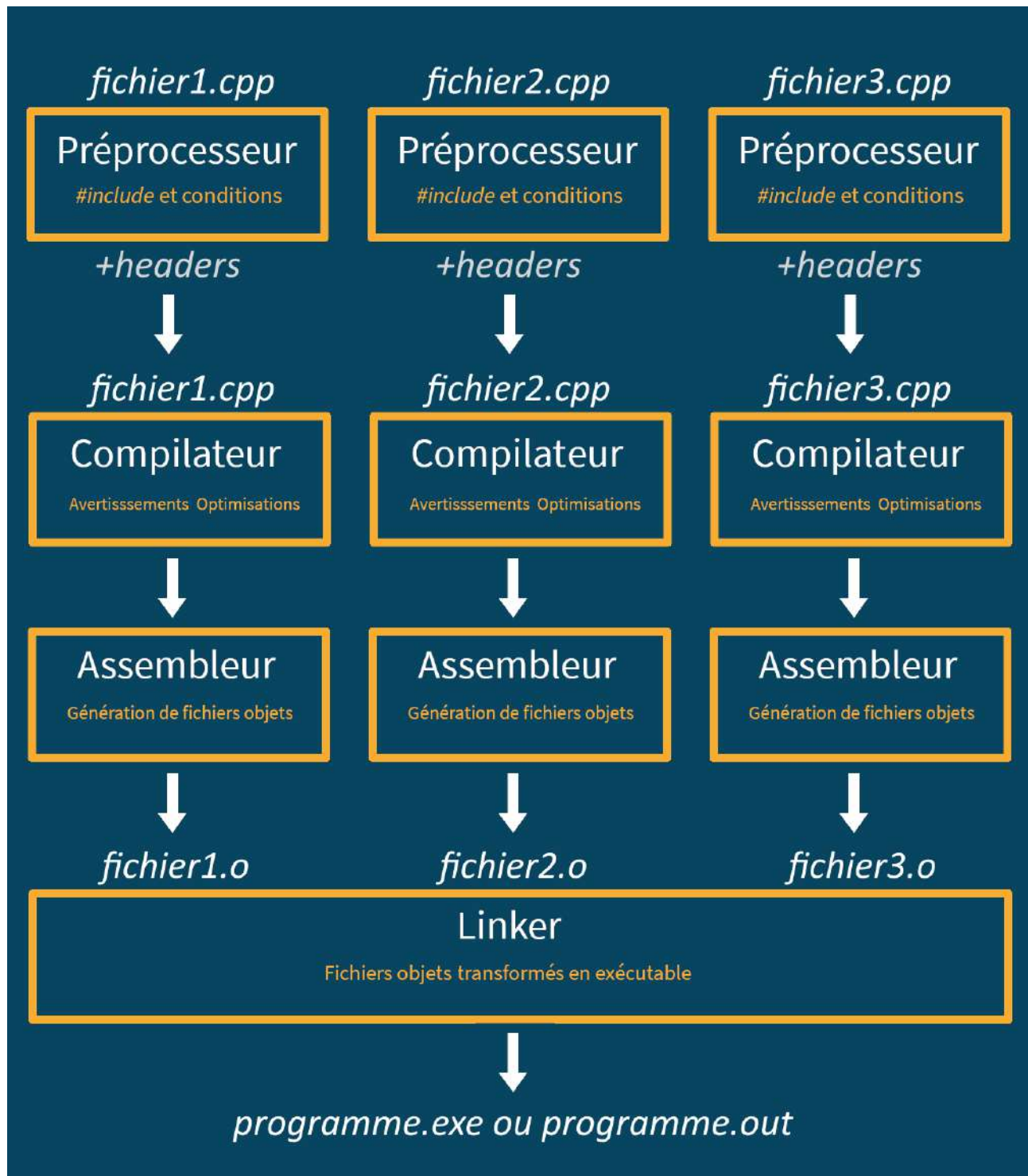


FIGURE IV.4.3. – Processus de compilation

IV.4.4.1. En résumé

- Le préprocesseur commence le travail en remplaçant chaque directive `#include` par le contenu du fichier demandé et en exécutant des tests conditionnels pour activer ou désactiver certains morceaux de code.

IV. Interlude - Être un développeur

- Le compilateur instancie les *templates*, exécute les fonctions `constexpr`, vérifie la qualité de notre code puis, en plusieurs phases, transforme le code C++ en fichier objet.
- Le *linker* fait le lien entre plusieurs fichiers objets pour résoudre des symboles manquants et achever le processus de compilation en générant un exécutable fonctionnel.
- Il n'est pas obligatoire de connaître chaque détail, mais avoir une vision globale est important.

IV.5. Chasse aux bugs !

Malgré tout le soin qu'on peut apporter à la conception, malgré tous les *warnings* qu'on active pendant la compilation, même avec toutes les bonnes pratiques du monde, nous faisons des erreurs et nous créons des *bugs*. Il nous faut donc des outils supplémentaires pour les traquer et les éliminer.

Ce chapitre va nous faire découvrir l'usage des [débugueurs](#) .

IV.5.1. Le principe

À l'exécution d'un programme, il se passe beaucoup de choses et très vite. Une fonction en appelle une autre, une variable est modifiée, une exception lancée, etc. Il est donc difficile, d'un simple coup d'œil, de savoir où est survenu le problème. Le principe du débogueur est donc de **dérouler l'exécution du programme pas-à-pas**, ce qui permet, à chaque instant, de voir où on en est et quel est l'état de nos variables et de la mémoire.

Pour ordonner au débogueur de s'arrêter à un endroit précis dans le code, on pose ce qu'on appelle un **point d'arrêt**, ou, en anglais, *breakpoint*. Chaque fois que le débogueur passe par un point d'arrêt, **le programme débogué se stoppe jusqu'à nouvel ordre** et toutes les informations courantes sont affichées, comme la liste des fonctions appelées jusque-là (la *stack trace* en anglais), la valeur de toutes les variables de la portée courante, etc.

Pour les débogueurs les plus avancés, il est également possible de modifier des données du programme, de poursuivre l'exécution à un autre endroit du code pour sauter un crash ou une exception, observer le code assembleur généré et même de déboguer des programmes à distance, par Internet notamment.

Pour faire tout ce travail, le débogueur se repose sur les **symboles de débogage**, qui sont présents quand on compile en *debug* mais souvent éliminés quand on passe en *release*. Donc, afin de pouvoir utiliser cet outil puissant, veillez à passer en mode **Debug** si vous utilisez Visual Studio, ou bien à compiler votre programme avec l'option `-g`, pour charger ces fameux symboles.

```
1 > g++ -std=c++17 -g test.cpp -o test.out
2 > clang++ -std=c++17 -g test.cpp -o test.out
```

IV.5.2. Un code d'exemple

Afin d'illustrer ce chapitre, nous allons utiliser le morceau de code suivant, qui est certes basique, mais qui va nous permettre de découvrir le principe du débogage en douceur.

```
1  #include <iostream>
2  #include <vector>
3
4  void fonction()
5  {
6      std::vector<int> tableau {};
7
8      tableau.push_back(10);
9      tableau.push_back(20);
10
11     int & a { tableau.front() };
12     int & b { tableau.back() };
13
14     std::cout << "Front : " << tableau.front() << std::endl;
15     std::cout << "Back : " << tableau.back() << std::endl;
16
17     b = 5;
18
19     std::cout << "Front : " << tableau.front() << std::endl;
20     std::cout << "Back : " << tableau.back() << std::endl;
21 }
22
23 int main()
24 {
25     fonction();
26     return 0;
27 }
```

IV.5.3. Visual Studio

Le débogueur inclus avec Visual Studio se lance très simplement en faisant **F5**, ou en cliquant sur l'option **Débogueur Windows local**. Mais si vous lancez le code tel quel en *debug*, la console va s'ouvrir et se refermer aussitôt. Il faut donc commencer par **poser des points d'arrêt**. Pour ça, c'est très simple, on clique sur la colonne à gauche de celle indiquant les numéros de ligne. Comme une image est plus claire, voici un point d'arrêt posé.

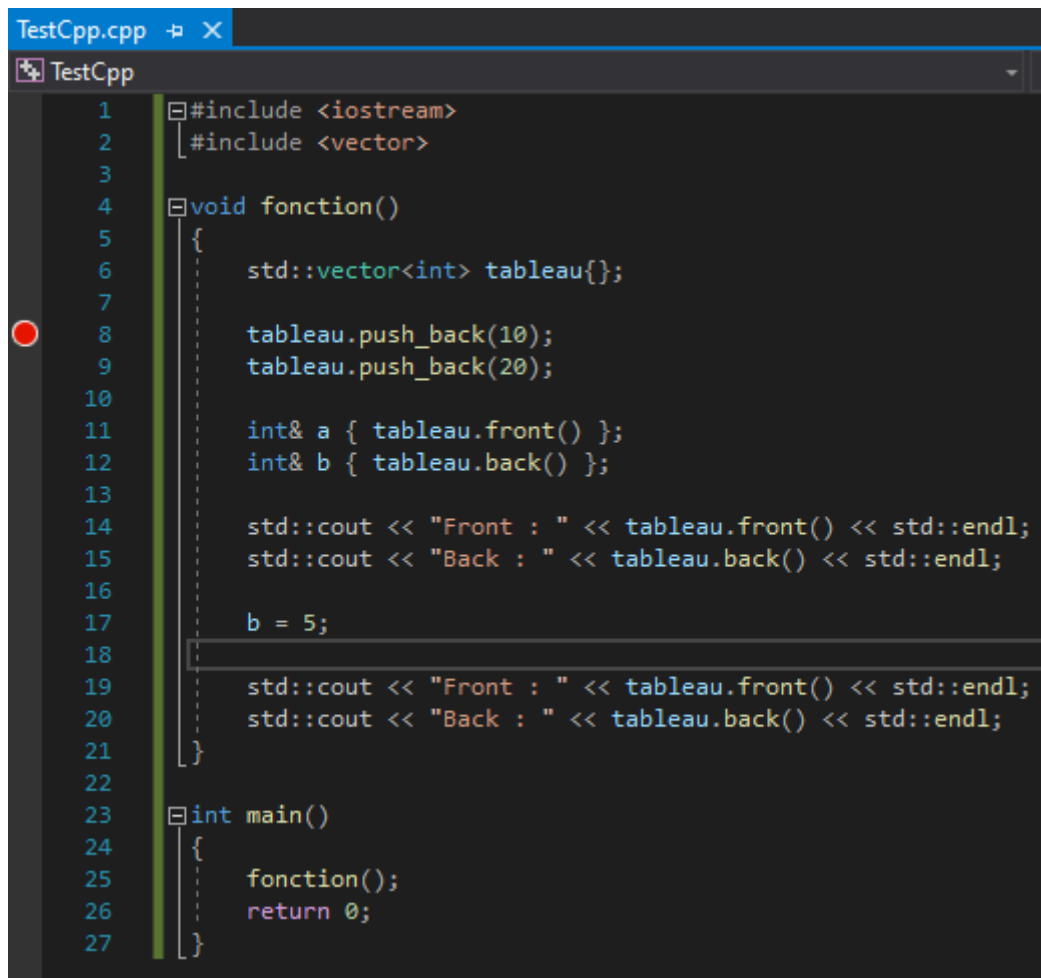


FIGURE IV.5.1. – Nous avons posé un point d'arrêt à la ligne 8.

IV.5.3.1. Interface

Une fois que le programme est lancé, l'aspect de Visual Studio change un peu. Examinons ce qu'il y a de nouveau, en commençant par la petite flèche jaune, actuellement sur la ligne 8, qui indique l'instruction à laquelle on se trouve.

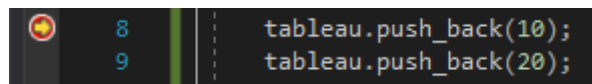


FIGURE IV.5.2. – Le curseur de debug se trouve actuellement à la ligne 8.

En bas à gauche, c'est la fenêtre des variables, grâce à laquelle on peut voir, à chaque étape, l'état des variables du programme.

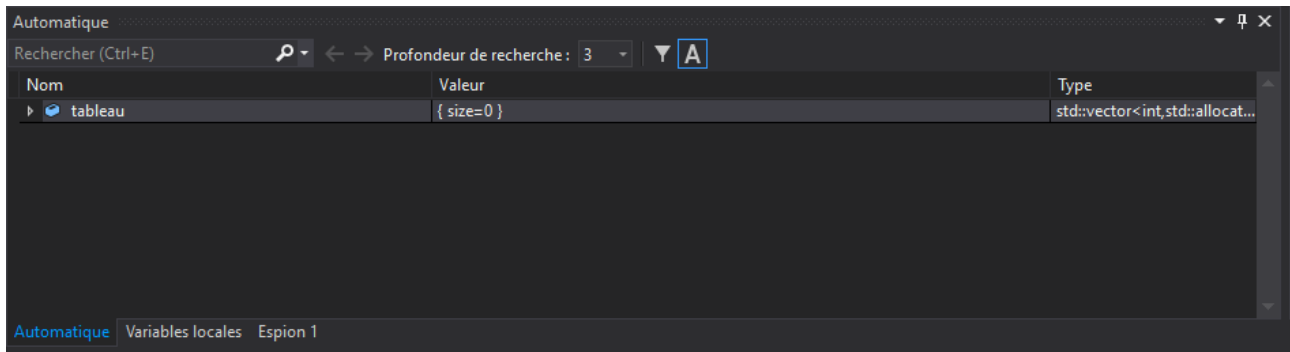


FIGURE IV.5.3. – Fenêtre des variables avec l’affichage de la variable `tableau`.

Elle est divisée en trois onglets.

- **Automatique** correspond à **toutes les variables existantes jusqu’à la ligne où l’on se trouve**. Celles situées après ne sont pas encore connues donc pas encore affichées. Pour l’instant, il n’y a que `tableau`.
- **Variables locales** correspond à **toutes les variables de la portée actuelle**, existant déjà en mémoire ou non. Dans notre exemple, on retrouve donc `tableau`, ainsi que `a` et `b`, même si celles-ci n’ont pas encore de valeur mais juste une ligne «Impossible de lire la mémoire.»
- **Espion 1** permet de **garder un œil sur une variable** et d’être au courant de toutes ses modifications. On peut, par exemple, garder une trace de la taille du tableau en créant un espion `tableau.size()`.

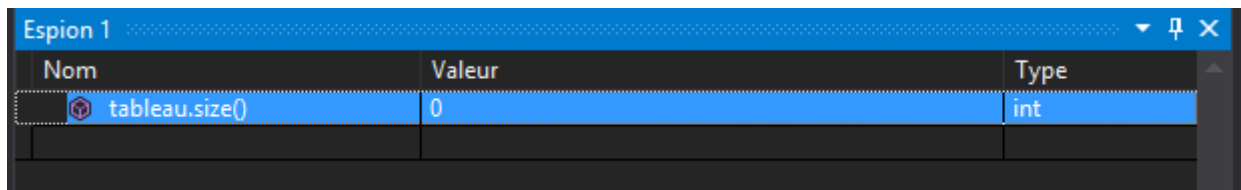


FIGURE IV.5.4. – Exemple d’espion. Quand le tableau sera agrandi ou rétréci, la valeur de l’espion sera automatiquement modifiée.

En bas à droite, une autre fenêtre permet de voir, entre autres, **la pile des appels**, c’est-à-dire quelle fonction appelle quelle autre fonction. Ici, il n’y en a que deux, donc ce n’est pas étonnant de voir `main` appeler `fonction`. Mais quand le code se complexifie, il est très utile de voir par quelle fonction on passe et laquelle déclenche le *bug*.

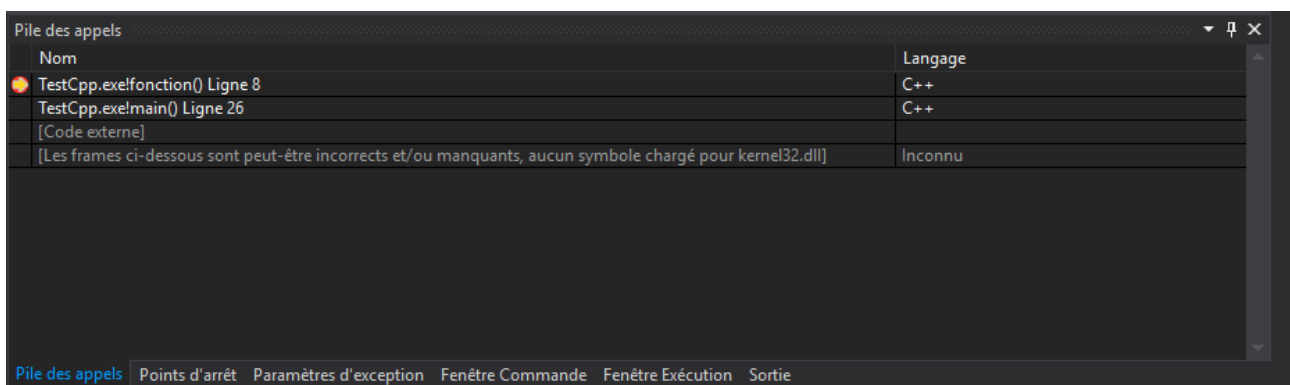


FIGURE IV.5.5. – Fenêtre de la pile des appels.

IV.5.3.2. Pas-à-pas

Que diriez-vous maintenant de passer à l'instruction suivante? Pour cela, on dispose de plusieurs choix possibles, selon ce qu'on décide de faire.

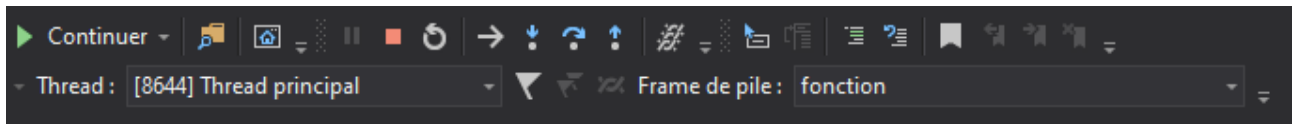


FIGURE IV.5.6. – Interface avec les différentes actions possibles.

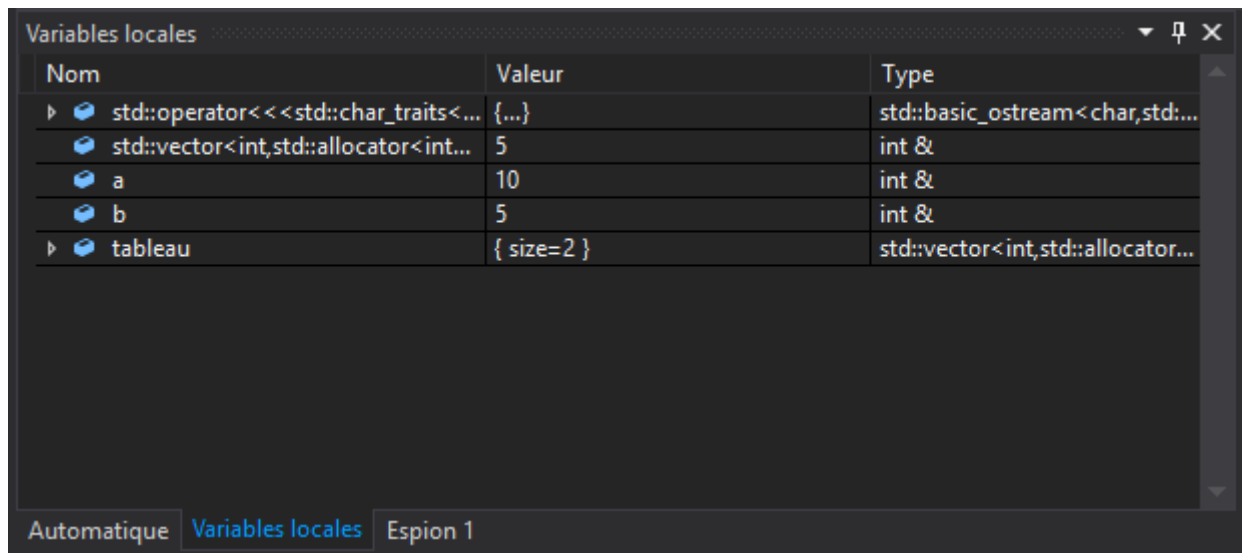
- Le bouton **Continuer** permet d'exécuter le code **jusqu'au prochain point d'arrêt**. Comme ici il n'y en a qu'un seul, cela signifie jusqu'à la fin du programme.
- Le carré rouge **arrête le débogage**. Avec le clavier, on fait **Maj + F5**.
- La flèche cerclée **redémarre le débogage**. Avec le clavier, on fait **Ctrl + Maj + F5**.
- Le **pas-à-pas principal** permet de **passer à l'instruction suivante**, mais sans aller plus loin dans la pile d'appel. Cela signifie que si l'instruction à exécuter est un appel de fonction, le pas-à-pas principal n'ira pas voir dedans. Avec le clavier, c'est **F10**.
- Le **pas-à-pas détaillé** permet d'examiner **en détails** les instructions. Ainsi, si vous passez sur une fonction, le pas-à-pas détaillé ira dans cette fonction examiner ses instructions. Avec le clavier, c'est **F11**.

En cliquant sur le pas-à-pas principal, nous voyons la flèche jaune se déplacer d'une ligne et passer à l'instruction suivante. Dans la fenêtre des variables locales, `tableau` s'est mis à jour et sa taille est passée à 1, tout comme l'espion `tableau.size()`. On a donc bien exécuté une instruction `push_back` sur le tableau.

Espion 1		
Nom	Valeur	Type
<code>tableau.size()</code>	1	int

FIGURE IV.5.7. – Variable espionne bien mise à jour après exécution de la première instruction, qui ajoute un élément au tableau.

L'image ci-dessous vous montre l'état des variables quand je fais un pas-à-pas principal jusqu'à la fin de la fonction. On voit bien que `a` et `b` existent maintenant. On en note également deux autres, dues à la manipulation du flux de sortie, qu'on peut ignorer parce qu'elles ne nous intéressent pas.



Nom	Valeur	Type
std::operator<<<std::char_traits<... {...}	{...}	std::basic_ostream<char,std::...
std::vector<int,std::allocator<int...	5	int &
a	10	int &
b	5	int &
tableau	{ size=2 }	std::vector<int,std::allocator...

Automatique Variables locales Espion 1

FIGURE IV.5.8. – Variables après le pas-à-pas principal jusqu'à la fin de la fonction.

IV.5.3.3. Point d'arrêt conditionnel

Faire un pas-à-pas classique, c'est bien, mais quand on doit dérouler une boucle de 100 éléments et qu'on veut la valeur à la 95ème itération, c'est embêtant. Sauf qu'on dispose d'un moyen pratique qui consiste à dire au débogueur de **s'arrêter quand une condition particulière est vérifiée**.

```

1  #include <iostream>
2  #include <vector>
3
4  void fonction()
5  {
6      for (int i { 0 }; i < 100; ++i)
7      {
8          std::cout << i * 2 << std::endl;
9      }
10 }
11
12 int main()
13 {
14     fonction();
15     return 0;
16 }
```

Prenons un code simple comme celui ci-dessus et posons un point d'arrêt sur la ligne 7. En faisant un clic-droit sur le point d'arrêt, puis cliquez sur **Conditions...**. Vous allez voir un cadre apparaître, dans lequel on peut décider d'une condition à respecter pour que le débogueur s'arrête sur le point d'arrêt désigné.

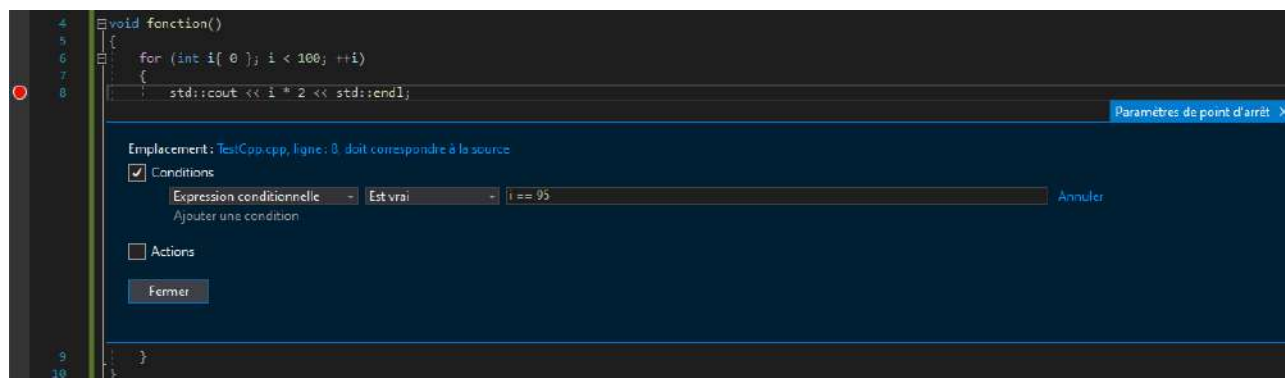


FIGURE IV.5.9. – Point d’arrêt conditionnel, uniquement quand `i` vaudra 95.

Si on lance le débogage, on voit bien que le code s’exécute jusqu’au moment où `i` vaut 95. Là, le programme est stoppé et le débogueur attend nos instructions. Pour preuve, regardez les variables locales.


Variables locales		
Nom	Valeur	Type
 <code>i</code>	95	int

FIGURE IV.5.10. – Notre variable locale `i` vaut bien 95.

IV.5.3.3.1. Aller plus loin

Tout présenter m’est impossible, surtout pour un outil qui mériterait un cours à lui tout seul. Je vous encourage à **jeter un regard sur les tutoriels** [de la documentation officielle Microsoft](#). Apprendre à bien maîtriser le débogueur de Visual Studio ne vous servira pas qu’en C++, mais également en C# ou en Python, si vous continuez à utiliser cet environnement pour développer avec ces langages.

IV.5.4. Qt Creator

Le principe du débogage est assez semblable avec Qt Creator. Cet environnement se base sur l’outil libre [gdb](#), qui s’utilise normalement en ligne de commande.

Si vous lancez le code tel quel en *debug*, la console va s’ouvrir et se refermer aussitôt. Il faut donc commencer par poser des points d’arrêt, ce qui se fait en posant un point d’arrêt en cliquant sur la colonne à gauche de celle des numéros de lignes. On lance ensuite le débogage avec **F5**.

IV.5.4.1. Interface

Commençons par examiner l’interface de débogage, pour le même code que ci-dessus. On trouve une petite flèche jaune qui indique **la ligne sur laquelle on se trouve**.

IV. Interlude - Être un développeur

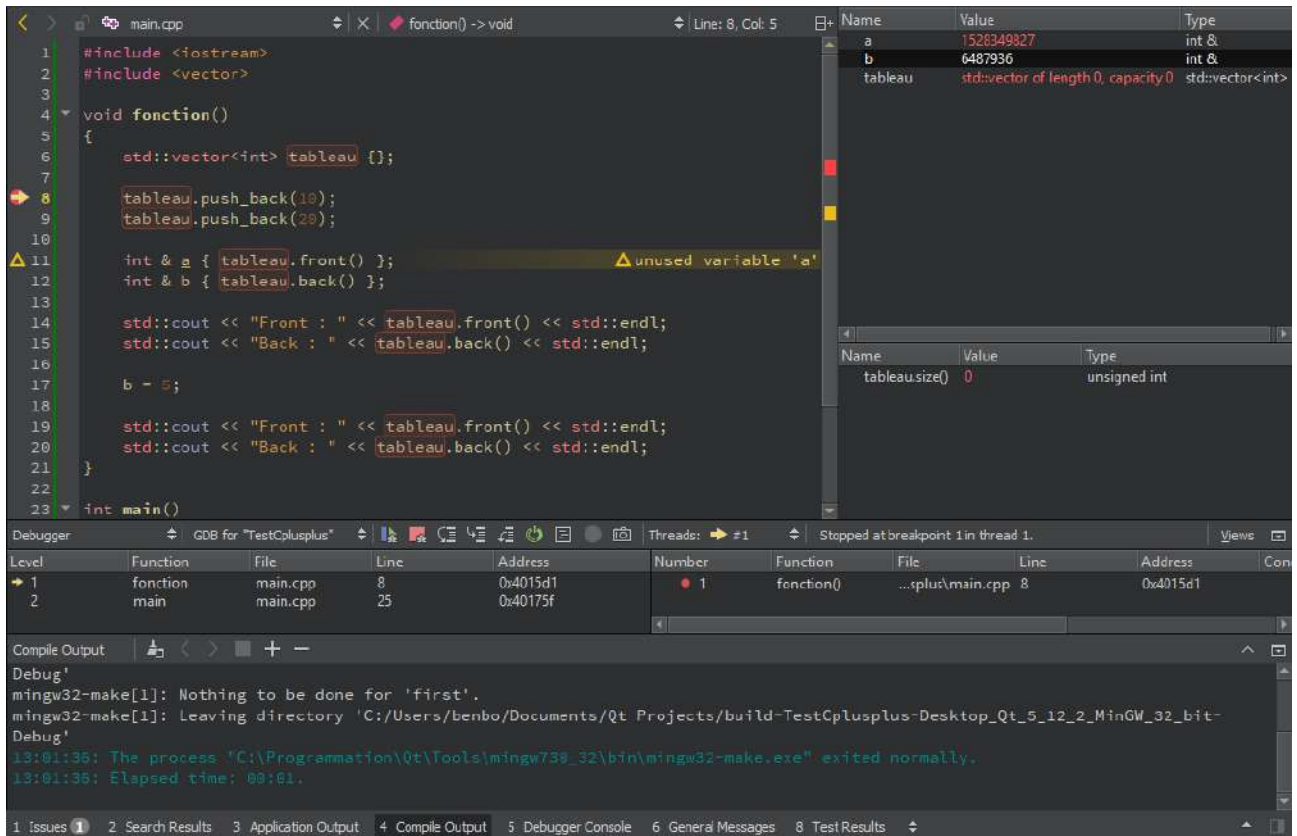


FIGURE IV.5.11. – Vue de Qt Creator pendant un débogage.

En haut à droite, on a d'abord **la liste des variables locales et les paramètres**. On retrouve bien `tableau` qui est vide; `a` et `b` contiennent des valeurs aléatoires, car elles sont encore non-initialisées.

Name	Value	Type
a	1528349827	int &
b	6487936	int &
tableau	std::vector of length 0, capacity 0	std::vector<int>

FIGURE IV.5.12. – Les variables locales.

Juste en dessous se trouve la fenêtre des **expressions**. On peut ici écrire n'importe quelle expression C++ valide, ce qui permet de toujours garder un œil sur la valeur d'une variable en particulier, comme ici la taille du tableau.

Name	Value	Type
tableau.size()	0	unsigned int

FIGURE IV.5.13. – Une expression évaluant la taille du tableau, automatiquement mise à jour quand le tableau est modifié.

Enfin, plus bas, en dessous du code, on trouve **la pile des appels**, c'est-à-dire quelle fonction appelle quelle autre fonction. Également, on trouve la liste de tous les points d'arrêt, ce qui est plutôt utile quand on en a beaucoup.

Debugger					Threads: #1 Stopped at breakpoint 1 in thread 1.					Views
Level	Function	File	Line	Address	Number	Function	File	Line	Address	Con
1	fonction	main.cpp	8	0x4015d1	1	fonction()	...splus\main.cpp	8	0x4015d1	
2	main	main.cpp	25	0x40175f						






FIGURE IV.5.14. – Liste des appels et des points d'arrêt.

IV.5.4.2. Pas-à-pas

Maintenant, il serait de bon ton de continuer le débogage de notre application. Faisons donc du pas-à-pas.



FIGURE IV.5.15. – Les différentes options de pas-à-pas.

- Le bouton Continuer  permet de **continuer l'exécution jusqu'au prochain point d'arrêt**. Avec le clavier, on fait **F5**.
- Le carré rouge  **arrête le débogage**.
- Le bouton  permet le **pas-à-pas principal**, c'est à dire se rendre à la prochaine instruction sans explorer la pile d'appel. Cela signifie que si l'instruction à exécuter est un appel de fonction, le pas-à-pas principal n'ira pas voir dedans. Avec le clavier, c'est **F10**.
- Le bouton  permet le **pas-à-pas détaillé**, c'est à dire se rendre à la prochaine instruction en explorant la pile d'appel. Ainsi, si vous passez sur une fonction, le pas-à-pas détaillé ira dans cette fonction examiner ses instructions. Avec le clavier, c'est **F11**.
- Le bouton  **redémarre la session de débogage**.

IV. Interlude - Être un développeur

En cliquant sur le pas-à-pas principal, on voit bien la flèche jaune se déplacer à l'instruction suivante, ajoutant ainsi une valeur au tableau, ce qui met également l'expression évaluée `tableau.size()` à jour.

Name	Value	Type
<code>tableau.size()</code>	1	unsigned int

FIGURE IV.5.16. – L'expression a bien été mise à jour.

Si on poursuit le pas-à-pas principal jusqu'à la fin de la fonction, on voit que toutes nos variables sont mises à jour et ont maintenant des valeurs cohérentes.

Name	Value	Type
<code>a</code>	10	int &
<code>b</code>	5	int &
▶ <code>tableau</code>	std::vector of length 2, capacity 2	std::vector<int>

FIGURE IV.5.17. – Les valeurs quand le pas-à-pas atteint la fin de la fonction.

IV.5.4.3. Point d'arrêt conditionnel

Avec Qt Creator, on peut tout à fait **poser une condition sur un point d'arrêt**, si l'on veut qu'il s'exécute à une condition précise. Dans le cas d'une boucle, on peut ainsi s'épargner de faire du pas-à-pas et aller directement à une valeur qui nous intéresse. Reprenons le code de la partie précédente et faisons le test.

```
1 #include <iostream>
2 #include <vector>
3
4 void fonction()
5 {
```

```
6      for (int i { 0 }; i < 100; ++i)
7      {
8          std::cout << i * 2 << std::endl;
9      }
10 }
11
12 int main()
13 {
14     fonction();
15     return 0;
16 }
```

Pour poser une condition, il suffit de faire un clic-droit sur un point d'arrêt et de choisir **Edit breakpoint...**. Une fenêtre apparaît. Dans la section **Advanced**, la ligne **Condition** permet de définir la condition à respecter pour que le débogueur s'arrête sur ce point d'arrêt. On peut ainsi demander à attendre que `i` vaille `95`.

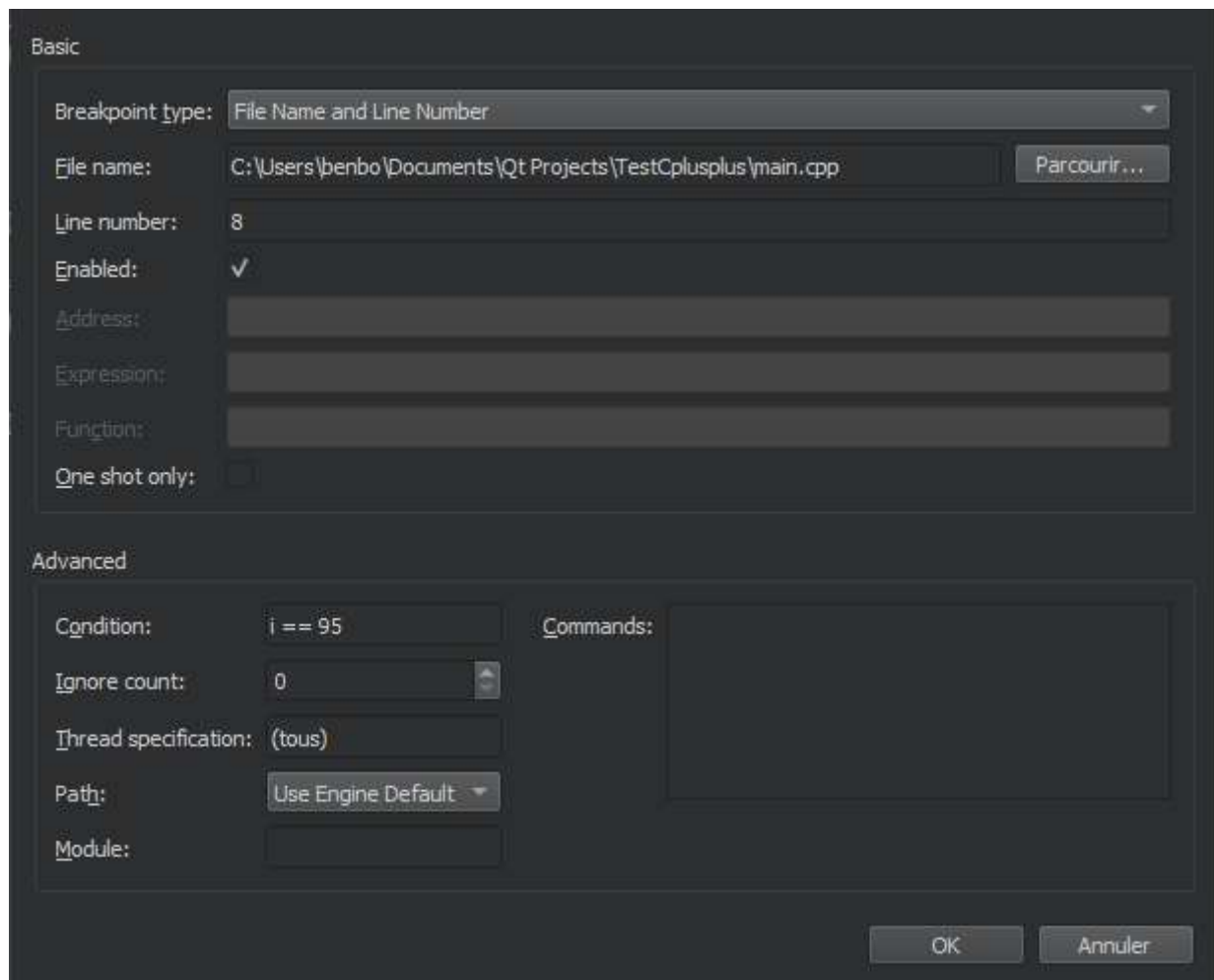



FIGURE IV.5.18. – Fenêtre de paramétrage d'un point d'arrêt.

Lancez le débogage et vous verrez que le programme s'exécutera sans interruption jusqu'à ce que `i` atteigne la valeur `95`, auquel cas le débogueur attendra nos instructions pour continuer.

IV.5.4.4. Aller plus loin

Comme il m'est impossible d'être exhaustif et que vous commencez à être des habitués de la documentation, je vous renvoie vers [la documentation Qt officielle](#) . Celle-ci est en anglais mais vous permettra d'aller plus loin dans votre apprentissage.

IV.5.5. En ligne de commande avec gdb

Nous avons dit, dans la partie précédente, que Qt Creator se base sur gdb pour offrir une expérience de débogage. Ce dernier s'utilise avec de nombreuses interfaces graphiques différentes, mais aussi **en ligne de commande**. Alors rouvrons notre shell et plongeons-nous à la découverte de gdb.

Pour lancer le débogage d'un programme, il suffit de taper **gdb** suivi du nom de ce programme. Celui-ci doit avoir été compilé avec les symboles de débogage. L'option **-quiet** évite l'affichage de toutes les informations légales et de licence.

```
1 > gdb -quiet programme.out
2 Reading symbols from programme.out...done.
3 (gdb)
```

IV.5.5.1. Poser un point d'arrêt

Poser un point d'arrêt avec gdb est faisable de trois façons différentes, selon ce qu'on veut faire.

- On indique un **endroit précis**, comme un numéro de ligne, une fonction, un fichier, etc. C'est un **point d'arrêt** classique (*break point*).
- Lors de la **modification** ou de la **lecture** d'une variable. C'est ce qu'on appelle un **point de surveillance** (*watch point*).
- Lors d'un certain **évènement particulier**, comme le lancement ou le rattrapage d'une exception, entre autres. C'est ce qu'on appelle un **point d'interception** (*catch point*).

Dans notre cas, contentons-nous de poser un point d'arrêt à une ligne précise, la numéro 8 du code suivant. Cela se fait en utilisant la commande **break** et en indiquant le numéro de la ligne voulue.

```
1 (gdb) break 8
2 Breakpoint 1 at 0xd0e: file main.cpp, line 8.
```

Maintenant, lançons l'exécution du programme avec la commande **run**.


```
1 (gdb) run
2 Starting program: /home/informaticienzero/Documents/programme.out
3
4 Breakpoint 1, fonction () at main.cpp:8
5 8         tableau.push_back(10);
```

Le débogueur s'est bien arrêté à la ligne 8, comme demandé. Il nous affiche en prime la fonction dans laquelle on se trouve, ainsi que le fichier, puis la ligne de code qui se trouve à la position demandée.

Dans le cas où l'on souhaite poser un point d'arrêt dans un autre fichier, il suffit de préciser celui-ci avant le numéro de la ligne, séparés par deux-points `:`.

```
1 (gdb) break autre.cpp:5
2 Breakpoint 2 at 0xd0e: file autre.cpp, line 5.
```

IV.5.5.2. Supprimer des points d'arrêt

Quand un point d'arrêt n'est plus utile, on peut **le supprimer**. On a pour cela plusieurs possibilités. Soit on se souvient de la position du point d'arrêt, auquel cas on le supprime avec la commande `clear position`, soit on se souvient de son numéro et on peut le supprimer avec `delete numéro`.

Et comme se souvenir du numéro est compliqué, on peut tout à fait **retrouver la liste des points d'arrêt** avec la commande `info breakpoints`.

```
1 (gdb) info breakpoints
2 Num      Type           Disp Enb Address                What
3 1        breakpoint     keep y   0x00000000000000d0e in test() at
   autre.cpp:5
4 2        breakpoint     keep y   0x00000000000000d9e in fonction() at
   main.cpp:6
```

IV.5.5.3. Désactiver des points d'arrêt

Parfois, on ne veut pas s'arrêter à un point d'arrêt donné, mais on ne veut pas non plus le supprimer car on en a besoin plus tard. Il suffit de **le désactiver** avec la commande `disable numéro`. Une fois qu'on en aura besoin, il suffira de **le réactiver** avec `enable numéro`.

IV.5.5.4. Afficher l'état d'une variable

Si on regardait l'état de notre tableau? Celui-ci est bien évidemment vide, mais demandons quand même à `gdb` de nous l'afficher. Il faut pour cela utiliser la commande `print` avec le nom de la variable voulue. Celle-ci doit exister à ce moment-là.

```
1 (gdb) print tableau
2 $1 = std::vector of length 0, capacity 0
3 (gdb) print a
4 $2 = (int &) @0x7fffffffede90: 0
5 (gdb) print x
6 No symbol "x" in current context.
```

Notre référence `a` existe, mais n'a pas encore de valeur, d'où le nombre aléatoire que vous verrez. Quant à `x`, le symbole n'existe pas car il n'y a aucune variable pourtant ce nom.

IV.5.5.5. Pas-à-pas

Bon, tout ça c'est bien, mais comment on passe à la suite? Comment exécuter les instructions qui suivent quand on est à un point d'arrêt? Nous avons deux choix, permettant de faire du **pas-à-pas détaillé**, en explorant la pile des appels, avec `step`, ou bien de **passer d'instruction en instruction** sans explorer le code des fonctions appelées avec `next`.

La sortie suivante montre que, quand je décide de faire `step` sur le point d'arrêt, `gdb` commence à me montrer en détail ce qui se passe et explore la fonction `std::vector::push_back`. Si je me contente de faire `next`, alors je passe simplement à la ligne d'après, soit le deuxième `push_back`.

```
1 (gdb) step
2 std::vector<int, std::allocator<int> >::push_back
   (this=0x7fffffffede90, __x=@0x7fffffffede7c: 10) at
   /usr/include/c++/7/bits/stl_vector.h:954
3 954         { emplace_back(std::move(__x)); }
4 (gdb) next
5 fonction () at main.cpp:9
6 9         tableau.push_back(20);
```

Chacune de ces commandes peut recevoir un entier en paramètre, indiquant de combien d'étapes on souhaite avancer. Ainsi, si je suis à la ligne 9 et que je fais `next 3`, j'avancerai de trois instructions. On saute les lignes 10 et 13 qui sont vides, on saute deux instructions aux lignes 11 et 12 et on arrive donc à la troisième après la ligne 9, qui se trouve ligne 14.

```
1 fonction () at main.cpp:9
2 9         tableau.push_back(20);
3 (gdb) next 3
4 14         std::cout << "Front : " << tableau.front() <<
        std::endl;
```

IV.5.5.6. Conditions

Dans le cas où l'on voudrait s'arrêter sur un point d'arrêt uniquement lors d'une certaine condition, c'est possible en rajoutant cette condition après **break**. Si je veux, dans le code suivant, m'arrêter quand **i** vaut **95**, je n'ai qu'à taper la commande ci-après.

```
1 #include <iostream>
2 #include <vector>
3
4 void fonction()
5 {
6     for (int i { 0 }; i < 100; ++i)
7     {
8         std::cout << i * 2 << std::endl;
9     }
10 }
11
12 int main()
13 {
14     fonction();
15     return 0;
16 }
```

```
1 > gdb -quiet programme.out
2 Reading symbols from programme.out...done.
3 (gdb) break 8 if i == 95
4 Breakpoint 1 at 0x89f: file main.cpp, line 8.
5 (gdb) run
6 Starting program: /home/informaticienzero/Documents/programme.out
7 0
8 2
9 // Des nombres.
10 186
11 188
12
13 Breakpoint 1, fonction () at main.cpp:8
14 8         std::cout << i * 2 << std::endl;
```

IV.5.5.7. Aller plus loin

Ce programme est vraiment complet et doté de nombreuses possibilités. Je vous encourage à en apprendre plus de votre côté, que vous vouliez l'utiliser en ligne de commande ou à travers l'interface graphique de votre choix. Vous pouvez commencer par [ce tutoriel](#) en français, qui vous montrera d'autres commandes possibles.

IV.5.5.8. En résumé


- Le débogueur est un précieux allié qui peut nous aider à traquer les *bugs* vicieux cachés dans nos codes.
- En posant des points d'arrêt, on demande au débogueur d'arrêter l'exécution du programme à une ligne précise.
- Certains points d'arrêt ne s'activeront que si une condition que nous avons définie devient vraie.
- On peut afficher la pile d'appel, pour voir la liste des fonctions appelées avant d'arriver à une ligne quelconque.
- On peut voir, à tout moment, l'état des variables et même définir nos propres expressions, ou espions, qui seront automatiquement mis à jour à la moindre modification.
- Il ne faut pas oublier de compiler le programme avec ses symboles de débogage.

IV.6. Une foule de bibliothèques

La bibliothèque standard, vous commencez à connaître et à vous sentir à l'aise avec. Mais soyons honnêtes, elle est limitée. Par exemple, impossible de faire communiquer deux programmes par le réseau ou de faire un jeu 2D avec. Ce n'est pas un problème, **il suffit d'installer des bibliothèques externes**, déjà programmées et testées par d'autres personnes.

Ce chapitre va aborder **l'installation et l'utilisation de bibliothèques externes**.

IV.6.1. Quelles bibliothèques choisir ?

La réponse à cette question dépend de ce que vous cherchez. C++ étant un langage populaire, vous trouverez des bibliothèques externes dans [de nombreux domaines](#) , comme les graphismes 2D, la communication réseau, la manipulation de sons, des outils mathématiques supplémentaires, etc. Ceci étant dit, nous sommes dans un cours d'introduction, donc je me dois de choisir pour vous. Dans ce chapitre, nous apprendrons donc à **installer et utiliser brièvement deux bibliothèques externes**, respectivement nommées **Boost** et **SFML**.


Vous aurez ainsi vu plusieurs manières d'installer une bibliothèque externe, donc vous ne serez pas perdus lorsque vous en aurez besoin. 🍊


i

Pas d'inquiétude

Une fois le principe bien compris, installer et utiliser d'autres bibliothèques externes n'aura rien de compliqué, il suffira de lire la documentation.

IV.6.1.1. Boost

La première que nous verrons, [Boost](#) , c'est un peu comme **l'extension officielle de la bibliothèque standard C++**. Au fil des années, de nombreuses fonctionnalités implémentées dans Boost se sont retrouvées intégrées à la bibliothèque standard C++. Citons par exemple `std::array`, `std::tuple` ou encore de nombreuses fonctions mathématiques.

L'installer, c'est s'offrir les services d'une bibliothèque de **grande qualité, répandue**, conçue par des développeurs experts, donc vous avez tout à gagner à apprendre à l'utiliser. Elle propose, entre autres, des **tests unitaires**, mais également un moyen d'interfacer son code avec Python, travailler avec des [quaternions](#)  et bien d'autres choses encore.

IV.6.1.2. SFML


L'autre que j'aimerais que vous découvriez est une **bibliothèque 2D** très connue et utilisée en C++, [SFML](#) . Elle vous permet de gérer **tout ce qui touche aux jeux 2D**, comme le son, les graphismes et le réseau. Et ne vous inquiétez pas, **elle est portable** et fonctionne sur de nombreuses plateformes, dont Windows et GNU/Linux.



FIGURE IV.6.1. – Exemple utilisant la SFML, tiré de la documentation.

En plus, la documentation est en français. C'est pas beau la vie? 🍊

IV.6.2. Généralités

La plupart des bibliothèques externes fonctionnent sur le même principe que la bibliothèque standard, c'est-à-dire qu'on a des **fichiers d'en-têtes** et des **fichiers objets**, entre autres. Les premiers sont nécessaires pour la compilation, les seconds pour le *linker*, comme nous l'avons vu dans le chapitre sur la compilation. Mais ce n'est pas tout. Souvent sont joints des **exemples**, ainsi qu'une version hors-ligne de la **documentation**.

IV.6.2.1. Statique ou dynamique ?

Les fichiers objets d'une bibliothèque externe peuvent être joints au programme de deux manières différentes.

- Avec une **bibliothèque statique**, les fichiers objets sont **directement liés au programme**. Celui-ci devient plus gros, mais est désormais utilisable tel quel, sans fichier supplémentaire à joindre avec l'exécutable.

IV. Interlude - Être un développeur

- Avec une **bibliothèque dynamique**, les fichiers objets sont chargés à l'**exécution** du programme. L'avantage, c'est qu'une même copie de la bibliothèque peut être utilisée par de nombreux programmes. L'inconvénient, c'est qu'il faut fournir ces fichiers avec l'exécutable du programme. Sous Windows, ce sont les fameux fichiers `.dll`.

La très grande majorité des bibliothèques externes sont utilisables de façon statique ou dynamique, au choix.

IV.6.2.2. *Debug ou release ?*

De la même manière que nos programmes, les bibliothèques externes sont **compilées tant en *debug* qu'en *release***. Cela permet d'offrir une expérience de débogage et des tests pendant qu'on développe notre programme, avec les symboles de débogage présents et une expérience optimisée et plus rapide une fois qu'on passe en *release*.

IV.6.3. Installer Boost

Commençons donc par Boost. Cette bibliothèque est composée majoritairement de fichiers d'en-tête, qu'on peut inclure simplement, sans plus d'histoire. Cependant, quelques parties demandent **d'être compilées**. Nous allons donc voir comment installer la version complète de Boost sur chaque plateforme.

IV.6.3.1. GNU/Linux

Vous êtes dans le cas le plus facile, puisqu'il y a de grandes chances que votre distribution préférée fournisse des paquets prêts à l'emploi. Par exemple, pour Ubuntu, c'est la commande suivante qui installe le tout.

```
1 sudo apt-get install libboost-all-dev
```

Renseignez-vous dans la documentation de votre distribution pour le nom exact du paquet.

IV.6.3.2. Windows

Les utilisateurs de Windows sont moins chanceux. De base, il faudrait **compiler nous-mêmes une version de Boost**. Pour ceux utilisant Visual Studio, une solution simple existe. Pour ceux utilisant Qt Creator... 🍊

IV.6.3.2.1. Visual Studio

Commencez par vous rendre [ici](#) . Sur la page des versions, prenez la dernière disponible (1.69 à l'heure où j'écris), puis téléchargez **boost_1_69_0-msvc-14.1-32.exe**, ainsi que **boost_1_69_0-msvc-14.1-64.exe** si vous avez un système d'exploitation 64 *bits*. Une fois l'exécutable téléchargé, installez-le dans le dossier de votre choix. Si vous prenez les deux, installez les deux dans le même dossier. Ne vous en faites pas, il n'y a pas de risque d'écrasement.

Maintenant, ouvrez un projet quelconque et rendez-vous dans ses propriétés, avec un clic-droit sur le nom du projet. Il va en effet falloir indiquer au compilateur **où se trouvent les fichiers de Boost**. Commençons par nous mettre en configuration générale, en mettant le champ **Configuration** à **Toutes les configurations** et **Plateforme** à **Toutes les plateformes**.

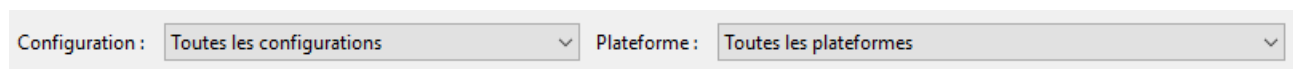


FIGURE IV.6.2. – Ce que vous devez avoir.

Incluons le dossier contenant tous les fichiers d'en-tête en allant dans **C/C + -> Général**, puis en faisant modifier sur le champ **Autres répertoires Include**. Dans le cadre du haut, mettez le chemin du dossier dans lequel vous avez installé Boost. Chez moi, le chemin complet est **C:\Programmation\Boost.1.69**.

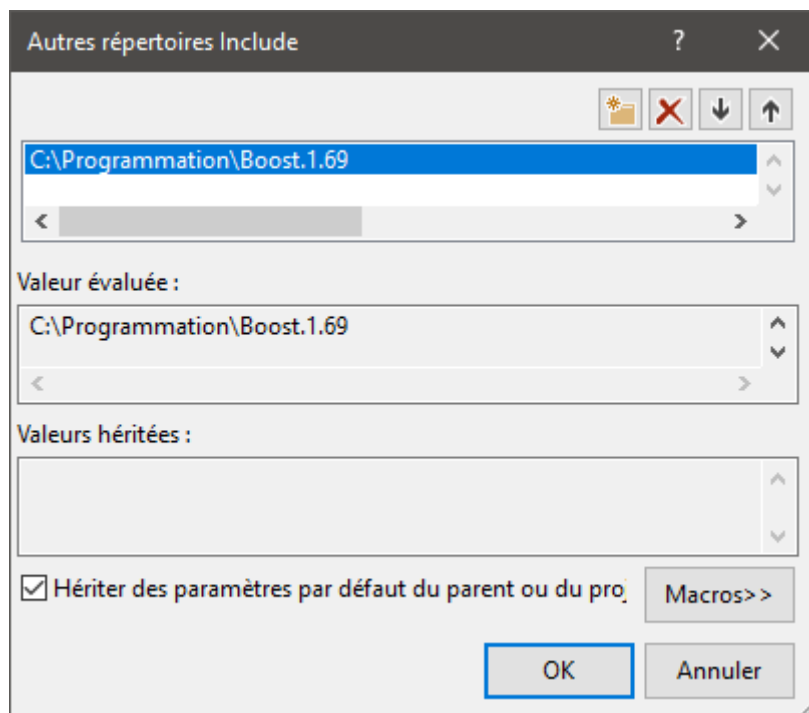


FIGURE IV.6.3. – Ce que vous devez avoir.

Maintenant, allons dans **Éditeur de liens -> Général** et modifions le champ **Répertoires de bibliothèques supplémentaires** pour rajouter le dossier contenant les fichiers objets. Dans le cas où vous n'avez que la version 32

IV. Interlude - Être un développeur

bits, modifiez-le et ajoutez le chemin où vous avez installé Boost plus **lib32-msvc-14.1**. Chez moi, le chemin complet est **C:\Programmation\Boost.1.69\lib32-msvc-14.1**.

Dans le cas où vous avez les deux versions, 32 *bits* et 64 *bits*, il va falloir **adapter le chemin donné**. Tout d'abord, modifiez le champ **Plateforme** en haut pour le mettre à **Win32**. Puis, dans **Éditeur de liens -> Général**, modifiez le champ **Répertoires de bibliothèques supplémentaires**. Ajoutez-y le chemin où vous avez installé Boost plus **lib32-msvc-14.1**. Une fois fait, cliquez sur **Appliquer** puis retourner modifier le champ **Plateforme** en le mettant cette fois à **x64**. Faites la même chose que précédemment mais en sélectionnant cette fois le dossier **lib64-msvc-14.1**

- Pour **Win32**, j'ai dans mon cas le chemin **C:\Programmation\Boost.1.69\lib32-msvc-14.1**.
- Pour **x64**, j'ai dans mon cas le chemin **C:\Programmation\Boost.1.69\lib64-msvc-14.1**.

Validez et tout est bon.

IV.6.3.2.2. Qt Creator

Quant à ceux utilisant Qt Creator, ils gagnent le droit de **compiler eux-mêmes Boost**. Ne vous inquiétez pas, ce n'est pas si compliqué qu'il n'y paraît et je vais tout vous expliquer. Il faut commencer par [télécharger](#) la dernière version disponible (1.69 à l'heure où j'écris) et la décompressez dans le dossier de votre choix.

Maintenant, on va lancer l'invite de commande installée en même temps que Qt Creator. Dans mon cas, je le trouve dans **Menu démarrer -> Qt -> 5.12.2 -> MinGW 7.3.0 (32 bits) -> Qt 5.12.2 (MinGW 7.3.0 32-bit)** (les numéros de version peuvent changer). Il va nous servir à compiler Boost.

Rendez-vous dans le dossier où vous avez décompressé Boost à l'aide de la console. Dans mon cas, c'est **C:\Users\informaticienzero\Downloads\boost_1_69_0**. La commande pour changer de dossier est **cd**.

```
1 cd C:\Users\informaticienzero\Downloads\boost_1_69_0
```

Une fois dedans, on va préparer l'installation en exécutant la commande suivante. Entre guillemets après **--prefix**, mettez le chemin de sortie où les bibliothèques composant Boost seront stockées. Pour moi, j'ai pris **C:\Programmation\Boost**.

```
1 bootstrap gcc --prefix="C:\Programmation\Boost"
```

Maintenant, il est temps de lancer la compilation pour de bon. Voici la ligne à taper. Oui, elle est très longue. 🍊

```
1 b2 install address-model=32 --build-type=complete architecture=x86
   threading=multi --build-dir="C:\Programmation\Boost"
   toolset=gcc debug release link=static runtime-link=static
   cxxflags="-std=c++17"
```

- L'option `address-model=32` indique qu'on veut compiler une version 32 *bits* de la bibliothèque.
- L'option `architecture=x86` indique qu'on compile pour un PC, qui utilisent quasiment tous cette architecture.
- L'option `--build-dir="C:\Programmation\Boost"` indique où on veut stocker les fichiers objets générés.
- Les options `debug release` indique qu'on veut disposer tant de la version *debug* que *release*.
- L'option `link=static` indique qu'on veut compiler une version statique uniquement, ce qui est fortement recommandé sous Windows.
- L'option `cxxflags="-std=c++17"` indique qu'on veut compiler avec C++17.

La compilation peut être assez longue donc lancez-la et faites autre chose en attendant. Une fois qu'elle a fini, il faut configurer Qt Creator. Cela se passe dans le fichier `.pro`. On commence par indiquer où trouver les fichiers d'en-tête.

```
1 # Adaptez en fonction du chemin d'installation.
2 INCLUDEPATH += C:/Boost/include/boost-1_69
```

Maintenant, on va ajouter le dossier contenant les **fichiers objets**.

```
1 # Adaptez en fonction du chemin d'installation.
2 LIBS += -L"C:/Boost/lib"
```

Voilà, l'installation est terminée, il ne restera plus qu'à ajouter individuellement les bibliothèques requises dans le fichier `.pro`.



Version 64 bits

Si vous voulez la version 64 *bits* de Boost, il faut refaire la même procédure mais avec la console **Qt 5.12.2 (MinGW 7.3.0 64-bit)**.

IV.6.3.3. Tester l'installation

Prenons par exemple le code d'illustration de [Boost.Log](#), un système d'écriture de messages de journalisation. Je le remets ci-dessous.

```
1 #include <boost/log/trivial.hpp>
2
3 int main()
4 {
5     BOOST_LOG_TRIVIAL(trace) << "A trace severity message";
6     BOOST_LOG_TRIVIAL(debug) << "A debug severity message";
7     BOOST_LOG_TRIVIAL(info) << "An informational severity message";
8     BOOST_LOG_TRIVIAL(warning) << "A warning severity message";
9     BOOST_LOG_TRIVIAL(error) << "An error severity message";
10    BOOST_LOG_TRIVIAL(fatal) << "A fatal severity message";
11
12    return 0;
13 }
```

Pour compiler en ligne de commande, il faut rajouter quelques options, dues à des dépendances du système de *logging* fourni par Boost.

- L'option `-static` indique au compilateur qu'on veut lier **statiquement** Boost.Log à notre programme.
- Les options `-lboost_log` et `-lboost_log_setup` permettent de lier les deux bibliothèques nécessaires à Boost.Log.
- Les options `-lboost_thread`, `-lboost_system` et `-lpthread` lient d'autres dépendances de Boost.Log à notre programme. On trouve notamment Boost.Thread et Boost.System, deux autres bibliothèques incluses par Boost, ainsi qu'une dépendance à une [bibliothèque commune](#) à toutes les distributions GNU/Linux et UNIX.

```
1 g++ -std=c++17 -static main.cpp -o main.out -lboost_log
   -lboost_log_setup -lboost_thread -lboost_system -lpthread
```

Pour ceux utilisant Qt Creator, on doit ajouter les lignes suivantes dans le fichier `.pro`. Ce sont les bibliothèques nécessaires pour compiler le code d'exemple de Boost.Log. On retrouve `log` et `log_setup`, ainsi que la bibliothèque `thread`. Notez qu'on fait la différence entre les versions *debug* (avec `d`) et *release* (sans `d`).

```
1 CONFIG(debug, debug|release) {
2     # Bibliothèques en debug.
3     LIBS += -lboost_log-mgw73-mt-sd-x64-1_69
4     LIBS += -lboost_log_setup-mgw73-mt-sd-x64-1_69
5     LIBS += -lboost_thread-mgw73-mt-sd-x64-1_69
6 }
7
8 CONFIG(release, debug|release) {
9     # Bibliothèques en release.
10    LIBS += -lboost_log-mgw73-mt-s-x64-1_69
```

```
11 LIBS += -lboost_log_setup-mgw73-mt-s-x64-1_69
12 LIBS += -lboost_thread-mgw73-mt-s-x64-1_69
13 }
```



32 bits et 64 bits

Si vous compilez avec MinGW 32 *bits*, alors il faut lier les bibliothèques 32 *bits*; pareillement pour 64 *bits*.

Dans tous les cas, une fois compilé et lancé, le programme affichera la sortie suivante, avec seulement l'heure changeant en fonction du moment où vous lancerez le programme.

```
1 [2019-03-23 19:13:42.109558] [0x00004e2c] [trace]    A trace
  severity message
2 [2019-03-23 19:13:42.111576] [0x00004e2c] [debug]    A debug
  severity message
3 [2019-03-23 19:13:42.111576] [0x00004e2c] [info]     An
  informational severity message
4 [2019-03-23 19:13:42.112561] [0x00004e2c] [warning] A warning
  severity message
5 [2019-03-23 19:13:42.112561] [0x00004e2c] [error]   An error
  severity message
6 [2019-03-23 19:13:42.112561] [0x00004e2c] [fatal]   A fatal
  severity message
```

IV.6.4. Installer SFML

Maintenant, abordons l'installation de la deuxième bibliothèque, la SFML. Encore une fois, nous allons détailler par environnement.


IV.6.4.1. GNU/Linux

Vous êtes encore des chanceux, puisque la plupart des distributions proposent des paquets permettant d'avoir une SFML à jour en une commande. Ainsi, avec Ubuntu, il suffit de taper la commande qui suit.


```
1 sudo apt-get install libsFML-dev
```

Renseignez-vous pour le nom exact du paquet pour votre distribution.


IV.6.4.2. Windows

Pour Windows, il va falloir installer manuellement. Pas de soucis, la documentation est très bien faite, en français et vous bénéficiez de nos explications en prime. Commençons par nous rendre sur la [page de téléchargement](#) . À l'heure où j'écris, elle propose la version 2.5.1 de SFML.

Plusieurs versions sont disponibles au téléchargement, en fonction du compilateur que vous utilisez. Dans notre cas, il s'agira de prendre soit la version **Visual C++ 15 (2017)** pour ceux ayant installé Visual Studio 2017, soit la version **GCC 7.3.0 MinGW** pour ceux utilisant Qt Creator. Dans les deux cas, prenez la version **32 bits**. Même si la très grande majorité des processeurs vendus aujourd'hui sont 64 *bits*, avec la version 32 *bits* vous pourrez cibler les deux-plateformes. Si vous prenez la version 64 *bits*, vous ne pourrez cibler que les ordinateurs ayant un processeur et un système d'exploitation 64 *bits*.

Dans tous les cas, vous obtenez une archive que vous pouvez décompresser **dans le dossier de votre choix**. Certains vont le mettre dans un dossier bien à part, comme `C:/Programmation/SFML`, d'autres préfèrent le [mettre dans le dossier de leur projet](#) , ce qui permet de distribuer les sources et de changer d'ordinateur en un simple copier-coller, car la configuration est enregistrée avec le projet. Ce choix est vôtre.

IV.6.4.2.1. Avec Visual Studio

Voici tout d'abord [la documentation](#)  détaillant l'installation avec Visual Studio. Elle est très claire et vous pouvez la suivre sans mon aide, vous en avez largement les capacités. 🍊

Néanmoins, pour ceux qui se sentiraient rassurés en me voyant faire, lisez la suite. D'abord, il faut modifier les propriétés de configuration du projet, en faisant un clic-droit dessus. Tout en haut à gauche, assurez-vous d'être en mode **Toutes les configurations** ainsi que **Toutes les plateformes**.

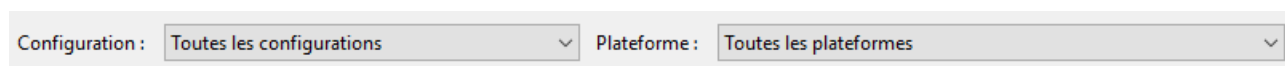


FIGURE IV.6.4. – Voici ce que vous devez avoir.

Il y a deux choses à faire qui sont communes à *debug* et *release*.

- Dans **C/C+ -> Général**, dans le champ **Autres répertoires Include**, faites modifier, puis dans le cadre du haut mettez le chemin du dossier **include** du répertoire où vous avez installé SFML.
- Dans **Éditeurs de liens -> Général**, dans le champ **Répertoires de bibliothèques supplémentaires**, faites modifier, puis dans le cadre du haut mettez le chemin du dossier **lib** du répertoire où vous avez installé SFML.

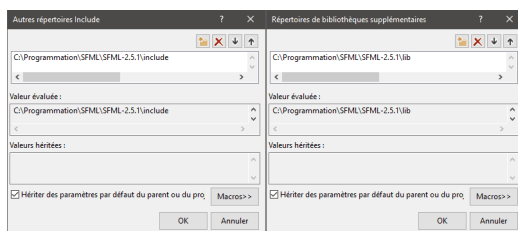


FIGURE IV.6.5. – Voilà ce que je mets dans mon cas.

Comme SFML est divisée en plusieurs modules, il faut ajouter plusieurs bibliothèques en particulier pour que notre code fonctionne. Dans l'étape précédente, on a indiqué au *linker* où chercher. Maintenant, on va indiquer exactement celles qu'on veut qu'il prenne.

Cette fois, le travail est à faire en deux fois, pour *debug* et pour *release*. Passez donc en configuration **Debug** puis allez dans **Éditeur de liens -> Entrée**. Là, modifiez le champ **Dépendances supplémentaires**, faites modifier puis ajouter, dans le cadre du haut, les valeurs suivantes.

- sfml-graphics-d.lib
- sfml-system-d.lib
- sfml-window-d.lib

i

Nom de la bibliothèque

Le **d** dans le nom des fichiers indique qu'on a affaire à la version *debug* de la bibliothèque. Cette convention est souvent reprise.

Faite la même chose en mode **Release** mais prenez cette fois les versions ne contenant pas **d**.

- sfml-graphics.lib
- sfml-system.lib
- sfml-window.lib

La configuration du projet est terminée, mais il reste une dernière étape. SFML ayant été installée de manière dynamique, il faut ajouter les DLLs, contenues dans le dossier **bin**, au projet. Copiez-collez les dans le répertoire du projet et tout est bon.

IV.6.4.2.2. Avec Qt Creator

Ouvrez votre projet, faites un clic-droit dessus et cliquez sur **Ajouter une bibliothèque...**. Dans la fenêtre suivante, cliquez sur **Bibliothèque externe**. Sur la fenêtre suivante, cliquez sur **Fichier de bibliothèque** puis rendez-vous dans le dossier **lib** de la SFML et choisissez **libsFML-main.a**. Dans la liste des plateformes, décochez Linux et Mac. Enfin, vérifiez que l'option **Ajouter un suffixe "d"** pour la version *debug* soit cochée.

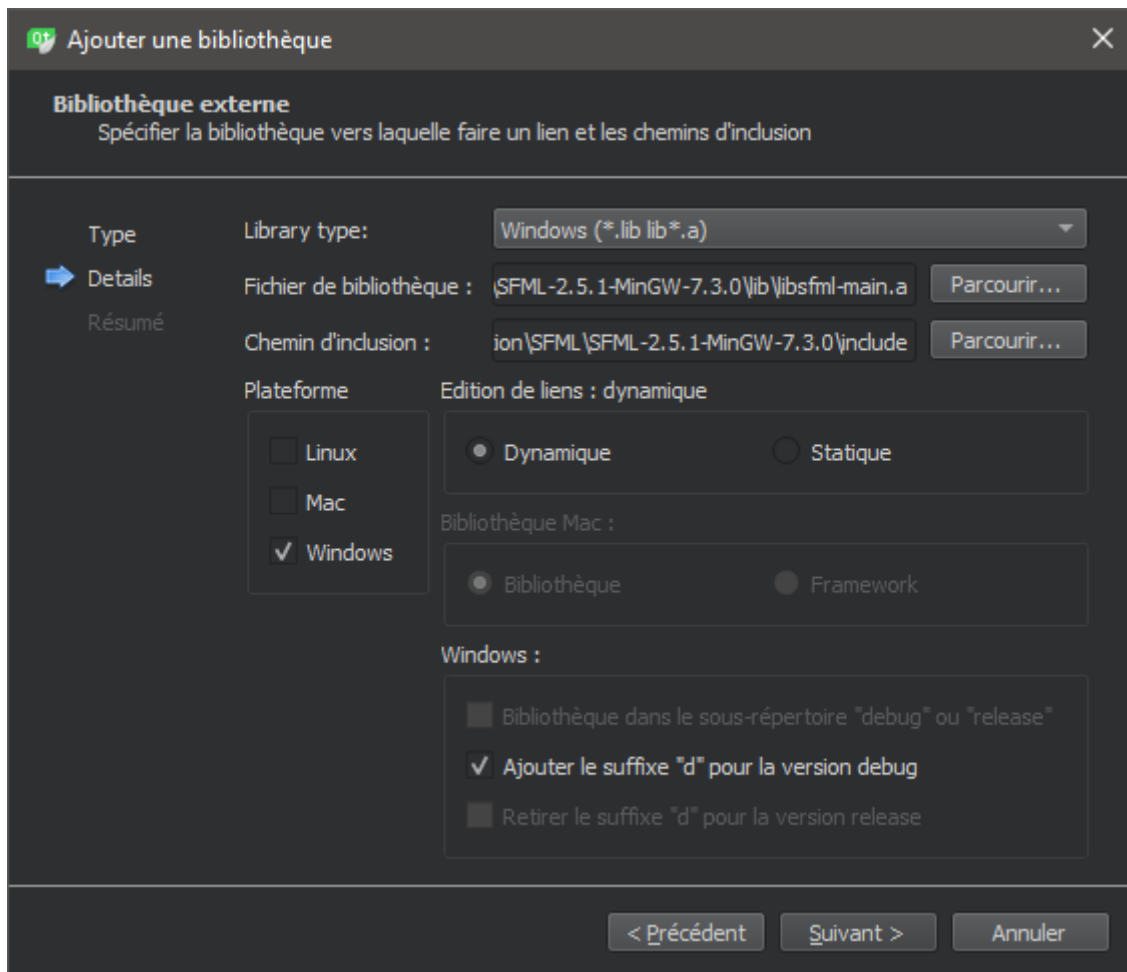


FIGURE IV.6.6. – Voilà ce que vous devriez avoir.

Validez et ouvrez votre fichier **.pro**, que nous allons devoir modifier. De nouvelles lignes ont en effet été ajoutées.

```
1 INCLUDEPATH += $$PWD/../../../../../../../../Programmation/SFML/SFML-2.5.1-
  MinGW-7.3.0/include
2 DEPENDPATH += $$PWD/../../../../../../../../Programmation/SFML/SFML-2.5.1-
  MinGW-7.3.0/include
```

Ces lignes indiquent où chercher les fichiers d'en-tête que nous incluons, nous n'allons donc plus y toucher. Par contre, les deux autres lignes nécessitent une petite modification.

```
1 win32:CONFIG(release, debug|release): LIBS +=
  -L$$PWD/../../../../../../../../Programmation/SFML/SFML-2.5.1-MinGW-
  7.3.0/lib/
  -lsfml-main
```

```
2 else:win32:CONFIG(debug, debug|release): LIBS +=  
  -L$${PWD}/../../../../../../../../Programmation/SFML/SFML-2.5.1-MinGW-  
  7.3.0/lib/  
  -lsfml-maind
```

La version de *debug* n'est pas bonne, car il manque un tiret -. Il faut donc remplacer `-lsfml-maind` par `-lsfml-main-d`. Ensuite, comme nous voulons charger d'autres modules, nous allons dupliquer ces lignes plusieurs fois, pour être en mesure d'ajouter **libsFML-window**, **libsFML-graphics** et **libsFML-system**. Voici les lignes à ajouter.

```
1 win32:CONFIG(release, debug|release): LIBS +=  
  -L$${PWD}/../../../../../../../../Programmation/SFML/SFML-2.5.1-MinGW-  
  7.3.0/lib/  
  -lsfml-window  
2 else:win32:CONFIG(debug, debug|release): LIBS +=  
  -L$${PWD}/../../../../../../../../Programmation/SFML/SFML-2.5.1-MinGW-  
  7.3.0/lib/  
  -lsfml-window-d  
3  
4 win32:CONFIG(release, debug|release): LIBS +=  
  -L$${PWD}/../../../../../../../../Programmation/SFML/SFML-2.5.1-MinGW-  
  7.3.0/lib/  
  -lsfml-graphics  
5 else:win32:CONFIG(debug, debug|release): LIBS +=  
  -L$${PWD}/../../../../../../../../Programmation/SFML/SFML-2.5.1-MinGW-  
  7.3.0/lib/  
  -lsfml-graphics-d  
6  
7 win32:CONFIG(release, debug|release): LIBS +=  
  -L$${PWD}/../../../../../../../../Programmation/SFML/SFML-2.5.1-MinGW-  
  7.3.0/lib/  
  -lsfml-system  
8 else:win32:CONFIG(debug, debug|release): LIBS +=  
  -L$${PWD}/../../../../../../../../Programmation/SFML/SFML-2.5.1-MinGW-  
  7.3.0/lib/  
  -lsfml-system-d
```

Enfin, il reste à **ajouter les DLLs au dossier de sortie**. Toutes celles contenant `-d` vont dans le répertoire *Debug*, celles sans dans le répertoire *Release*.

IV.6.4.3. Tester l'installation

Il ne reste plus qu'à tester en utilisant le code donné en exemple à la fin de la page de documentation. Si vous voyez un cercle vert s'afficher, c'est gagné.


```
1 #include <SFML/Graphics.hpp>
2
3 int main()
4 {
5     sf::RenderWindow window(sf::VideoMode(200, 200),
6                             "SFML works!");
7     sf::CircleShape shape(100.f);
8     shape.setFillColor(sf::Color::Green);
9
10    while (window.isOpen())
11    {
12        sf::Event event;
13        while (window.pollEvent(event))
14        {
15            if (event.type == sf::Event::Closed)
16                window.close();
17        }
18
19        window.clear();
20        window.draw(shape);
21        window.display();
22    }
23
24    return 0;
25 }
```

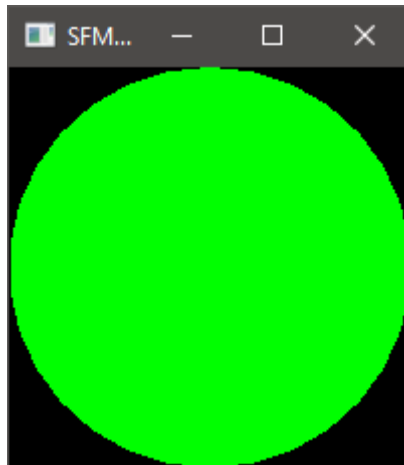


FIGURE IV.6.7. – Un beau cercle vert en guise de résultat.

IV.6.4.4. En résumé

- Quand une fonctionnalité dont nous avons besoin n'existe pas dans la bibliothèque standard, on peut choisir d'installer des bibliothèques externes.

IV. Interlude - Être un développeur

- Boost est une des bibliothèques externes les plus célèbres et les plus utilisées en C++. Elle comporte de nombreuses fonctionnalités comme le *logging*, des fonctions mathématiques ou la communication réseau.
- SFML est une bibliothèque réputée permettant de faire, entre autres, des jeux 2D.
- Chaque bibliothèque externe vient avec sa documentation expliquant comment l'installer et l'utiliser. C'est votre meilleure amie et le premier réflexe à avoir quand vous avez besoin d'une information.

IV.7. Améliorer ses projets

Vous avez, au cours de cette troisième partie, énormément appris sur ce qu'être un développeur signifie. Bien entendu, il y a encore de nombreuses choses à apprendre et cette section ne pourra jamais être exhaustive. Il est temps pour nous de la conclure avec un dernier chapitre présentant quelques outils qui vous seront quasi-indispensables pour passer au niveau supérieur.

Ce chapitre va présenter plusieurs outils pour **améliorer le développement de vos projets**.

IV.7.1. git— Sauvegarder et versionner son code

IV.7.1.1. La problématique

Sauvegarder, c'est important. Peu importe qu'on parle de code, de photos, de documents, ne pas avoir de sauvegarde peut nous créer de gros problèmes quand un document est corrompu ou effacé par erreur. Ça nous est tous arrivé au moins une fois et c'est très embêtant.

Nos codes n'échappent pas à cette logique. Il est important de **sauvegarder notre code régulièrement**. Quand nos programmes sont tout petits, une solution consistant à copier/coller le dossier du projet peut suffire. Mais cette solution apporte en réalité plus d'inconvénients qu'autre chose.



- Nous n'avons aucun moyen de voir **l'historique des modifications** d'un fichier.
- C'est très **lourd** et **encombrant** puisqu'on se retrouve avec de plus en plus de dossiers.
- Si deux dossiers ne contiennent pas les mêmes fichiers, **les fusionner peut s'avérer complexe**.
- Dans le cas où on travaille à plusieurs sur le projet, **impossible de savoir qui a quelle version**.
- En cas de *bug*, **revenir à une ancienne version est complexe**, voire impossible.

Heureusement et comme toujours, des gens très doués ont créé des outils adaptés, efficaces et il ne nous reste qu'à les utiliser. Ces outils s'appellent des **gestionnaires de code source** [↗](#). Ils répondent à toutes les problématiques que nous avons soulevées plus haut. Le plus célèbre, utilisé et apprécié s'appelle **git** [↗](#).

IV.7.1.2. Installation

Dans le cas où vous êtes sous Windows, téléchargez la version la plus récente et installez-la dans le répertoire de votre choix. Pour ceux utilisant GNU/Linux, le paquet **git** est très certainement déjà installé. Dans le cas contraire, ce qui serait extrêmement étonnant, vous pouvez l'obtenir avec votre gestionnaire de paquet.

```
1 sudo apt install git
```

Bien qu'il existe des surcouches graphiques, je vous conseille d'apprendre à **utiliser git en ligne de commande**. De cette manière, vous serez autonomes peu importe le système d'exploitation ou l'environnement sur lesquels vous travaillerez, et vous en aurez une utilisation **plus souple**. Nous allons voir les bases dans ce cours, mais pour aller plus loin, il faudra vous baser sur des ressources externes. Je vous conseille notamment [celle-ci](#) , pour un guide de base, et le [Pro Git Book](#) , une référence très complète, et traduite en français qui plus est. 🍏



Outil fondamental

Git est vraiment un outil fondamental pour le développeur, il est très rentable de passer du temps à apprendre à le maîtriser. À la fin de ce cours, je vous suggère fortement de jeter un œil attentif au Pro Git Book.

Sous GNU/Linux, vous n'avez qu'à lancer les commandes directement depuis le terminal. Pour Windows, il faut lancer **Git Bash**, qui se trouve dans le dossier **Git** du Menu Démarrer.

IV.7.1.3. Initialiser git

La première chose à faire est d'**entrer un identifiant**, une **adresse mail** et **l'éditeur de votre choix**. Cela permet de tracer qui a fait quelle modification. Les deux premières commandes à entrer sont les suivantes.

```
1 > git config --global user.name "nom"
2 > git config --global user.email mail
3 > git config --global core.editor Éditeur
```

IV.7.1.4. Créer un dépôt

Maintenant, nous allons indiquer à git que nous voulons **sauvegarder le dossier contenant notre code source**. Il suffit de se positionner dans le répertoire voulu et de lancer la commande **git init**. Pour Windows, je procède ainsi, avec un projet Visual Studio en exemple.

```
1 informaticienzero@DESKTOP-GGDRVV5 MINGW64 ~
2 $ cd /d/Documents/Visual\ Studio\ 2017/Projects/TestCpp/
3 $ git init
```

Pour GNU/Linux, faites de même. Je prends ici un simple répertoire qui contient mes sources.

```
1 informaticienzero@DESKTOP-GGDRVV5:~$ cd Documents/TestCpp
2 informaticienzero@DESKTOP-GGDRVV5:~/Documents/TestCpp$ git init
```

Votre projet est maintenant géré par git. C'est aussi simple que ça. 🍊

IV.7.1.5. Connaître l'état de ses fichiers

Et si nous demandions à git de **nous donner les fichiers modifiés**? Cela se fait très simplement en tapant `git status`. Pour le cas d'un projet fraîchement initialisé, aucun fichier n'étant suivi, on obtient un message proche du message suivant.

```
1 informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
   Studio 2017/Projects/ZdSCpp (master)
2 $ git status
3 On branch master
4 No commits yet
5
6 Untracked files:
7   (use "git add <file>..." to include in what will be committed)
8
9       ZdSCpp/main.cpp
10
11 nothing added to commit but untracked files present (use "git add"
    to track)
```

Modifions donc un fichier quelconque. Dans mon cas, je supprime tout le contenu de `main.cpp` pour ne plus afficher qu'un message. Je relance la commande précédente et cette fois, il y a du changement.

```
1 informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
   Studio 2017/Projects/ZdSCpp (master)
2 $ git status
3 On branch master
4 Changes not staged for commit:
5   (use "git add <file>..." to update what will be committed)
6   (use "git checkout -- <file>..." to discard changes in working
   directory)
7
8       modified:   ZdSCpp/main.cpp
9
10 no changes added to commit (use "git add" and/or "git commit -a")
```

On voit bien que git a noté la modification du fichier `main.cpp`. Et si j'ajoute un nouveau fichier, comme un simple document texte?

```
1 informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
   Studio 2017/Projects/ZdSCpp (master)
2 $ git status
3 On branch master
4 Changes not staged for commit:
5   (use "git add <file>..." to update what will be committed)
6   (use "git checkout -- <file>..." to discard changes in working
   directory)
7
8       modified:   ZdSCpp/main.cpp
9
10 Untracked files:
11   (use "git add <file>..." to include in what will be committed)
12
13       ZdSCpp/Fichier_texte.txt
14
15 no changes added to commit (use "git add" and/or "git commit -a")
```

Cette fois, il nous indique qu'un fichier non suivi a été ajouté. Cela signifie que git ne le versionnera pas et n'enregistrera pas son historique.

IV.7.1.6. Ajouter un fichier

Il faut donc être en mesure de dire à git de **suivre, ou de ne plus suivre, tel ou tel fichier**. Dans le premier cas, cela se fait avec la commande `git add nom_fichier`.

```
1 informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
   Studio 2017/Projects/ZdSCpp (master)
2 $ git add ZdSCpp/Fichier_texte.txt
3
4 informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
   Studio 2017/Projects/ZdSCpp (master)
5 $ git status
6 On branch master
7 Changes to be committed:
8   (use "git reset HEAD <file>..." to unstage)
9
10       new file:   ZdSCpp/Fichier_texte.txt
11
12 Changes not staged for commit:
13   (use "git add <file>..." to update what will be committed)
14   (use "git checkout -- <file>..." to discard changes in working
   directory)
15
16       modified:   ZdSCpp/main.cpp
```

IV. Interlude - Être un développeur

Le fichier ajouté se trouve maintenant sous la section «*Changes to be committed*», soit les changements à enregistrer. Au contraire, le fichier `main.cpp` ne le sera pas. Il faut l'ajouter lui aussi, pour que les modifications apportées à ce fichier soit elles aussi enregistrées.

```
1 informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
   Studio 2017/Projects/ZdSCpp (master)
2 $ git add ZdSCpp/main.cpp
3
4 informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
   Studio 2017/Projects/ZdSCpp (master)
5 $ git status
6 On branch master
7 Changes to be committed:
8   (use "git reset HEAD <file>..." to unstage)
9
10      new file:   ZdSCpp/Fichier_texte.txt
11      modified:  ZdSCpp/main.cpp
```

IV.7.1.7. Valider les modifications

Nous avons vérifié, tout est bon, nous voulons enregistrer ce que nous avons fait. On parle de **commiter** ou de **faire un commit**. On utilise la commande suivante `git commit -m "Message"`, qui permet d'enregistrer les modifications avec un message explicatif.

```
1 informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
   Studio 2017/Projects/ZdSCpp (master)
2 $ git commit -m
   "Réécriture de main.cpp et ajout d'un document texte."
3 [master d83131f] Réécriture de main.cpp et ajout d'un document
   texte.
4 2 files changed, 7 insertions(+), 13 deletions(-)
5 create mode 100644 ZdSCpp/Fichier_texte.txt
6 rewrite ZdSCpp/main.cpp (93%)
```



Message explicatif

Le message accompagnant le commit doit être **de qualité**, puisqu'il indique ce que vous avez fait. Des messages comme `correction`, `xxx` ou `changement` sont donc très mauvais puisqu'ils ne sont pas clairs du tout.

IV.7.1.8. Voir l'historique

Vous pouvez voir l'évolution de votre code en demandant à git de vous afficher l'historique des *commits*. Dans notre cas, nous en avons trois. Le premier affiché est le dernier en date, celui que nous venons de faire. Les deux autres ont automatiquement été faits par git lors de l'initialisation du dépôt.

```
1  informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
   Studio 2017/Projects/ZdSCpp (master)
2  $ git log
3  commit d83131f1a0f860d40e06921c113affd16700c3e7 (HEAD -> master)
4  Author: informaticienzero <informaticienzero@mail.com>
5  Date:   Mon Mar 25 12:05:25 2019 +0100
6
7      Réécriture de main.cpp et ajout d'un document texte.
8
9  commit 242b99d89769b2d409213db3d25c1f60c6c9088a
10 Author: informaticienzero <informaticienzero@mail.com>
11 Date:   Sun Mar 24 12:13:46 2019 +0100
12
13     Ajoutez des fichiers projet.
14
15 commit 40f39687c1ad0ef616c7d6fae8549865cb5c4c54
16 Author: informaticienzero <informaticienzero@mail.com>
17 Date:   Sun Mar 24 12:13:42 2019 +0100
18
19     Ajoutez les fichiers .gitignore et .gitattributes.
```

Notez que l'auteur et la date sont affichés, ainsi que le *checksum*, la somme de contrôle identifiant de manière unique chaque *commit*.

IV.7.1.9. Bloquer certains fichiers

Si versionner le code est une bonne pratique, certains fichiers ne doivent pas être intégrés au dépôt. C'est le cas des fichiers objets, entre autres. Ceux-ci sont des produits de la compilation, donc rapidement périmés, pas adaptés en cas de passage à un autre système d'exploitation ou un autre compilateur, etc. On veut donc **les exclure**. De même, on n'inclut généralement pas les exécutables, les fichiers de configuration de l'éditeur utilisé, etc.

Pour ce faire, on dispose d'un fichier **.gitignore**, un bête fichier texte qui indique à git que certains fichiers ou certaines extensions sont à exclure. Il suffit de le créer et d'ajouter dedans **les règles souhaitées**. Par exemple, exclure tous les fichiers objets avec l'extension **.o** se fait en ajoutant la ligne ***.o**.



Astuce Windows

Si vous tentez de créer un fichier nommé **.gitignore** avec l'explorateur Windows, celui-ci protestera que vous n'avez pas entré de nom de fichier. Une astuce consiste à créer un fichier s'appelant **.gitignore.** (notez le point final supplémentaire), que Windows va automatiquement renommer correctement.

Dans le cas d'un projet Visual Studio, je me mets à la racine du projet, dans le même dossier où se trouve la solution **.sln** et je crée le fichier **.gitignore**. Les valeurs que je mets dedans sont celles que Visual Studio recommande, que j'ai pris de précédents projets et que je vous mets ci-dessous. On retrouve notamment les dossiers *Debug* et *Release* qui sont exclus, ainsi que les fichiers objets **.obj**.

☞ Fichier **.gitignore** pour Visual Studio

Pour GNU/Linux, pareil, créer ce fichier puis ajouter dedans les lignes ci-dessous.

☞ Fichier **.gitignore** pour GNU/Linux

Je vous fournis ici des fichiers pré-remplis, mais quand vous aurez acquis de l'expérience, vous saurez les adapter à vos projets et vos contraintes.

IV.7.1.9.1. Exemple concret

Je suis sous GNU/Linux et je compile un fichier source pour obtenir mon programme. Sans fichier **.gitignore**, mon exécutable est inclus aussi et je ne veux pas.

```
1 informaticienzero@DESKTOP-GGDRV5:~/Documents/TestCpp$ git status
2 Sur la branche master
3
4 Aucun commit
5
6 Modifications qui seront validées :
7   (utilisez "git rm --cached <fichier>..." pour désindexer)
8
9     nouveau fichier : main.cpp
10    nouveau fichier : programme.out
```

Si je crée mon fichier **.gitignore** comme vu juste avant, cette fois, mon exécutable est bien exclu et je peux commiter mon code.

```
1 informaticienzero@DESKTOP-GGDRVV5:~/Documents/TestCpp$ git status
2 Sur la branche master
3
4 Aucun commit
5
6 Modifications qui seront validées :
7   (utilisez "git rm --cached <fichier>..." pour désindexer)
8
9     nouveau fichier : .gitignore
10    nouveau fichier : main.cpp
```

IV.7.1.10. Aller plus loin

Comment faire pour supprimer un fichier? Comment revenir à un commit précédent? Qu'est-ce que le concept de branches et comment l'utiliser? Toutes ces questions trouvent leurs réponses dans le [Pro Git Book](#) [↗](#), que je vous encourage une fois de plus à lire. Il est d'excellente qualité, progressif et vous enseignera tout ce que vous devez savoir sur git.

IV.7.1.11. Visual Studio

Il est possible de gérer son code source avec git **directement depuis Visual Studio**. Regardez si vous avez la même petite ligne en bas à droite.

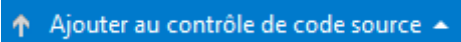


FIGURE IV.7.1. – Avez-vous cette ligne ?

Si ce n'est pas le cas, il suffit d'installer une extension. Dans le menu Démarrer, cherchez le programme qui s'appelle **Visual Studio Installer**. Ce programme accompagne Visual Studio 2019 et permet d'installer des mises à jour et des composants supplémentaires.

Cliquez sur **Modifier -> Composants individuels**, puis descendez tout en bas chercher la ligne **GitHub Extension for Visual Studio**. Cochez-la et cela lancera son installation. Une fois fini, rouvrez Visual Studio, puis votre projet et vous verrez vous aussi la même ligne que moi.

Cliquez dessus, sélectionnez **git** et voilà, votre projet est géré par git lui aussi. Dans l'onglet **Team Explorer** en haut à droite, vous pouvez faire les mêmes opérations que nous avons vu plus haut, directement dans Visual Studio, sans passer par la console.

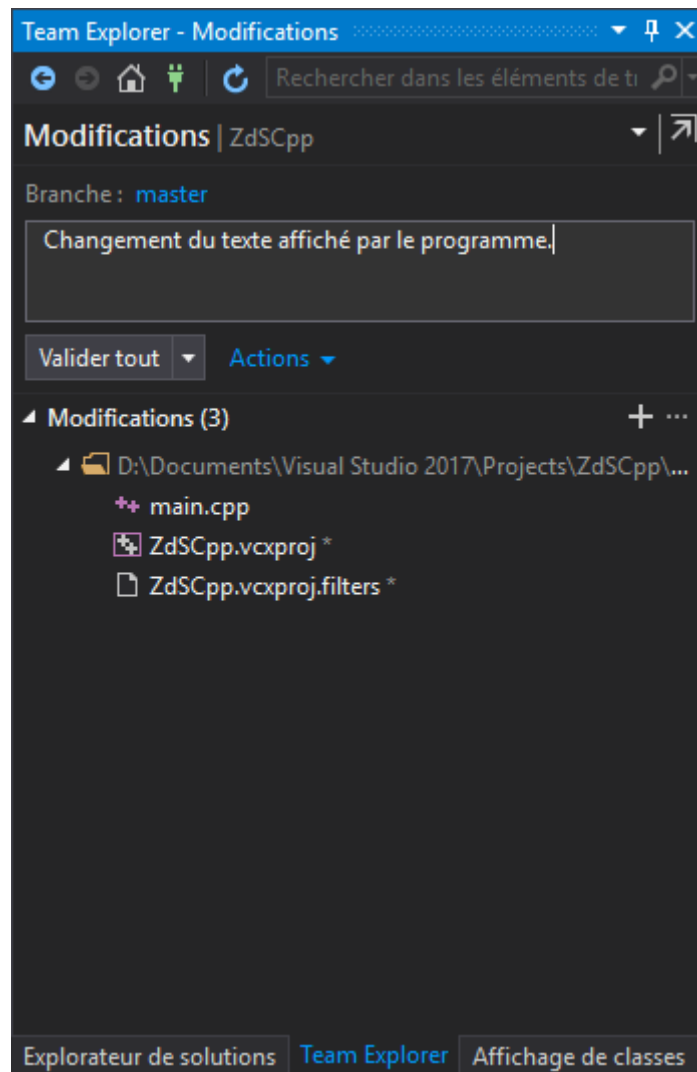


FIGURE IV.7.2. – Je peux ici taper mon message de commit, voir les fichiers modifiés puis valider.

Le guide officiel contient [une page](#) dédiée au sujet.

IV.7.2. GitHub—Partager son code

IV.7.2.1. La problématique

Notez que, dans le nom de cet outil, on retrouve Git. C'est normal, car les deux outils sont fortement liés. Si Git permet de sauvegarder l'état de son code, d'en vérifier l'historique, d'annuler ou de retravailler des *commits*, etc, il n'en reste pas moins que **votre code n'est disponible que sur votre ordinateur**. GitHub est là en complément.

[GitHub](#) est un **service en ligne** qui permet d'**héberger ses dépôts**. Puisque ceux-ci sont sur Internet, librement disponibles, vous êtes en mesure de partager votre code avec le monde entier. D'autres personnes peuvent ainsi le récupérer, le modifier et envoyer les modifications. C'est le fonctionnement et la base des projets dits **open-source**. Par exemple, Zeste de Savoir,

IV. Interlude - Être un développeur

le site sur lequel vous lisez ce cours, [utilise GitHub](#) pour stocker son code source et permettre à plusieurs développeurs de collaborer.

Parmi les autres avantages qu'on peut citer, utiliser un service en ligne nous **protège des pannes** et autres effacements sur notre poste de travail. Si demain votre disque dur meurt, le fait d'utiliser git seul pour versionner un projet ne vous sera d'aucune aide. Par contre, s'il est hébergé sur un service en ligne, comme GitHub, il suffira de récupérer les sources et le développement peut repartir.



Conseil

Lors de la création de votre compte, utilisez la même adresse mail que vous avez utilisé lors de la configuration de git.

IV.7.2.2. Création d'un projet

Je vais partir du principe que vous avez créé votre compte et que tout est bon. La première étape est de **créer un nouveau projet**. Pour illustrer le principe, nous allons créer un projet **Discographie**, pour permettre de sauvegarder et partager le code du T.P de la partie II du cours.

Create a new repository

A repository contains all project files, including the revision history.

Owner: informaticienzero / Repository name *: TutorielCpp-Discographie ✓

Great repository names are short and memorable. Need inspiration? How about [bug-free-succotash](#)?

Description (optional): Le T.P de discographie du tutorial C++ de Zeste de Savoir.

☒ Public
Anyone can see this repository. You choose who can commit.

☐ Private
You choose who can see and commit to this repository.

☒ Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None ⓘ

Create repository

FIGURE IV.7.3. – Ma page est noire parce que j’ai un thème foncé activé, mais sinon les informations contenues dedans sont les mêmes.

Plusieurs choses sont à indiquer.

- Le **nom du projet**.
- Une **courte description**, qui indique à quoi on a affaire.
- Le choix de la **visibilité**. Public signifie que tout le monde verra votre projet, alors que *private* permet de la restreindre à vous et aux utilisateurs que vous autoriserez.
- *Initialize this repository with a README* permet de créer, en plus du projet, le fameux **fichier README**, qui contient souvent la description du projet, les aspects légaux, les informations d’installation, etc.
- On peut ajouter également un **fichier .gitignore** et **une licence**. Le premier, vous connaissez. Le deuxième indique sous quels termes vous souhaitez partager votre code. Certaines licences permettent de réutiliser le code dans des applications payantes alors que d’autres l’interdisent, etc. De très nombreuses variantes sont disponibles, mais tout ça ne nous intéresse pas dans notre cas.

Quand tout est bon, vous n’avez plus qu’à cliquer sur le bouton vert **Create repository**. Voici [le lien](#) ↗.

On va partir du principe que vous avez initialisé le dépôt localement, créé un **.gitignore** et que tout est commité.

```
1  informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
   Studio 2017/Projects/Discographie (master)
2  $ git ls-files
3  .gitattributes
4  .gitignore
5  Discographie.sln
6  Discographie/Discographie.vcxproj
7  Discographie/Discographie.vcxproj.filters
8  Discographie/discographie.cpp
9  Discographie/discographie.hpp
10 Discographie/donnees_disco.cpp
11 Discographie/donnees_disco.hpp
12 Discographie/donnees_disco_tests.cpp
13 Discographie/donnees_disco_tests.hpp
14 Discographie/main.cpp
15 Discographie/systeme_commandes.cpp
16 Discographie/systeme_commandes.hpp
17 Discographie/utils.cpp
18 Discographie/utils.hpp
19
20 informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
   Studio 2017/Projects/Discographie (master)
21 $ git status
22 On branch master
23 nothing to commit, working tree clean
```

IV. Interlude - Être un développeur

Il faut maintenant indiquer à git où se trouve le répertoire distant dans lequel nous voulons stocker le code et que nous venons de créer. Cela se fait avec la commande `git remote add origin URL`, où URL est l'adresse de votre dépôt. Celle-ci s'obtient en cliquant sur le bouton vert `Clone or Download`.

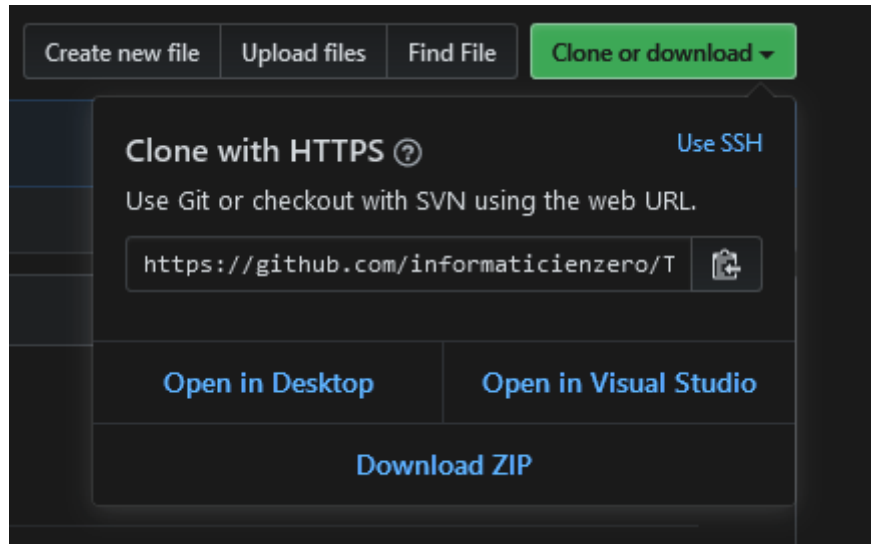


FIGURE IV.7.4. – L'URL de mon projet.

```
1 informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
  Studio 2017/Projects/Discographie (master)
2 $ git remote add origin
  https://github.com/informaticienzero/TutorielCpp-
  Discographie.git
```

IV.7.2.3. Récupérer localement les modifications

Maintenant qu'on est connecté avec le dépôt distant, nous sommes capables de **récupérer des modifications faites par d'autres**. Dans notre cas, personne d'autre n'a touché au projet, mais comme nous avons créé un fichier README, qui n'est pas présent localement, il faut se **synchroniser**. C'est ce que la commande `git pull origin master --allow-unrelated-histories -m "Message"` fait.

L'option `origin` désigne le dépôt distant et `master` représente la branche sur laquelle on travaille, celle par défaut. Quant au message, il résume le pourquoi de l'opération.

```
1 informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
  Studio 2017/Projects/Discographie (master)
2 $ git pull origin master --allow-unrelated-histories -m
  "Récupération du dépôt distant."
3 From https://github.com/informaticienzero/TutorielCpp-Discographie
```

```
4 * branch          master      -> FETCH_HEAD
5 Merge made by the 'recursive' strategy.
6 README.md | 2 ++
7 1 file changed, 2 insertions(+)
8 create mode 100644 README.md
```

Dans le cas où vous travaillez à plusieurs sur le même dépôt distant, il est important de **régulièrement récupérer les modifications** des autres, pour vous assurer que vous restez à jour et pour que la gestion des conflits (deux modifications du même fichier) soit plus simple.

IV.7.2.4. Pousser nos modifications

Il ne reste plus que l'étape consistant à **pousser nos modifications** sur le dépôt distant. Celle-ci se fait avec la commande `git push origin master`.

```
1 informaticienzero@DESKTOP-GGDRVV5 MINGW64 /d/Documents/Visual
  Studio 2017/Projects/Discographie (master)
2 $ git push origin master
3 Counting objects: 23, done.
4 Delta compression using up to 8 threads.
5 Compressing objects: 100% (23/23), done.
6 Writing objects: 100% (23/23), 10.25 KiB | 2.56 MiB/s, done.
7 Total 23 (delta 2), reused 0 (delta 0)
8 remote: Resolving deltas: 100% (2/2), done.
9 To https://github.com/informaticienzero/TutorielCpp-
  Discographie.git
10 a496595..d0535ce master -> master
```

Actualisez votre page GitHub et vous verrez que le code est apparu. Félicitations, votre code est maintenant publiquement disponible et accessible à tous. 🍊

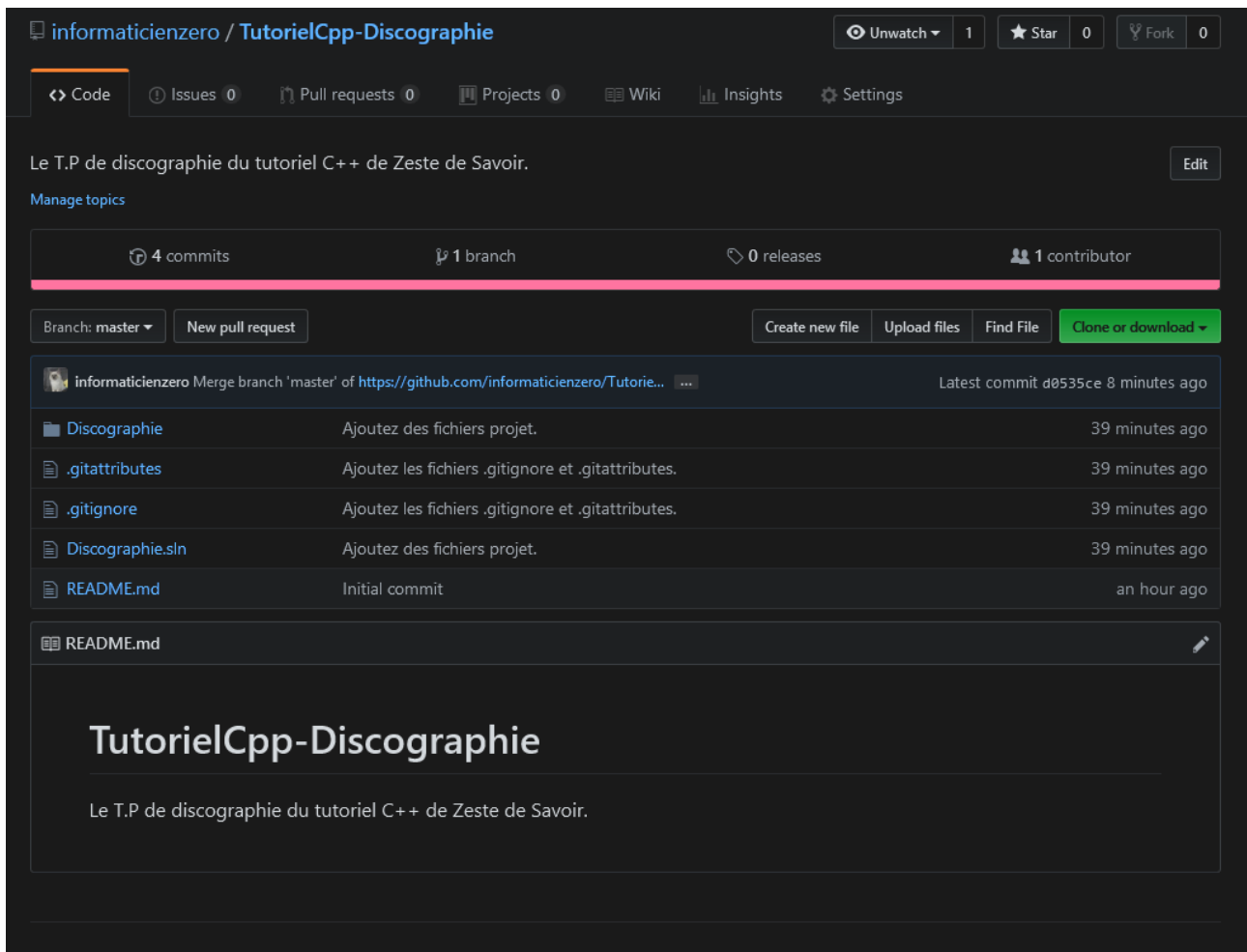


FIGURE IV.7.5. – Ma page GitHub après avoir poussé mes modifications.

IV.7.2.5. Aller plus loin

Nous n'avons fait que découvrir une fonctionnalité, importante certes, mais basique de GitHub. Il vous reste encore beaucoup à apprendre. Comme ce service est très populaire, vous trouverez beaucoup de tutoriels et d'aide sur Internet. Je vous encourage à commencer [par le guide officiel](#) [↗](#).

Dans celui-ci, vous apprendrez notamment à récupérer les projets des autres (on appelle ça *forker*), à leur proposer des modifications (des *pull requests*), à ouvrir et traiter des rapports de *bugs* (des *issues*) et bien d'autres choses encore.

IV.7.3. CMake—Automatiser la compilation de nos programmes

IV.7.3.1. La problématique

Nous avons du code versionné, en ligne pour plus de sécurité et d'accessibilité et c'est déjà très bien. Mais **si je n'ai pas le même environnement?** Dans la partie précédente, le projet posté sur GitHub a été développé avec Visual Studio. Vais-je devoir taper plusieurs lignes de

commandes pour le compiler chez moi? C'est assez galère, avouons-le. Et dans le cas de projets complexes comme Boost, ça peut être difficile de savoir **quelles commandes écrire**.

CMake fait partie de la famille des outils permettant ce qu'on appelle en anglais du ***build automation***, soit de l'automatisation de la compilation. Il est disponible notamment pour **GNU/Linux** et **Windows**, ce qui le rend intéressant, car à partir d'un fichier d'instructions écrit dans un langage propre à CMake, l'outil est capable de générer les instructions de compilation, selon le système d'exploitation et le compilateur utilisés, sans que l'on ait à s'occuper de quoi que ce soit. Il est capable de **compiler d'autres langages que C++**, donc apprendre à l'utiliser vous sera utile même en dehors de ce tutoriel.

IV.7.3.2. Un exemple simple

Reprenons, dans un dossier à part, nos fichiers d'en-têtes et sources qui composent le projet Discographie, puis ajoutons un fichier **CMakeLists.txt**, qui est le fichier expliquant comment compiler notre projet. La première chose à faire est de définir **quelle version minimale de CMake est autorisée**. Cela permet d'imposer l'utilisation d'une version récente, ce qui est nécessaire pour compiler en C++17.

```
1 cmake_minimum_required (VERSION 3.10.0)
```

Ensuite, autre étape importante, **le nom du projet**. Dans notre cas, il est tout trouvé.

```
1 project (Discographie)
```

Maintenant, il faut lister les fichiers qui constituent notre projet, en les séparant entre fichiers d'en-têtes et fichiers sources. CMake utilise des variables pour ça. On va définir, grâce à la fonction **set**, une variable **HEADERS** regroupant tous nos fichiers d'en-têtes et une variable **SOURCES** regroupant les fichiers sources.

```
1 set (HEADERS
2     discographie.hpp
3     donnees_disco.hpp
4     donnees_disco_tests.hpp
5     systeme_commandes.hpp
6     utils.hpp
7 )
8
9 set (SOURCES
10    discographie.cpp
11    donnees_disco.cpp
12    donnees_disco_tests.cpp
13    systeme_commandes.cpp
```

IV. Interlude - Être un développeur

```
14     utils.cpp
15     main.cpp
16 )
```

Terminons en indiquant comment construire une sortie nommée **Discographie**, en utilisant nos variables préalablement définies. La ligne juste au-dessus indique qu'on veut que CMake configure sa sortie pour utiliser C++17.

```
1 set (CMAKE_CXX_STANDARD 17)
2 add_executable (Discographie ${SOURCES} ${HEADERS})
```

Au final, voilà le `CMakeLists.txt` qu'on obtient, avec commentaires en prime.

```
1 # On n'autorise l'utilisation qu'à partir de la version 3.10.0.
2 cmake_minimum_required (VERSION 3.10.0)
3
4 # Il s'agit du projet Discographie.
5 project (Discographie)
6
7 # Les fichiers d'en-tête.
8 set (HEADERS
9     discographie.hpp
10    donnees_disco.hpp
11    donnees_disco_tests.hpp
12    systeme_commandes.hpp
13    utils.hpp
14 )
15
16 # Les fichiers sources.
17 set (SOURCES
18     discographie.cpp
19     donnees_disco.cpp
20     donnees_disco_tests.cpp
21     systeme_commandes.cpp
22     utils.cpp
23     main.cpp
24 )
25
26 # C++17
27 set (CMAKE_CXX_STANDARD 17)
28 # On veut un exécutable Discographie compilé à partir des fichiers
   décrits par SOURCES et HEADERS.
29 add_executable (Discographie ${SOURCES} ${HEADERS})
```

Il ne reste plus qu'à tester en lançant `cmake` avec la commande ci-dessous. Dans mon cas, sous Windows j'obtiens un fichier `.sln`, c'est-à-dire une solution Visual Studio qu'il n'y a plus qu'à

IV. Interlude - Être un développeur

ouvrir et compiler, en 32 *bits*. Pour du 64 *bits*, il faut remplacer `-A Win32` par `-A x64`

```
1 PS D:\Téléchargements\Tests> cmake . -G "Visual Studio 15 2017" -A
  Win32
2 -- Building for: Visual Studio 15 2017
3 -- The C compiler identification is MSVC 19.16.27027.1
4 -- The CXX compiler identification is MSVC 19.16.27027.1
5 -- Check for working C compiler: C:/Programation/Visual Studio
  2017 Community/VC/Tools/MSVC/14.16.27023/bin/Hostx86/x86/cl.exe
6 -- Check for working C compiler: C:/Programation/Visual Studio
  2017 Community/VC/Tools/MSVC/14.16.27023/bin/Hostx86/x86/cl.exe
  -- works
7 -- Detecting C compiler ABI info
8 -- Detecting C compiler ABI info - done
9 -- Detecting C compile features
10 -- Detecting C compile features - done
11 -- Check for working CXX compiler: C:/Programation/Visual Studio
  2017 Community/VC/Tools/MSVC/14.16.27023/bin/Hostx86/x86/cl.exe
12 -- Check for working CXX compiler: C:/Programation/Visual Studio
  2017 Community/VC/Tools/MSVC/14.16.27023/bin/Hostx86/x86/cl.exe
  -- works
13 -- Detecting CXX compiler ABI info
14 -- Detecting CXX compiler ABI info - done
15 -- Detecting CXX compile features
16 -- Detecting CXX compile features - done
17 -- Configuring done
18 -- Generating done
19 -- Build files have been written to: D:/Téléchargements/Tests
```

Sous GNU/Linux, on obtient un fichier Makefile, qu'il n'y a plus qu'à exécuter en lançant la commande `make`, qui va construire l'exécutable.

```
1 informaticienzero@DESKTOP-GGDRVV5:~/Documents/TutorielCpp-
Discographie/Discographie$ cmake -g
  CMakeLists.txt
2 -- The C compiler identification is GNU 7.3.0
3 -- The CXX compiler identification is GNU 7.3.0
4 -- Check for working C compiler: /usr/bin/cc
5 -- Check for working C compiler: /usr/bin/cc -- works
6 -- Detecting C compiler ABI info
7 -- Detecting C compiler ABI info - done
8 -- Detecting C compile features
9 -- Detecting C compile features - done
10 -- Check for working CXX compiler: /usr/bin/c++
11 -- Check for working CXX compiler: /usr/bin/c++ -- works
12 -- Detecting CXX compiler ABI info
13 -- Detecting CXX compiler ABI info - done
```

```
14 -- Detecting CXX compile features
15 -- Detecting CXX compile features - done
16 -- Configuring done
17 -- Generating done
18 -- Build files have been written to:
    /home/informaticienzero/Documents/TutorielCpp-
    Discographie/Discographie
19
20 informaticienzero@DESKTOP-GGDRVV5:~/Documents/TutorielCpp-
    Discographie/Discographie$
    make
21 [ 14%] Building CXX object
    CMakeFiles/Discographie.dir/discographie.cpp.o
22 [ 28%] Building CXX object
    CMakeFiles/Discographie.dir/donnees_disco.cpp.o
23 [ 42%] Building CXX object
    CMakeFiles/Discographie.dir/donnees_disco_tests.cpp.o
24 [ 57%] Building CXX object
    CMakeFiles/Discographie.dir/systeme_commandes.cpp.o
25 [ 71%] Building CXX object CMakeFiles/Discographie.dir/utls.cpp.o
26 [ 85%] Building CXX object CMakeFiles/Discographie.dir/main.cpp.o
27 [100%] Linking CXX executable Discographie
28 [100%] Built target Discographie
```

Dans le cas où vous avez Qt Creator d'installé, nous allons nous baser sur le compilateur fourni avec, MinGW. Pour que CMake puisse l'utiliser, il faut lancer la console incluse avec Qt. Dans mon cas, elle se trouve dans **Menu Démarrer -> Qt -> 5.12.2 -> MinGW 7.3.0 (32 bit) -> Qt 5.12.2 (MinGW 7.3.0 32-bit)**. Le chemin est à adapter en fonction de votre version de Qt, notamment.

Maintenant, on précise à CMake que l'on veut utiliser **"MinGW Makefiles"**, ce qui va produire un fichier Makefile, comme sous l'environnement GNU/Linux.



Pour MinGW

Petite particularité concernant MinGW, il faut d'abord ajouter la ligne suivante dans le fichier **CMakeLists.txt**, avant la ligne **add_executable**.

```
1 set (CMAKE_EXE_LINKER_FLAGS
    "-static-libgcc -static-libstdc++ -static")
```

En sortie, j'obtiens le résultat suivant. Le code est généré en lançant **mingw32-make**.

```
1 C:\Users\informaticienzero\Downloads\Discographie>cmake . -G "MinGW
    Makefiles"
2 -- The C compiler identification is GNU 7.3.0
```

```
3 -- The CXX compiler identification is GNU 7.3.0
4 -- Check for working C compiler:
  C:/Programmation/Qt/Tools/mingw730_32/bin/gcc.exe
5 -- Check for working C compiler:
  C:/Programmation/Qt/Tools/mingw730_32/bin/gcc.exe -- works
6 -- Detecting C compiler ABI info
7 -- Detecting C compiler ABI info - done
8 -- Detecting C compile features
9 -- Detecting C compile features - done
10 -- Check for working CXX compiler:
   C:/Programmation/Qt/Tools/mingw730_32/bin/g++.exe
11 -- Check for working CXX compiler:
   C:/Programmation/Qt/Tools/mingw730_32/bin/g++.exe -- works
12 -- Detecting CXX compiler ABI info
13 -- Detecting CXX compiler ABI info - done
14 -- Detecting CXX compile features
15 -- Detecting CXX compile features - done
16 -- Configuring done
17 -- Generating done
18 -- Build files have been written to:
   C:/Users/benbo/Downloads/Discographie
19
20 C:\Users\informaticienzero\Downloads\Discographie>mingw32-make
21 Scanning dependencies of target Discographie
22 [ 14%] Building CXX object
   CMakeFiles/Discographie.dir/discographie.cpp.obj
23 [ 28%] Building CXX object
   CMakeFiles/Discographie.dir/donnees_disco.cpp.obj
24 [ 42%] Building CXX object
   CMakeFiles/Discographie.dir/donnees_disco_tests.cpp.obj
25 [ 57%] Building CXX object
   CMakeFiles/Discographie.dir/systeme_commandes.cpp.obj
26 [ 71%] Building CXX object
   CMakeFiles/Discographie.dir/utils.cpp.obj
27 [ 85%] Building CXX object CMakeFiles/Discographie.dir/main.cpp.obj
28 [100%] Linking CXX executable Discographie.exe
29 [100%] Built target Discographie
```

IV.7.3.3. Lier des bibliothèques externes

Très bien, nous sommes capables de compiler notre projet peu importe la plateforme. C'est déjà une très bonne base. Mais allons plus loin en intégrant une bibliothèque externe, Boost. Je veux en effet pouvoir utiliser Boost.Log dans mon fichier `main.cpp`.

```
1 #include "donnees_disco_tests.hpp"
2 #include "discographie.hpp"
```

```
3 #include "systeme_commandes.hpp"
4 #include "utils.hpp"
5
6 #include <stdexcept>
7 #include <boost/log/trivial.hpp>
8
9 int main()
10 {
11     // ...
12
13     BOOST_LOG_TRIVIAL(info) << "Lancement du projet Discographie.";
14
15     bool continuer { true };
16     do
17     {
18         // Du code.
19
20     } while (continuer);
21
22     BOOST_LOG_TRIVIAL(info) << "Fin du programme.";
23     return 0;
24 }
```

La première chose qu'il faut prendre en compte, ce sont les **différences entre Windows et GNU/Linux**. Eh oui, Boost n'est pas installé au même endroit selon le système d'exploitation utilisé. Heureusement, CMake permet de **préciser des instructions** pour un système d'exploitation ou un compilateur en particulier.

IV.7.3.3.1. Liaison statique

Commençons par définir une variable bien utile pour la suite, qui indique qu'on veut lier statiquement Boost à notre programme. Cela se fait toujours avec le mot-clé **set**.

```
1 set (Boost_USE_STATIC_LIBS ON)
```

IV.7.3.3.2. GNU/Linux

Ensuite, configurons les spécificités de GNU/Linux. Pour ça, rien de compliqué, un simple **if (UNIX)** suffit. Notez la présence d'un **ENDIF (UNIX)**, qui clôt le bloc conditionnel de la même manière que nos accolades en C++.

```
1 if (UNIX)
2
```

```
3 endif (UNIX)
```

De quoi a-t-on besoin? Il existe un [module](#) qui cherche spécifiquement Boost sur le système où CMake est installé. On l'utilise comme suit, en remplaçant `composants` par les parties de Boost dont nous avons besoin.

```
1 find_package (Boost COMPONENTS composants REQUIRED)
```

Souvenez-vous du chapitre précédent, nous avons besoin non seulement de `log` et `log_setup`, mais également de `system` et de `thread`. La commande à écrire est donc la suivante.

```
1 if (UNIX)
2     find_package (Boost COMPONENTS log log_setup system thread
3         REQUIRED)
3 endif (UNIX)
```

IV.7.3.3.3. Visual Studio

Rien de bien méchant pour la configuration de MSVC, le doux nom du compilateur fourni avec Visual Studio. À noter que seuls `log` et `log_setup` sont requis.

```
1 # Configuration pour Windows avec Visual Studio.
2 if (MSVC)
3     set (BOOST_ROOT "C:/Programmation/Boost.1.69/boost")
4     set (BOOST_INCLUDEDIR "C:/Programmation/Boost.1.69")
5     # Avec Visual Studio, seuls log et log_setup sont requis.
6     find_package (Boost COMPONENTS log log_setup REQUIRED)
7 endif (MSVC)
```

IV.7.3.3.4. Qt Creator

Comme Qt Creator utilise notamment MinGW, on va demander à ce que CMake fasse de même. Et la première chose est de lui indiquer où se trouve l'exécutable `make`, qui permet de produire l'exécutable depuis le fichier `Makefile` généré par CMake. Ce chemin est bien entendu à adapter en fonction de l'endroit où Qt Creator est installé.

```
1 if (MINGW)
2     set (CMAKE_PREFIX_PATH
3         "C:/Programmation/Qt/Tools/mingw730_64/bin")
```

```
3 endif (MINGW)
```

Une autre variable est à créer à cause d'un [bug](#) . Sans cette variable, CMake ne trouvera pas nos bibliothèques 64 *bits*. Bien entendu, cela n'est valable que si vous utilisez la version 64 *bits* de la bibliothèque.

```
1 if (MINGW)
2     set (CMAKE_PREFIX_PATH
3         "C:/Programmation/Qt/Tools/mingw730_64/bin")
4     set (Boost_ARCHITECTURE "-x64")
5 endif (MINGW)
```

Comme pour Visual Studio, il faut préciser le dossier contenant Boost, ainsi que les fichiers d'en-tête, puis appeler la commande `find_package`.

```
1 if (MINGW)
2     set (CMAKE_PREFIX_PATH
3         "C:/Programmation/Qt/Tools/mingw730_64/bin")
4     set (Boost_ARCHITECTURE "-x64")
5     set (BOOST_ROOT "C:/Boost")
6     set (BOOST_INCLUDEDIR "C:/Boost/include/boost-1_69")
7     find_package (Boost COMPONENTS log log_setup thread REQUIRED)
8 endif (MINGW)
```

IV.7.3.3.5. A-t-on trouvé Boost ?

Il est important de gérer le cas où Boost ne serait pas présent ou pas trouvé. Heureusement, c'est très simple. Quand la commande `find_package` est invoquée pour trouver une bibliothèque, elle a défini une variable `XXX_FOUND`, un booléen indiquant si oui ou non la bibliothèque demandée est présente. Dans notre cas, c'est `Boost_FOUND`.

```
1 if (NOT Boost_FOUND)
2
3 else ()
4
5 endif (NOT Boost_FOUND)
```

Dans le cas où Boost est introuvable, il faut afficher un message d'erreur. La commande `message` prend justement un niveau d'erreur ainsi que le message à afficher.


```
1 if (NOT Boost_FOUND)
2     message (FATAL_ERROR "Erreur : Boost est introuvable !")
3 else ()
4
5 endif (NOT Boost_FOUND)
```

Dans le cas contraire, on peut afficher les chemins des fichiers d'en-têtes et objets, par exemple. Cela se fait en affichant `Boost_INCLUDE_DIRS` et `Boost_LIBRARY_DIRS`.

```
1 if (NOT Boost_FOUND)
2     message (FATAL_ERROR "Erreur : Boost est introuvable !")
3 else ()
4
5     message (STATUS
6         "Boost Fichiers en-têtes - ${Boost_INCLUDE_DIRS}")
7     message (STATUS
8         "Boost Fichiers objets - ${Boost_LIBRARY_DIRS}")
9
10 endif (NOT Boost_FOUND)
```

IV.7.3.3.6. Inclure les fichiers nécessaires

Ces deux variables ne sont pas juste là pour être affichées. Pour que notre programme soit correctement généré, il faut qu'on lui indique les fichiers d'en-têtes et objets à inclure. C'est justement ce que font les commandes `include_directories` et `link_directories`.

```
1 if (NOT Boost_FOUND)
2     message (FATAL_ERROR "Erreur : Boost est introuvable !")
3 else ()
4
5     message (STATUS
6         "Boost Fichiers en-têtes - ${Boost_INCLUDE_DIRS}")
7     message (STATUS
8         "Boost Fichiers objets - ${Boost_LIBRARY_DIRS}")
9
10     include_directories (${Boost_INCLUDE_DIRS})
11     link_directories (${Boost_LIBRARY_DIRS})
12
13 endif (NOT Boost_FOUND)
```

IV.7.3.3.7. La construction de l'exécutable

Comme dans l'étape précédente, nous voulons configurer le programme pour qu'il soit compilé avec C++17. Les deux commandes vues plus haut, dans l'exemple simple, ne changent donc pas.

```
1  if (NOT Boost_FOUND)
2      message (FATAL_ERROR "Erreur : Boost est introuvable !")
3  else ()
4
5      message (STATUS
6          "Boost Fichiers en-têtes - ${Boost_INCLUDE_DIRS}")
7      message (STATUS
8          "Boost Fichiers objets - ${Boost_LIBRARY_DIRS}")
9
10     include_directories (${Boost_INCLUDE_DIRS})
11     link_directories (${Boost_LIBRARY_DIRS})
12
13     set (CMAKE_CXX_STANDARD 17)
14     add_executable (Discographie ${SOURCES})
15
16 endif (NOT Boost_FOUND)
```

Il ne reste plus qu'à lier Boost à notre exécutable. Cette étape nécessite elle aussi une séparation entre Windows et GNU/Linux. En effet, la façon d'ajouter les bibliothèques n'est pas la même et GNU/Linux réclame une option supplémentaire.

```
1  if (NOT Boost_FOUND)
2      message (FATAL_ERROR "Erreur : Boost est introuvable !")
3  else ()
4
5      message (STATUS
6          "Boost Fichiers en-têtes - ${Boost_INCLUDE_DIRS}")
7      message (STATUS
8          "Boost Fichiers objets - ${Boost_LIBRARY_DIRS}")
9
10     include_directories (${Boost_INCLUDE_DIRS})
11     link_directories (${Boost_LIBRARY_DIRS})
12
13     set (CMAKE_CXX_STANDARD 17)
14     add_executable (Discographie ${SOURCES})
15
16     if (UNIX)
17
18     else ()
19
20     endif (UNIX)
```

```
19  
20 endif (NOT Boost_FOUND)
```

Dans les deux cas, on utilise la commande `target_link_library`. Pour Windows, c'est simple, il suffit de donner le chemin des fichiers objets, c'est-à-dire `Boost_LIBRARY_DIRS`. Pour GNU/Linux, il faut rechercher une bibliothèque supplémentaire, `-lpthread`, puis préciser quelles bibliothèques sont précisément requises.



Parallèle

Ce qui se passe ici est en fait la même chose que ce que nous faisons manuellement au chapitre précédent.

Je vous mets donc les commandes nécessaires ci-dessous.

```
1  if (NOT Boost_FOUND)  
2      message (FATAL_ERROR "Erreur : Boost est introuvable !")  
3  else ()  
4  
5      message (STATUS  
6          "Boost Fichiers en-têtes - ${Boost_INCLUDE_DIRS}")  
7      message (STATUS  
8          "Boost Fichiers objets - ${Boost_LIBRARY_DIRS}")  
9  
10     include_directories (${Boost_INCLUDE_DIRS})  
11     link_directories (${Boost_LIBRARY_DIRS})  
12  
13     set (CMAKE_CXX_STANDARD 17)  
14     add_executable (Discographie ${SOURCES})  
15  
16     if (UNIX)  
17         set (THREADS_PREFER_PTHREAD_FLAG ON)  
18         find_package (Threads REQUIRED)  
19         target_link_libraries (Discographie -static  
20             Threads::Threads Boost::system Boost::log  
21             Boost::log_setup Boost::thread)  
22     else ()  
23         target_link_libraries (Discographie ${Boost_LIBRARIES})  
24     endif (UNIX)  
25 endif (NOT Boost_FOUND)
```

IV.7.3.3.8. Fichier final

Voici le fichier au final, commenté et complet.

```
1 # On n'autorise l'utilisation qu'à partir de la version 3.0.
2 cmake_minimum_required(VERSION 3.10)
3
4 # Il s'agit du projet Discographie.
5 project (Discographie)
6
7 # Couleurs en sortie pour les Makefiles sur GNU/Linux.
8 set (CMAKE_COLOR_MAKEFILE ON)
9
10 # Les fichiers d'en-tête.
11 set (HEADERS
12     discographie.hpp
13     donnees_disco.hpp
14     donnees_disco_tests.hpp
15     systeme_commandes.hpp
16     utils.hpp
17 )
18
19 # Les fichiers sources.
20 set (SOURCES
21     discographie.cpp
22     donnees_disco.cpp
23     donnees_disco_tests.cpp
24     systeme_commandes.cpp
25     utils.cpp
26     main.cpp
27 )
28
29 # On veut lier Boost de façon statique.
30 set (Boost_USE_STATIC_LIBS ON)
31
32 # Configuration pour GNU/Linux et les UNIX.
33 if (UNIX)
34     set (BOOST_ROOT "/user/include/boost")
35     set (BOOST_INCLUDEDIR "/user/include/boost")
36     # Utiliser Boost.Log réclame plusieurs bibliothèques sous
37     # GNU/Linux.
38     find_package (Boost COMPONENTS log log_setup system thread
39                     REQUIRED)
40 endif (UNIX)
41
42 # Configuration pour Windows avec Visual Studio.
43 if (MSVC)
44     set (BOOST_ROOT "C:/Programmation/Boost.1.69/boost")
45     set (BOOST_INCLUDEDIR "C:/Programmation/Boost.1.69")
46     # Avec Visual Studio, seuls log et log_setup sont requis.
47     find_package (Boost COMPONENTS log log_setup REQUIRED)
48 endif (MSVC)
```

```

48 # Configuration pour Windows avec Qt Creator.
49 if (MINGW)
50     # Cas particulier, pour indiquer où se trouve make.
51     set (CMAKE_PREFIX_PATH
52         "C:/Programmation/Qt/Tools/mingw730_64/bin")
53     # Pour utiliser les bibliothèques 64 bits.
54     set (Boost_ARCHITECTURE "-x64")
55
56     set (BOOST_ROOT "C:/Boost")
57     set (BOOST_INCLUDEDIR "C:/Boost/include/boost-1_69")
58     # Avec Qt Creator, il faut aussi ajouter thread.
59     find_package (Boost COMPONENTS log log_setup thread REQUIRED)
60 endif (MINGW)
61
62 # Teste si Boost a bien été trouvé.
63 if (NOT Boost_FOUND)
64     message (FATAL_ERROR "Erreur : Boost est introuvable !")
65 else ()
66     message (STATUS
67         "Boost Fichiers en-têtes - ${Boost_INCLUDE_DIRS}")
68     message (STATUS
69         "Boost Fichiers objets - ${Boost_LIBRARY_DIRS}")
70
71     include_directories (${Boost_INCLUDE_DIRS})
72     link_directories (${Boost_LIBRARY_DIRS})
73
74     # C++17
75     set (CMAKE_CXX_STANDARD 17)
76     # On veut un exécutable Discographie compilé à partir des
77     # fichiers décrits par SOURCES et HEADERS.
78     add_executable (Discographie ${SOURCES})
79
80     # Les options pour lier l'exécutable et Boost sont différentes
81     # selon Windows ou GNU/Linux.
82     if (UNIX)
83         set (THREADS_PREFER_PTHREAD_FLAG ON)
84         # Requis pour GNU/Linux.
85         find_package (Threads REQUIRED)
86         target_link_libraries (Discographie -static
87             Threads::Threads Boost::system Boost::log
88             Boost::log_setup Boost::thread)
89     else ()
90         target_link_libraries (Discographie ${Boost_LIBRARIES})
91     endif (UNIX)
92 endif (NOT Boost_FOUND)

```

IV.7.3.3.9. Exécuter CMake

Maintenant que le fichier `CMakeLists.txt` est prêt, alors il ne nous reste plus qu'à lancer la génération de notre programme. Voici comment je construis le programme avec Boost 32 *bits*. J'obtiens une solution `.sln` prête à être compilée et exécutée.

```
1 PS D:\Téléchargements\Tests> cmake . -G "Visual Studio 15 2017" -A
  Win32
2 -- The C compiler identification is MSVC 19.16.27027.1
3 -- The CXX compiler identification is MSVC 19.16.27027.1
4 -- Check for working C compiler: C:/Programation/Visual Studio
  2017 Community/VC/Tools/MSVC/14.16.27023/bin/Hostx86/x86/cl.exe
5 -- Check for working C compiler: C:/Programation/Visual Studio
  2017 Community/VC/Tools/MSVC/14.16.27023/bin/Hostx86/x86/cl.exe
  -- works
6 -- Detecting C compiler ABI info
7 -- Detecting C compiler ABI info - done
8 -- Detecting C compile features
9 -- Detecting C compile features - done
10 -- Check for working CXX compiler: C:/Programation/Visual Studio
  2017 Community/VC/Tools/MSVC/14.16.27023/bin/Hostx86/x86/cl.exe
11 -- Check for working CXX compiler: C:/Programation/Visual Studio
  2017 Community/VC/Tools/MSVC/14.16.27023/bin/Hostx86/x86/cl.exe
  -- works
12 -- Detecting CXX compiler ABI info
13 -- Detecting CXX compiler ABI info - done
14 -- Detecting CXX compile features
15 -- Detecting CXX compile features - done
16 -- Looking for pthread.h
17 -- Looking for pthread.h - not found
18 -- Found Threads: TRUE
19 -- Boost version: 1.69.0
20 -- Found the following Boost libraries:
21 --   log
22 --   log_setup
23 --   date_time
24 --   filesystem
25 --   thread
26 --   regex
27 --   chrono
28 --   atomic
29 -- Boost Fichier en-têtes - C:/Programation/Boost.1.69
30 -- Boost Fichiers objets -
  C:/Programation/Boost.1.69/lib32-msvc-14.1
31 -- Configuring done
32 -- Generating done
33 -- Build files have been written to: D:/Téléchargements/Tests
```

Sous GNU/Linux, j'obtiens un fichier `Makefile` qu'il ne reste plus qu'à lancer en appelant

IV. Interlude - Être un développeur

`make`. J'obtiens au final mon programme fonctionnel.

```
1  informaticienzero@DESKTOP-GGDRVV5:~/Documents/TutorielCpp-  
    Discographie/Discographie$ cmake -g  
    CMakeLists.txt  
2  -- The C compiler identification is GNU 7.3.0  
3  -- The CXX compiler identification is GNU 7.3.0  
4  -- Check for working C compiler: /usr/bin/cc  
5  -- Check for working C compiler: /usr/bin/cc -- works  
6  -- Detecting C compiler ABI info  
7  -- Detecting C compiler ABI info - done  
8  -- Detecting C compile features  
9  -- Detecting C compile features - done  
10 -- Check for working CXX compiler: /usr/bin/c++  
11 -- Check for working CXX compiler: /usr/bin/c++ -- works  
12 -- Detecting CXX compiler ABI info  
13 -- Detecting CXX compiler ABI info - done  
14 -- Detecting CXX compile features  
15 -- Detecting CXX compile features - done  
16 -- Looking for pthread.h  
17 -- Looking for pthread.h - found  
18 -- Looking for pthread_create  
19 -- Looking for pthread_create - not found  
20 -- Looking for pthread_create in pthreads  
21 -- Looking for pthread_create in pthreads - not found  
22 -- Looking for pthread_create in pthread  
23 -- Looking for pthread_create in pthread - found  
24 -- Found Threads: TRUE  
25 -- Boost version: 1.65.1  
26 -- Found the following Boost libraries:  
27 --   log  
28 --   log_setup  
29 --   system  
30 --   thread  
31 --   date_time  
32 --   filesystem  
33 --   regex  
34 --   chrono  
35 --   atomic  
36 -- Boost Fichier en-têtes - /usr/include  
37 -- Boost Fichiers objets - /usr/lib/x86_64-linux-gnu  
38 -- Check if compiler accepts -pthread  
39 -- Check if compiler accepts -pthread - yes  
40 -- Configuring done  
41 -- Generating done  
42 -- Build files have been written to:  
    /home/informaticienzero/Documents/TutorielCpp-  
    Discographie/Discographie
```

```
43 informaticienzero@DESKTOP-GGDRVV5:~/Documents/TutorielCpp-  
Discographie/Discographie$  
    make  
44 Scanning dependencies of target Discographie  
45 [ 14%] Building CXX object  
    CMakeFiles/Discographie.dir/discographie.cpp.o  
46 [ 28%] Building CXX object  
    CMakeFiles/Discographie.dir/donnees_disco.cpp.o  
47 [ 42%] Building CXX object  
    CMakeFiles/Discographie.dir/donnees_disco_tests.cpp.o  
48 [ 57%] Building CXX object  
    CMakeFiles/Discographie.dir/systeme_commandes.cpp.o  
49 [ 71%] Building CXX object CMakeFiles/Discographie.dir/utls.cpp.o  
50 [ 85%] Building CXX object CMakeFiles/Discographie.dir/main.cpp.o  
51 [100%] Linking CXX executable Discographie  
52 [100%] Built target Discographie  
53 informaticienzero@DESKTOP-GGDRVV5:~/Documents/TutorielCpp-  
Discographie/Discographie$  
    ./Discographie  
54 [2019-03-26 16:22:56.944143] [0x000000000220c900] [info]  
    Lancement du programme Discothèque.  
55 > quitter  
56 [2019-03-26 16:23:02.046525] [0x00001d30] [info]      Fin du  
    programme.
```

Avec MinGW, pareil, je relance une console Qt et j'exécute la commande `cmake` avec les bons paramètres.

```
1 C:\Users\informaticienzero\Downloads\Discographie>cmake . -G "MinGW  
  Makefiles"  
2 -- The C compiler identification is GNU 7.3.0  
3 -- The CXX compiler identification is GNU 7.3.0  
4 -- Check for working C compiler:  
    C:/Programmation/Qt/Tools/mingw730_64/bin/gcc.exe  
5 -- Check for working C compiler:  
    C:/Programmation/Qt/Tools/mingw730_64/bin/gcc.exe -- works  
6 -- Detecting C compiler ABI info  
7 -- Detecting C compiler ABI info - done  
8 -- Detecting C compile features  
9 -- Detecting C compile features - done  
10 -- Check for working CXX compiler:  
    C:/Programmation/Qt/Tools/mingw730_64/bin/g++.exe  
11 -- Check for working CXX compiler:  
    C:/Programmation/Qt/Tools/mingw730_64/bin/g++.exe -- works  
12 -- Detecting CXX compiler ABI info  
13 -- Detecting CXX compiler ABI info - done  
14 -- Detecting CXX compile features  
15 -- Detecting CXX compile features - done
```



```
16 -- Looking for pthread.h
17 -- Looking for pthread.h - found
18 -- Looking for pthread_create
19 -- Looking for pthread_create - found
20 -- Found Threads: TRUE
21 -- Boost version: 1.69.0
22 -- Found the following Boost libraries:
23 --   log
24 --   log_setup
25 --   thread
26 --   date_time
27 --   filesystem
28 --   regex
29 --   chrono
30 --   atomic
31 -- Boost Fichier en-têtes - C:/Boost/include/boost-1_69
32 -- Boost Fichiers objets - C:/Boost/lib
33 -- Configuring done
34 -- Generating done
35 -- Build files have been written to:
    C:/Users/benbo/Downloads/Discographie
36
37 C:\Users\informaticienzero\Downloads\Discographie>mingw32-make
38 Scanning dependencies of target Discographie
39 [ 14%] Building CXX object
    CMakeFiles/Discographie.dir/discographie.cpp.obj
40 [ 28%] Building CXX object
    CMakeFiles/Discographie.dir/donnees_disco.cpp.obj
41 [ 42%] Building CXX object
    CMakeFiles/Discographie.dir/donnees_disco_tests.cpp.obj
42 [ 57%] Building CXX object
    CMakeFiles/Discographie.dir/systeme_commandes.cpp.obj
43 [ 71%] Building CXX object
    CMakeFiles/Discographie.dir/utils.cpp.obj
44 [ 85%] Building CXX object CMakeFiles/Discographie.dir/main.cpp.obj
45 [100%] Linking CXX executable Discographie.exe
46 [100%] Built target Discographie
47
48
49 C:\Users\informaticienzero\Downloads\Discographie>Discographie.exe
50 [2019-03-26 17:12:07.500987] [0x00001d30] [info]    Lancement du
    projet Discographie.
51 > quitter
52 [2019-03-26 17:12:12.046525] [0x00001d30] [info]    Fin du
    programme.
```

IV.7.3.4. Définir des variables au lancement

Typiquement, pour la partie MinGW, j'ai défini en dur la variable `Boost_ARCHITECTURE` comme valant `"-x64"`. Mais si jamais on veut compiler en 32 *bits*? Le mieux serait de ne pas avoir cette variable en dur, mais plutôt de **la définir quand on en a besoin**. C'est justement ce que nous permet de faire CMake.

```
1 cmake . -G "MinGW Makefiles" -DBoost_ARCHITECTURE="-x64"
```

L'option `-DNOM` permet de définir une variable `NOM` avec la valeur de notre choix. On peut ainsi préciser explicitement `"-x64"` quand on le veut, sans être forcé comme avant.

Le principe est le même pour les autres valeurs en dur, qui peuvent changer en fonction de l'environnement d'exécution, comme le chemin de Boost notamment. Je vous invite à définir ce genre de variable au lancement pour que n'importe quel utilisateur puisse **personnaliser l'exécution**.

IV.7.3.5. Aller plus loin

CMake est un outil puissant, qui permet de gagner en souplesse, mais le maîtriser demande bien plus que de savoir les quelques commandes que nous avons vues ici. De nombreuses ressources sont disponibles, à commencer par [le cours officiel](#) sur le site de CMake. Vous pouvez également jeter un œil à ce [tutoriel en français](#).

Par exemple, il est possible de **lancer automatiquement Doxygen** en même temps que la compilation, afin de disposer également de la documentation du projet. Vous pouvez aussi apprendre des techniques pour ne pas avoir à **mettre à jour à la main la liste des fichiers d'en-têtes et sources**. Mais tout ça, c'est de votre ressort maintenant. 🍏


IV.7.4. Aller plus loin

Si je devais vous présenter tous les outils utiles à un développeur C++, ce tutoriel serait deux fois plus gros minimum et ce chapitre bien indigeste. Nous en avons déjà vu trois, importants et très largement répandus. Mais il en existe encore **de nombreux autres**, que vous serez amenés à utiliser (ou pas) au fur et à mesure que vous progresserez.

IV.7.4.1. TravisCI— De l'intégration continue

Imaginez qu'à chaque *commit* sur GitHub, votre code soit automatiquement compilé, les tests unitaires lancés et la documentation automatiquement générée et mise à jour? C'est ce que permet de faire [TravisCI](#) et c'est ce qui s'appelle **l'intégration continue**. En plus, l'outil est **gratuit** si votre projet est open-source.

IV.7.4.2. CppCheck—Vérification du code

Un autre outil qui rend de fiers services, j'ai nommé [CppCheck](#) . Cet outil effectue de **nombreuses vérifications** sur votre code pour vous aider à le rendre encore plus robuste. On peut citer notamment la vérification des limites de tableaux, les divisions par zéro, ou encore la vérification des exceptions lancées.

```


1  informaticienzero@DESKTOP-GGDRVV5:~/Documents/TutorielCpp-
   Discographie/Discographie$ cppcheck
   ./
2  Checking CMakeFiles/3.10.2/CompilerIdC/CMakeCCompilerId.c ...
3  Checking CMakeFiles/3.10.2/CompilerIdC/CMakeCCompilerId.c:
   MPRAS;_MPRAS...
4  Checking CMakeFiles/3.10.2/CompilerIdC/CMakeCCompilerId.c:
   SCO_SV;_SCO_SV;sco_sv...
5  Checking CMakeFiles/3.10.2/CompilerIdC/CMakeCCompilerId.c:
   SDCC;__SDCC_VERSION_MAJOR...
6  Checking CMakeFiles/3.10.2/CompilerIdC/CMakeCCompilerId.c:
   SDCC;__SDCC_VERSION_MAJOR;__SDCC_VERSION_MAJOR...
7  Checking CMakeFiles/3.10.2/CompilerIdC/CMakeCCompilerId.c:
   SIMULATE_VERSION_PATCH...
8  Checking CMakeFiles/3.10.2/CompilerIdC/CMakeCCompilerId.c:
   SIMULATE_VERSION_PATCH;SIMULATE_VERSION_TWEAK...
9  Checking CMakeFiles/3.10.2/CompilerIdC/CMakeCCompilerId.c:
   WIN32;_WIN32;__WIN32__...
10 Checking CMakeFiles/3.10.2/CompilerIdC/CMakeCCompilerId.c:
   XENIX;_XENIX;__XENIX__...
11 Checking CMakeFiles/3.10.2/CompilerIdC/CMakeCCompilerId.c:
   _AIX;__AIX;__AIX__;__aix;__aix__...
12 Checking CMakeFiles/3.10.2/CompilerIdC/CMakeCCompilerId.c:
   _BEOS;__BEOS__;__BeOS...
13 Checking CMakeFiles/3.10.2/CompilerIdC/CMakeCCompilerId.c:
   _COMPILER_VERSION;_SGI_COMPILER_VERSION...
14 1/10 files checked 10% done
15 Checking CMakeFiles/3.10.2/CompilerIdCXX/CMakeCXXCompilerId.cpp ...
16 Checking CMakeFiles/3.10.2/CompilerIdCXX/CMakeCXXCompilerId.cpp:
   MPRAS;_MPRAS...
17 Checking CMakeFiles/3.10.2/CompilerIdCXX/CMakeCXXCompilerId.cpp:
   SCO_SV;_SCO_SV;sco_sv...
18 Checking CMakeFiles/3.10.2/CompilerIdCXX/CMakeCXXCompilerId.cpp:
   SIMULATE_VERSION_PATCH...
19 Checking CMakeFiles/3.10.2/CompilerIdCXX/CMakeCXXCompilerId.cpp:
   SIMULATE_VERSION_PATCH;SIMULATE_VERSION_TWEAK...
20 Checking CMakeFiles/3.10.2/CompilerIdCXX/CMakeCXXCompilerId.cpp:
   WIN32;_WIN32;__WIN32__...
21 Checking CMakeFiles/3.10.2/CompilerIdCXX/CMakeCXXCompilerId.cpp:
   XENIX;_XENIX;__XENIX__...
22 Checking CMakeFiles/3.10.2/CompilerIdCXX/CMakeCXXCompilerId.cpp:
   _AIX;__AIX;__AIX__;__aix;__aix__...

```

```
23 Checking CMakeFiles/3.10.2/CompilerIdCXX/CMakeCXXCompilerId.cpp:
    _BEOS;__BEOS__;__BeOS...
24 Checking CMakeFiles/3.10.2/CompilerIdCXX/CMakeCXXCompilerId.cpp:
    _COMPILER_VERSION;_SGI_COMPILER_VERSION...
25 Checking CMakeFiles/3.10.2/CompilerIdCXX/CMakeCXXCompilerId.cpp:
    _COMPILER_VERSION;_SGI_COMPILER_VERSION;_SGI_COMPILER_VERSION...
26 Checking CMakeFiles/3.10.2/CompilerIdCXX/CMakeCXXCompilerId.cpp:
    _CRAYC...
27 2/10 files checked 20% done
28 Checking CMakeFiles/feature_tests.c ...
29 Checking CMakeFiles/feature_tests.c: __STDC_VERSION__...
30 3/10 files checked 30% done
31 Checking CMakeFiles/feature_tests.cxx ...
32 Checking CMakeFiles/feature_tests.cxx:
    __GXX_EXPERIMENTAL_CXX0X__...
33 4/10 files checked 40% done
34 Checking discographie.cpp ...
35 5/10 files checked 50% done
36 Checking donnees_disco.cpp ...
37 6/10 files checked 60% done
38 Checking donnees_disco_tests.cpp ...
39 7/10 files checked 70% done
40 Checking main.cpp ...
41 8/10 files checked 80% done
42 Checking systeme_commandes.cpp ...
43 9/10 files checked 90% done
44 Checking utils.cpp ...
45 10/10 files checked 100% done
```

Beaucoup d'informations, mais aucun problème trouvé sur le projet Discographie. Ouf! 🍊

IV.7.4.3. StackOverflow— La réponses à quasiment toutes les questions

S'il y a un site que tous les développeurs connaissent et lisent, c'est bien [StackOverflow](#) . C'est **LE site de questions-réponses** sur Internet. Si vous n'avez aucune réponse à votre problème sur StackOverflow, alors vous n'avez vraiment pas de chance du tout. Peu importe le langage, peu importe la technologie, tout est abordé sur ce site. Mettez-le dès maintenant en favori, il va vous sauver la vie!

IV.7.4.4. En résumé

- Le gestionnaire de code source git permet de sauvegarder et versionner son code source. Il permet ainsi de tracer un historique du projet, de revenir à des versions antérieures et de développer plusieurs fonctionnalités en parallèle.

IV. Interlude - Être un développeur

- Le service en ligne GitHub permet de partager son code source aux autres, permettant ainsi une sauvegarde pérenne, la création de *bugs*, la proposition de correctifs ou d'améliorations, entre autres choses.
- CMake automatise la génération du code, permettant, avec un unique fichier de configuration, de générer des projets Visual Studio, des *Makefiles* et bien d'autres choses encore.
- Chacun de ces outils offre de nombreuses possibilités que ce cours ne peut pas aborder et qu'il convient au lecteur de découvrir par lui-même.
- De nombreux outils pour C++ sont disponibles, qu'un développeur découvre et utilise au fil du temps.

Finalement, être un développeur, ça recouvre bien plus que simplement écrire du code. Tous ces outils sont là pour nous faire gagner du temps et de l'efficacité. Bien entendu, tous les développeurs C++ n'utilisent pas tous les mêmes outils. Certains vous seront peut-être imposés, en fonction du projet sur lequel vous travaillerez. Mais certains principes restent universels.

- La documentation est votre meilleure amie.
- Une bonne documentation épargnera beaucoup de peine à ceux qui passeront derrière vous ou qui utiliseront vos produits.
- On crée tout le temps des *bugs*, alors apprendre à les détecter et les éliminer est une bonne habitude. On essaiera bien entendu d'en éliminer un maximum dès la compilation, mais pour ceux qui restent, le *débogueur* vous attend.
- Comprendre la compilation est utile, parce qu'en comprenant mieux ce qui se passe sous le capot, vous pouvez optimiser (comme avec `constexpr`) et mieux comprendre les erreurs (erreur de *linkage* par exemple).

Le fait est qu'en terminant cette partie, vous avez bien progressé. Vous commencez à avoir la mentalité et les pratiques d'un bon développeur. Nous allons justement mettre l'ensemble du cours en pratique avec une étude de cas.

Contenu masqué

Contenu masqué n°72 :

Fichier .gitignore pour Visual Studio

```
1 ## Ignore Visual Studio temporary files, build results, and
2 ## files generated by popular Visual Studio add-ons.
3
4 # User-specific files
5 *.suo
6 *.user
7 *.useroscach
8 *.sln.docstates
9
```

```
10 # User-specific files (MonoDevelop/Xamarin Studio)
11 *.userprefs
12
13 # Build results
14 [Dd]ebug/
15 [Dd]ebugPublic/
16 [Rr]elease/
17 [Rr]eleases/
18 x64/
19 x86/
20 bld/
21 [Bb]in/
22 [Oo]bj/
23 [Ll]og/
24
25 # Visual Studio 2015 cache/options directory
26 .vs/
27 # Uncomment if you have tasks that create the project's static
    files in wwwroot
28 #wwwroot/
29
30 # MSTest test Results
31 [Tt]est[Rr]esult*/
32 [Bb]uild[Ll]og.*
33
34 # NUNIT
35 *.VisualState.xml
36 TestResult.xml
37
38 # Build Results of an ATL Project
39 [Dd]ebugPS/
40 [Rr]eleasePS/
41 dlldata.c
42
43 # DNX
44 project.lock.json
45 project.fragment.lock.json
46 artifacts/
47
48 *_i.c
49 *_p.c
50 *_i.h
51 *.ilk
52 *.meta
53 *.obj
54 *.pch
55 *.pdb
56 *.pgc
57 *.pgd
58 *.rsp
```

```
59 *.sbr
60 *.tlb
61 *.tli
62 *.tlh
63 *.tmp
64 *.tmp_proj
65 *.log
66 *.vsp SCC
67 *.vssscc
68 .builds
69 *.pidb
70 *.svcllog
71 *.scc
72
73 # Chutzpah Test files
74 _Chutzpah*
75
76 # Visual C++ cache files
77 ipch/
78 *.aps
79 *.ncb
80 *.opendb
81 *.opensdf
82 *.sdf
83 *.cache file
84 *.VC.db
85 *.VC.VC.opendb
86
87 # Visual Studio profiler
88 *.psess
89 *.vsp
90 *.vspx
91 *.sap
92
93 # TFS 2012 Local Workspace
94 $tf/
95
96 # Guidance Automation Toolkit
97 *.gpState
98
99 # ReSharper is a .NET coding add-in
100 _ReSharper*/
101 *.[Rr]e[Ss]harper
102 *.DotSettings.user
103
104 # JustCode is a .NET coding add-in
105 .JustCode
106
107 # TeamCity is a build add-in
108 _TeamCity*
```

```
109
110 # DotCover is a Code Coverage Tool
111 *.dotCover
112
113 # NCrunch
114 _NCrunch_*
115 *.crunch*.local.xml
116 nCrunchTemp_*
117
118 # MightyMoose
119 *.mm.*
120 AutoTest.Net/
121
122 # Web workbench (sass)
123 .sass-cache/
124
125 # Installshield output folder
126 [Ee]xpress/
127
128 # DocProject is a documentation generator add-in
129 DocProject/buildhelp/
130 DocProject/Help/*.HxT
131 DocProject/Help/*.HxC
132 DocProject/Help/*.hhc
133 DocProject/Help/*.hhk
134 DocProject/Help/*.hhp
135 DocProject/Help/Html2
136 DocProject/Help/html
137
138 # Click-Once directory
139 publish/
140
141 # Publish Web Output
142 *.[Pp]ublish.xml
143 *.azurePubxml
144 # TODO: Comment the next line if you want to checkin your web
145 #       deploy settings
146 # but database connection strings (with potential passwords) will
147 #       be unencrypted
148 #*.pubxml
149 *.publishproj
150
151 # Microsoft Azure Web App publish settings. Comment the next line
152 #       if you want to
153 # checkin your Azure Web App publish settings, but sensitive
154 #       information contained
155 # in these scripts will be unencrypted
156 PublishScripts/
157
158 # NuGet Packages
```



```
155 *.nupkg
156 # The packages folder can be ignored because of Package Restore
157 **/packages/*
158 # except build/, which is used as an MSBuild target.
159 !**/packages/build/
160 # Uncomment if necessary however generally it will be regenerated
    when needed
161 #!**/packages/repositories.config
162 # NuGet v3's project.json files produces more ignoreable files
163 *.nuget.props
164 *.nuget.targets
165
166 # Microsoft Azure Build Output
167 csx/
168 *.build.csdef
169
170 # Microsoft Azure Emulator
171 ecf/
172 rcf/
173
174 # Windows Store app package directories and files
175 AppPackages/
176 BundleArtifacts/
177 Package.StoreAssociation.xml
178 _pkginfo.txt
179
180 # Visual Studio cache files
181 # files ending in .cache can be ignored
182 *.[Cc]ache
183 # but keep track of directories ending in .cache
184 !*.[Cc]ache/
185
186 # Others
187 ClientBin/
188 ~$*
189 *~
190 *.dbmdl
191 *.dbproj.schemaview
192 *.jfm
193 *.pfx
194 *.publishsettings
195 node_modules/
196 orleans.codegen.cs
197
198 # Since there are multiple workflows, uncomment next line to ignore
    bower_components
199 # (https://github.com/github/gitignore/pull/1529#issuecomment-104372622)
200 #bower_components/
201
```

```
202 # RIA/Silverlight projects
203 Generated_Code/
204
205 # Backup & report files from converting an old project file
206 # to a newer Visual Studio version. Backup files are not needed,
207 # because we have git ;- )
208 _UpgradeReport_Files/
209 Backup*/
210 UpgradeLog*.XML
211 UpgradeLog*.htm
212
213 # SQL Server files
214 *.mdf
215 *.ldf
216
217 # Business Intelligence projects
218 *.rdl.data
219 *.bim.layout
220 *.bim_*.settings
221
222 # Microsoft Fakes
223 FakesAssemblies/
224
225 # GhostDoc plugin setting file
226 *.GhostDoc.xml
227
228 # Node.js Tools for Visual Studio
229 .ntvs_analysis.dat
230
231 # Visual Studio 6 build log
232 *.plg
233
234 # Visual Studio 6 workspace options file
235 *.opt
236
237 # Visual Studio LightSwitch build output
238 **/*.HTMLClient/GeneratedArtifacts
239 **/*.DesktopClient/GeneratedArtifacts
240 **/*.DesktopClient/ModelManifest.xml
241 **/*.Server/GeneratedArtifacts
242 **/*.Server/ModelManifest.xml
243 _Pvt_Extensions
244
245 # Paket dependency manager
246 .paket/paket.exe
247 paket-files/
248
249 # FAKE - F# Make
250 .fake/
251
```

```
252 # JetBrains Rider
253 .idea/
254 *.sln.iml
255
256 # CodeRush
257 .cr/
258
259 # Python Tools for Visual Studio (PTVS)
260 __pycache__/_
261 *.pyc
```

[Retourner au texte.](#)

Contenu masqué n°73 : Fichier .gitignore pour GNU/Linux

```
1 # Prerequisites
2 *.d
3
4 # Compiled Object files
5 *.slo
6 *.lo
7 *.o
8 *.obj
9
10 # Precompiled Headers
11 *.gch
12 *.pch
13
14 # Compiled Dynamic libraries
15 *.so
16 *.dylib
17 *.dll
18
19 # Fortran module files
20 *.mod
21 *.smod
22
23 # Compiled Static libraries
24 *.lai
25 *.la
26 *.a
27 *.lib
28
29 # Executables
30 *.exe
```

IV. Interlude - Être un développeur

```
31 *.out  
32 *.app
```

[Retourner au texte.](#)

Cinquième partie

La Programmation Orientée Objet

V. La Programmation Orientée Objet

Lors de ce cours, nous avons vu que le C++ est un langage très riche qui intègre plusieurs manières de programmer, plusieurs *paradigmes*. Nous avons notamment vu les paradigmes impératif et générique.

Maintenant, nous allons en découvrir un troisième: la **Programmation Orientée Objet**, ou POO pour les intimes. C'est un paradigme puissant qui, combiné avec ceux que l'on a déjà étudiés, permet une conception d'un tout autre niveau en termes d'élégance, d'expressivité, de robustesse. En bref, **un code de qualité supérieure**.

i

Difficulté

Nous allons passer à un niveau de difficulté supérieur. Beaucoup de notions vont arriver d'un coup, donc ne vous en faites pas si vous avez du mal à en comprendre certaines, c'est normal. Reprenez les explications autant que besoin, posez des questions, faites des tests et vous y arriverez. 🍊

V.1. Premiers pas avec la POO

La programmation orientée objet. C'est un terme qu'on rencontre souvent en programmation. Sa définition exacte et son utilisation varient en fonction des points de vue, des langages qui l'utilisent, des contraintes. Une chose néanmoins est sûre, c'est qu'il s'agit d'un paradigme **très répandu, très utilisé** et trop souvent **mal enseigné**.

Ce chapitre a pour but de **vous introduire à la programmation orientée objet**, en définissant les principes qui la régissent et en expliquant ce qu'elle peut nous apporter.

V.1.1. Le principe : des objets bien serviables

V.1.1.1. Le principe

V.2. Penser en terme de données...

Depuis que nous avons commencé ce cours, nous avons pensé et codé en termes de **structures de données** et d'**algorithmes**. Nous imaginions notre code comme un suite d'opérations, avec tel algorithme appliqué à telle structure ou tel type, afin d'obtenir telle information en sortie. Le code se concevait et se découpait comme une suite d'instructions et d'opérations.

Prenons un exemple tiré de la vie réelle et parlons d'une laverie (merci à @lmghs pour cet exemple). Dans celle-ci, on met à la disposition du client des machines à laver, mais **reste à sa charge de faire le reste**. C'est à lui de préparer son linge, d'amener sa lessive, de trier par couleur, de vérifier le poids inséré dans la machine, de lancer celle-ci avec le programme adapté au type de vêtement, de les essorer, etc.

On est typiquement dans le cas d'une solution assez complexe, avec plusieurs opérations à effectuer les unes à la suite des autres, des traitements particuliers (tri du linge, tri par couleur, etc). Tout tourne autour du linge, la «donnée centrale» de notre programme de laverie. Si on devait coder ça, on aurait sans doute une structure pour le linge et plusieurs fonctions comme `tri_linge_par_couleur`, `ajouter_lessive`, etc.

V.2.0.0.1. ...ou de services ?

La programmation orientée objet elle se base sur le principe du **service**. Plutôt que de fournir une liste de fonctions, d'opérations, d'algorithmes à appliquer, on va **abstraire cette complexité technique**, la cacher. On va enfermer, camoufler tout ce qui est complexe et n'offrir qu'une **interface** beaucoup plus simple à utiliser.

Appliquer ce principe à notre laverie revient à créer **un pressing**. Dans ce cas, toute la complexité de tri par couleur, de bonne quantité de lessive et autres est cachée, déléguée. L'utilisation est beaucoup plus simple. Il suffit de venir, déposer le linge sale et repartir avec du linge propre. Nous n'avons rien d'autre à faire qu'attendre (et accessoirement, payer). Le pressing s'occupe de nous rendre un service «Laver votre linge» et nous nous contentons de l'utiliser, sans chercher plus loin.

On n'a pas besoin de maîtriser les arcanes du nettoyage, des gens/spécialistes en qui on peut avoir confiance vont en assumer la responsabilité. On fait une **abstraction** et ce principe est une des pierres angulaires de la programmation orientée objet.



En résumé

La programmation orientée objet, c'est créer des services.

V.2.0.1. Exemple concret

Vous voulez un exemple concret, avec du code? Ça tombe bien, la bibliothèque standard en est remplie! Prenez le cas des **chaînes de caractères**. Plusieurs fois dans le cours, j'ai fait remarquer qu'il existe en C++ un type de chaînes de caractères hérité du C. Dans ce langage, manipuler les chaînes est plus compliqué. Il faut passer par des fonctions pour les concaténer, pour supprimer un caractère, bref, du paradigme impératif typique.

Le type `std::string` permet quand à lui de manipuler les chaînes de caractères de façon beaucoup plus aisée. C'est la raison pour laquelle on l'utilise tout le temps. Sauf qu'en interne, `std::string` manipule des chaînes de caractères version C. Simplement voilà, la tambouille qu'il fait pour concaténer deux chaînes ou en vider une troisième ne nous intéresse pas et n'est justement **pas exposée**. C'est donc un très bon exemple de service.

V.2.1. Un peu de vocabulaire

V.2.1.0.1. L'objet

Un objet n'est ni plus ni moins qu'une **donnée qu'on manipule**. Ainsi, dans le code `std::string chaine`, `chaine` n'est ni plus ni moins qu'un objet de type `std::string`. De même pour `std::vector<int> tableau`, où notre variable `tableau` est un objet de type `std::vector<int>`.

V.2.1.0.2. Le type

Afin de définir quels services sont rendus par un objet, il faut définir son modèle, son type. On parle de **la classe d'un objet**. À partir de cette classe, on pourra créer autant d'objets de ce type qu'on veut. On parle d'**instancier la classe**. Quand on écrit `std::string chaine`, on crée une instance du type `std::string`, instance qui se nomme `chaine` et qui est notre objet.



Point vocabulaire

Des types très basiques, comme `int`, `double` ou `bool`, peuvent être considérés comme des objets d'un point de vue théorique. En pratique, en C++, on fera la distinction entre ces types dits **natifs** et les instances de classes, comme `std::string`.

V.2.1.0.3. L'interface d'une classe

Un objet peut nous rendre plus ou moins de services, en fonction de son type. Dans le cas de `std::string`, on peut accéder au premier caractère avec `front`, on peut ajouter un caractère à la fin avec `push_back`, etc. Ces fonctions, **qui s'appliquent à une instance donnée d'une classe**, sont appelées les **fonctions membres**. Nous en avons déjà parlé un peu dans le chapitre sur la documentation. C'est elles que nous retrouvons sous le titre *Member functions*.



Point vocabulaire

Ce terme «fonction membre» est à mettre en relation avec «**fonction libre**» (ou *Non-member functions* sur la documentation), qui désigne les fonctions qui ne font partie d'aucun objet. Toutes les fonctions que nous avons écrites jusque là étaient des fonctions libres. Beaucoup de celles que nous avons aussi utilisé aussi, comme la fonction `std::size`.

Toutes les fonctions membres d'une classe, ainsi que toutes les fonctions libres qui peuvent manipuler cette classe, composent ce qu'on nomme **l'interface d'une classe**.



Teaser

L'interface d'une classe regroupe encore d'autres concepts, pas uniquement les fonctions membres et les fonctions libres qui manipule des instances de cette classe. On peut citer, entres autres, la surcharge d'opérateurs. Nous verrons le reste plus tard dans ce cours.

V.2.2. En C++, ça donne quoi ?

Créer des types, ça, nous savons le faire depuis que nous avons découvert les structures. Jusqu'ici, nous les avons traitées comme de simples agrégats de données. Il est temps de changer notre vision des choses et de passer à une approche objet.

V.2.2.1. Penser services

Reprenons un exemple de la partie II, où nous avons manipulés des fractions. **Quels services peut-on attendre de la part d'une fraction?** Comment celle-ci se manipule-t-elle? Qu'est ce qui fait sens de faire avec une fraction? La liste ci-dessous décrit les services sur lesquels nous allons travailler. Elle n'est pas exhaustive, mais constitue un bon début.

- Obtenir sa valeur réelle.
- La simplifier.
- Calculer sa puissance au degré x .

V.2.2.2. Penser tests

Maintenant que nous avons défini une liste des services que nous aimerions appliquer à un objet de type `Fraction`, il faut réfléchir **aux tests** que nous allons écrire pour vérifier que notre implémentation est correcte. Prenez le temps de les écrire, c'est eux qui vous apporteront la garantie que votre code est de qualité.

Ci-dessous, voici les tests que j'ai écrits.

```
1 #include <cassert>
2
3 int main()
4 {
5     Fraction f1 { 5, 2 };
6     assert(f1.valeur_reelle() == 2.5 && "5/2 équivaut à 2.5.");
7     f1.simplification();
8     assert(f1.numerateur == 5 && f1.denominateur == 2 &&
9         "5/2 après simplification donne 5/2.");
10    f1 = pow(f1, 2);
11    assert(f1.numerateur == 25 && f1.denominateur == 4 &&
12        "5/2 au carré équivaut à 25/4.");
13
14    Fraction f2 { 258, 43 };
15    assert(f2.valeur_reelle() == 6 && "258/43 équivaut à 6.");
16    f2.simplification();
17    assert(f2.numerateur == 6 && f2.denominateur == 1 &&
18        "258/43 après simplification donne 6/1.");
19    f2 = pow(f2, 2);
20    assert(f2.numerateur == 36 && f2.denominateur == 1 &&
21        "6/1 au carré équivaut à 36/1.");
22
23    return 0;
24 }
```

Notez que les tests montrent qu'on aura à manipuler tant des fonctions membres que des fonctions libres.

V.2.2.3. Définition de la classe

V.3. Fonctions membres

Nous sommes maintenant prêts à définir notre classe. Commençons par les fonctions membres. Celles-ci se définissent au sein même de la classe, comme des fonctions classiques.

```
1 struct Fraction
2 {
3     double valeur_reelle();
4     void simplification();
5 };
```

Le prototype n'a rien d'extraordinaire, mais l'implémentation est quand même un poil plus inhabituelle. En effet, pour écrire l'implémentation d'une fonction membre, il faut **préfixer son identifiant par le nom de la classe**, avec l'opérateur `::` en guise de séparateur. C'est nécessaire pour que le compilateur sache que la fonction définie est membre d'une classe.

```
1 struct Fraction
2 {
3     double valeur_reelle();
4     void simplification();
5 };
6
7 double Fraction::valeur_reelle()
8 {
9
10 }
11
12 void Fraction::simplification()
13 {
14
15 }
```



Un mot sur

On retrouve le même principe avec des objets que vous manipulez depuis le premier jour, comme `std::string` ou `std::vector`. C'est aussi lui qu'on utilise avec les énumérations. L'opérateur `::` est ce qu'on appelle un **opérateur de résolution de portée**. Il permet de lever toute ambiguïté éventuelle sur l'origine d'un type, d'une fonction ou d'une variable.



Dans le cas de `Fraction::valeur_reelle`, on indique qu'il existe une fonction `valeur_reelle` dans la portée `Fraction`, qui est ici une classe. Dans le cas de `std::string`, on indique cette fois qu'il existe un type `string` dans une portée `std`. Cette fois, ce n'est pas d'une classe dont il s'agit, mais d'un **espace de noms** (en anglais *namespace*), qui permet de regrouper plusieurs identificateurs sous une portée commune.

Grâce à ces portées, on peut mieux «cloisonner» son code, ce qui est plus propre. L'espace de nom `std` permet, par exemple, de regrouper tout ce qui fait partie de la bibliothèque standard. De même, `Fraction::` regroupe tout ce qui a trait à notre classe `Fraction`.

V.3.0.0.1. Fonctions libres

Continuons sur notre lancée en définissant des fonctions libres. De la même manière que `size` est à la fois une fonction membre des classes `std::string` et `std::vector` et à la fois une fonction libre, nous sommes **libres de faire pareil pour une classe `Fraction`**. Nous devons simplement réfléchir au pour et au contre.

- Certaines fonctions sont obligatoirement implémentées sous la forme **membre**, comme l'opérateur `[]`.
- D'autres sont obligatoirement implémentées sous la forme **libre**, comme l'opérateur `<`.
- Enfin, pour celles qui peuvent être implémentées tant libres que membres, il faut comparer pour savoir **quelle façon de faire est la plus simple**. Notamment, si un service peut être rendu en utilisant uniquement d'autres services, mieux vaut l'écrire sous forme de fonction libre. Si, au contraire, il a besoin de manipuler des détails internes, alors il vaut mieux l'implémenter en tant que fonction membre.

Dans le cas d'une fraction, il semble logique d'attendre d'une instance qu'elle nous donne sa valeur réelle et sa version simplifiée. Par contre, mettre une fraction à une puissance x , c'est comme `std::pow`, mais pour une fraction. Il paraît donc bien d'offrir ce service sous forme de fonction libre. Cela uniformise l'utilisation de `pow`, qui a alors la même utilisation que ce soit pour un entier ou pour une `Fraction`.

```
1 Fraction pow(Fraction const & fraction, int puissance)
2 {
3 }
```

V.3.0.0.2. Les attributs

Maintenant que nous avons défini l'interface, il est temps de se plonger dans les détails d'implémentation. N'oublions pas que, même si nous pensons et concevons services, il faut bien des données à manipuler. Dans notre cas, il s'agit d'un **numérateur** et d'un **dénominateur**. On peut maintenant compléter la définition de la classe.

```

1  #include <cassert>
2  #include <cmath>
3  #include <numeric>
4
5  struct Fraction
6  {
7      // Nos fonctions membres.
8      double valeur_reelle();
9      void simplification();
10
11     // Nos attributs.
12     int numerateur;
13     int denominateur;
14 };
15
16 double Fraction::valeur_reelle()
17 {
18     return static_cast<double>(numerateur) / denominateur;
19 }
20
21 void Fraction::simplification()
22 {
23     int const pgcd { std::gcd(numerateur, denominateur) };
24     numerateur /= pgcd;
25     denominateur /= pgcd;
26 }
27
28 // Déclaration et implémentation d'un service sous forme libre.
29 Fraction pow(Fraction const & fraction, int puissance)
30 {
31     Fraction copie { fraction };
32     copie.numerateur = std::pow(fraction.numerateur, puissance);
33     copie.denominateur = std::pow(fraction.denominateur,
34     puissance);
35     return copie;
36 }

```



Note sur `static_cast`

Si l'on divise un entier par un autre entier, on obtient un entier. Notre résultat serait alors faussé, puisqu'on perdrait toute la partie décimale. Pour éviter ce problème, il suffit que l'un des deux opérandes soit de type `double`. C'est ce que fait `static_cast`, en indiquant au compilateur que cette variable est en fait un réel et non plus un entier.

Notez que les fonctions membres manipulent sans problème les attributs de la classe. Chaque instance aura ses propres attributs, donc quand j'écris `f1.simplification()`, c'est le numérateur et le dénominateur de `f1` qui sont modifiés et uniquement eux.

V.3.1. Désolé, cet objet n'est pas modifiable

Revenons à un mot-clé que nous connaissons depuis longtemps, que l'on utilise beaucoup mais que j'ai volontairement laissé de côté depuis l'introduction à la POO. Il s'agit de `const`. Il nous permet de déclarer qu'un objet est constant et ainsi empêcher d'éventuelles modifications. Sauf que si on déclare notre objet `Fraction` comme étant `const`, ça ne compile plus.

```

1  #include <cmath>
2  #include <iostream>
3  #include <numeric>
4
5  struct Fraction
6  {
7      double valeur_reelle();
8      void simplification();
9
10     int numérateur;
11     int dénominateur;
12 };
13
14 double Fraction::valeur_reelle()
15 {
16     return static_cast<double>(numérateur) / dénominateur;
17 }
18
19 void Fraction::simplification()
20 {
21     int const pgcd { std::gcd(numérateur, dénominateur) };
22     numérateur /= pgcd;
23     dénominateur /= pgcd;
24 }
25
26 Fraction pow(Fraction const & fraction, int puissance)
27 {
28     Fraction copie { fraction };
29     copie.numérateur = std::pow(fraction.numérateur, puissance);
30     copie.dénominateur = std::pow(fraction.dénominateur,
31     puissance);
32     return copie;
33 }
34
35 int main()
36 {
37     Fraction const f1 { 5, 2 };
38     std::cout << "Voici la valeur réelle de 5/2 : " <<
39         f1.valeur_reelle() << "\n";
40     return 0;
41 }

```

```

1 [Visual Studio]
2 E1086      l'objet a des qualificateurs de type incompatibles
   avec le membre fonction "Fraction::valeur_reelle"
3 C2662      'double Fraction::valeur_reelle(void)' : impossible de
   convertir un pointeur 'this' de 'const Fraction' en 'Fraction
   &'
4
5 -----
6
7 [GCC]
8 tests.cpp: In function 'int main()':
9 tests.cpp:48:74: error: passing 'const Fraction' as 'this' argument
   discards qualifiers [-fpermissive]
10    48 |      std::cout << "Voici la valeur réelle de 5/2 : " <<
      |      f1.valeur_reelle() << "\n";
11        |      ^
12 tests.cpp:18:8: note:   in call to 'double
   Fraction::valeur_reelle()'
13    18 | double Fraction::valeur_reelle()
      |      |      ^~~~~~
14
15 -----
16
17 [Clang]
18 prog.cc:43:57: error: 'this' argument to member function
   'valeur_reelle' has type 'const Fraction', but function is not
   marked const
19    std::cout << "Voici la valeur réelle de 5/2 : " <<
20    f1.valeur_reelle() << "\n";
21                                     ^~
22 prog.cc:15:18: note: 'valeur_reelle' declared here
23 double Fraction::valeur_reelle()
24                ^
25 1 error generated.

```

Le message émis par GCC est un peu moins clair que les autres, mais les trois indiquent que notre fonction membre n'est pas utilisable si l'instance est déclarée comme constante, **parce que cette fonction membre ne l'est pas**.

Une fonction membre constante, c'est **une fonction qui promet de ne pas modifier l'objet auquel elle appartient**. Par exemple, dans le cas de `std::vector`, la fonction membre `size` retourne simplement la taille du tableau. Cette fonction donc est déclarée comme étant constante et on peut l'utiliser sur des tableaux constants. Au contraire, `push_back` modifie le tableau et n'est donc utilisable que sur un tableau non `const`.

Le mieux est d'avoir **le plus de fonctions constantes possibles**. Moins on modifie de choses, moins on a de chance d'introduire des erreurs et des bugs. Il convient donc de réfléchir, dès

la conception d'une classe, de façon à maximiser les fonctions constantes. Ici, on voit que `valeur_reelle` est un très bon candidat pour ça.

Pour qu'une fonction membre soit utilisable même quand l'objet est constant, **il faut qu'elle soit explicitement marquée comme constante**. Cela se fait très simplement, en rajoutant le mot-clef `const` après la signature de la fonction en question, tant dans le prototype que dans l'implémentation.

```
1 struct Fraction
2 {
3     double valeur_reelle() const;
4 };
5
6 // Ne pas oublier const dans l'implémentation aussi.
7 double Fraction::valeur_reelle() const
8 {
9     return static_cast<double>(numérateur) / dénominateur;
10 }
```

Bien sûr, certaines fonctions ne pourront jamais être `const`. Par exemple, vouloir déclarer `simplification` comme étant constante n'a aucun sens, puisque c'est sa raison d'être que de modifier l'instance sur laquelle elle est appelée. Si vous essayez, le compilateur va vous envoyer balader.

```
1 [Visual Studio]
2 C3490 impossible de modifier 'numérateur' car il est
   accessible via un objet const
3 C3490 impossible de modifier 'dénominateur' car il est
   accessible via un objet const
4
5 -----
6
7 [GCC]
8 tests.cpp: In member function 'void Fraction::simplification()
   const':
9 tests.cpp:24:16: error: assignment of member 'Fraction::numérateur'
   in read-only object
10 24 |         numérateur /= pgcd;
11    |         ~~~~~^~~~~~
12 tests.cpp:25:18: error: assignment of member
   'Fraction::dénominateur' in read-only object
13 25 |         dénominateur /= pgcd;
14    |         ~~~~~^~~~~~
15
16 -----
17
18 [Clang]
```

```

19 prog.cc:24:16: error: cannot assign to non-static data member
   numerateur /= pgcd;
   ~~~~~^
22 prog.cc:21:16: note: member function 'Fraction::simplification' is
   declared const here
23 void Fraction::simplification() const
24 ~~~~~^~~~~~
25 prog.cc:25:18: error: cannot assign to non-static data member
   denominateur /= pgcd;
   ~~~~~^
28 prog.cc:21:16: note: member function 'Fraction::simplification' is
   declared const here
29 void Fraction::simplification() const
30 ~~~~~^~~~~~
31 2 errors generated.

```

V.3.2. On ne fait pas d'exception

Dans le chapitre sur la documentation, nous avons vu qu'il existe une section **Exceptions**, qui nous indique si une fonction peut potentiellement lancer une exception, ou non. Pour celles qui ont la garantie de ne rien lancer, il y a le mot-clé **noexcept**. Nous avons la possibilité de **faire de même pour les classes que nous créons**.

Par exemple, la fonction membre **valeur_reelle** consiste en une simple division et ne lève donc jamais d'exception. De même, **simplification** ne fait appel qu'à **std::gcd** en interne. Si on regarde la [documentation](#) [↗](#), on a la garantie que **std::gcd** ne lance jamais d'exception non plus. Quant à **std::pow**, aucune information n'est donnée. Nous la laisserons telle quelle.

Comme pour **const**, le mot-clé **noexcept** s'écrit **deux fois**, tant après le prototype au sein de la classe qu'après la signature au moment de l'implémentation.

```

1  #include <cmath>
2  #include <iostream>
3  #include <numeric>
4
5  struct Fraction
6  {
7      double valeur_reelle() const noexcept;
8      void simplification() noexcept;
9
10     int numerateur;
11     int denominateur;
12 };
13

```

```

14 double Fraction::valeur_reelle() const noexcept
15 {
16     return static_cast<double>(numérateur) / dénominateur;
17 }
18
19 void Fraction::simplification() noexcept
20 {
21     int const pgcd { std::gcd(numérateur, dénominateur) };
22     numérateur /= pgcd;
23     dénominateur /= pgcd;
24 }
25
26 Fraction pow(Fraction const & fraction, int puissance)
27 {
28     Fraction copie { fraction };
29     copie.numérateur = std::pow(fraction.numérateur, puissance);
30     copie.dénominateur = std::pow(fraction.dénominateur,
31     puissance);
32     return copie;
33 }
34
35 int main()
36 {
37     Fraction const f1 { 5, 2 };
38     std::cout << "Voici la valeur réelle de 5/2 : " <<
39         f1.valeur_reelle() << "\n";
40     return 0;
41 }

```

V.3.3. Découpage en fichiers

Bien que tout écrire dans un fichier soit plus rapide et bien plus rapide pour des codes d'exemples, dans la réalité, on sépare les définitions des implémentations, vous le savez depuis le chapitre sur le découpage en fichier. Le principe est le même que ce que vous connaissez.

Fraction.hpp

```
1  #ifndef FRACTION_HPP
2  #define FRACTION_HPP
3
4  struct Fraction
5  {
6      double valeur_reelle() const noexcept;
7      void simplification() noexcept;
8
9      int numerateur;
10     int denominateur;
11 };
12
13 Fraction pow(Fraction const & fraction, int puissance);
14
15 #endif
```

Fraction.cpp

```

1  #include <cmath>
2  #include <numeric>
3
4  #include "Fraction.hpp"
5
6  double Fraction::valeur_reelle() const noexcept
7  {
8      return static_cast<double>(numérateur) / dénominateur;
9  }
10
11 void Fraction::simplification() noexcept
12 {
13     int const pgcd { std::gcd(numérateur, dénominateur) };
14     numérateur /= pgcd;
15     dénominateur /= pgcd;
16 }
17
18 Fraction pow(Fraction const & fraction, int puissance)
19 {
20     Fraction copie { fraction };
21     copie.numérateur = std::pow(fraction.numérateur,
22                                 puissance);
23     copie.dénominateur = std::pow(fraction.dénominateur,
24                                   puissance);
25     return copie;
26 }

```

main.cpp

```

1  #include <iostream>
2  #include "Fraction.hpp"
3
4  int main()
5  {
6      Fraction const f1 { 5, 2 };
7      std::cout << "5/2 = " << f1.valeur_reelle() << "\n";
8      return 0;
9  }

```

Pour la suite du cours, la majorité des exemples resteront écrits dans un seul fichier, par simplicité et facilité. Néanmoins, quand vous commencerez des projets plus gros, découpez et

vous gagnerez en lisibilité et compréhension.

V.3.4. Exercices

V.3.4.1. Cercle

Imaginons que l'on veuille représenter un cercle. De nombreuses opérations sont possibles.

- On peut calculer sa surface.
- On peut calculer son périmètre.
- On peut déterminer si un point fait partie du cercle ou non.

Si on se base sur ces informations, alors on peut déjà imaginer les tests qui suivent.

```
1 #include <cassert>
2 #include <iostream>
3
4 bool compare(double lhs, double rhs)
5 {
6     return std::abs(lhs - rhs) < 0.001;
7 }
8
9 int main()
10 {
11     Point const centre { 0, 0 };
12     Cercle const c1 { 1, centre };
13
14     std::cout << "Voici l'aire du cercle : " << c1.aire() << "\n";
15     assert(compare(c1.aire(), 3.1415) &&
16            "L'aire du cercle c1 doit valoir pi.");
17
18     std::cout << "Voici le périmètre du cercle : " <<
19             c1.perimetre() << "\n";
20     assert(compare(c1.perimetre(), 3.1415 * 2) &&
21            "Le périmètre du cercle doit valoir pi * 2.");
22
23     std::cout << std::boolalpha;
24     Point const p1 { 2, -18 };
25     std::cout <<
26         "Est-ce que le point p1 appartient au cercle c1 ? " <<
27         appartient(c1, p1) << "\n";
28     assert(!appartient(c1, p1) &&
29            "Le point p1 n'appartient pas au cercle c1.");
30     return 0;
31 }
```



Un mot sur la fonction `compare`

Je vous ai ajouté une fonction `compare`, qui permet de dire si deux nombres réels sont égaux en ne se basant **que sur les trois premières décimales**. Sans ça, si on les compare directement avec `==`, alors la comparaison se fera sur les milliers de décimales qui composent nos `double`.

On risque d'avoir des **résultats faussés**, car le programme nous dira alors que deux nombres sont différents car leurs 20000ème décimales sont différentes, ce qui est négligeable à notre niveau.

👁 Indices

👁 Correction

V.3.4.2. Informations personnelles

Vous souvenez-vous de la structure `InformationsPersonnelles`, que nous avons écrite au chapitre 5 de la partie II? Pourrait-on la réécrire avec un point de vue objet? Quels services pourrait-elle rendre? Quels invariants pourraient être mis en place? Je laisse ces questions ouvertes, à vous de décider. 🍌

V.3.4.3. En résumé

- La programmation orientée objet, c'est penser en termes de services rendus et non en terme de données.
- Une classe est un type personnalisé que l'on définit comme on veut.
- On parle d'instancier une classe quand on crée un objet de ce type.
- L'interface d'une classe comprend notamment les fonctions membres et les fonctions libres qui manipulent les instances de cette classe.
- Seules les fonctions membres constantes peuvent manipuler des instances déclarées `const`.
- Les fonctions membres doivent être déclarées constantes le plus possible.
- De la même manière, il est bien de déclarer `noexcept` toutes les fonctions membres qui le sont.
- La séparation en fichiers nous permet de séparer la définition d'une classe de l'implémentation de ses fonctions membres.

Contenu masqué

Contenu masqué n°74 :

Indices

Petits rappels mathématiques.

- L'aire d'un cercle de rayon r vaut $r^2 \times \pi$.
- Le périmètre d'un cercle de rayon r vaut $2 \times r \times \pi$.
- La distance entre deux points $P1(x; y)$ et $P2(a; b)$ se calcule avec la formule $\sqrt{(x - a)^2 + (y - b)^2}$.

[Retourner au texte.](#)

Contenu masqué n°75 : Correction

Comme un cercle a un point central, j'ai séparé les opérations concernant les points de celles concernant les cercles. En effet, le calcul de la distance entre deux points n'est pas une opération réservée aux seuls cercles, mais peut être effectuée pour n'importe quel point.

J'ai aussi décidé de fournir le service `distance` et `appartient` comme des fonctions libres, car, de mon point de vue, ces opérations ne sont pas «liées à une instance en particulier», mais sont générales. C'est un choix de conception.

Point.hpp

```
1  #ifndef POINT_HPP
2  #define POINT_HPP
3
4  struct Point
5  {
6      int x;
7      int y;
8  };
9
10 double distance(Point const & lhs, Point const & rhs);
11
12 #endif
```


Point.cpp

```
1 #include <cmath>
2 #include "Point.hpp"
3
4 double distance(Point const & lhs, Point const & rhs)
5 {
6     return std::sqrt(std::pow(lhs.x - rhs.x, 2) +
7         std::pow(lhs.y - rhs.y, 2));
8 }
```

Cercle.hpp

```
1 #ifndef CERCLE_HPP
2 #define CERCLE_HPP
3
4 #include <cmath>
5 #include "Point.hpp"
6
7 struct Cercle
8 {
9     double aire() const noexcept;
10    double perimetre() const noexcept;
11
12    double rayon;
13    Point centre;
14 };
15
16 bool appartient(Cercle const & cercle, Point const & point);
17
18 // Il n'y a malheureusement pas (encore) de constante PI dans
19 // la bibliothèque standard.
20 // On doit donc créer la notre en utilisant la propriété que
21 // arcsinus(-1) vaut PI.
22 double const pi { std::acos(-1) };
23
24 #endif
```

Cercle.cpp

```
1 #include "Cercle.hpp"
2
3 double Cercle::aire() const noexcept
4 {
5     return rayon * rayon * pi;
6 }
7
8 double Cercle::perimetre() const noexcept
9 {
10    return 2 * rayon * pi;
11 }
12
13 bool appartient(Cercle const & cercle, Point const & point)
14 {
15    return distance(cercle.centre, point) <= cercle.rayon;
16 }
```

[Retourner au texte.](#)

V.4. Qui c'est qui va construire tout ça ?

Dans le chapitre précédent, nous avons appris à réfléchir en terme de services et nous avons vu comment le faire concrètement. Mais la programmation orientée objet, ce n'est pas que ça, autrement elle ne vaudrait pas le détour. Dans ce chapitre, nous allons voir quelques problèmes qui existent avec notre classe `Fraction` et les techniques qui existent pour les résoudre et améliorer la qualité de notre code.

Ce chapitre va nous introduire **aux invariants**, **aux constructeurs** et à la **gestion d'erreurs**.

V.4.1. Encapsulation et invariants

V.4.1.1. Cette classe, c'est *open bar*

Notre classe `Fraction` a été définie et implémentée suivant une logique de services et c'est une bonne chose. Reste que, pour l'instant, nos attributs sont toujours **visibles et librement modifiables**. Quel intérêt d'aller au pressing si n'importe qui peut avoir accès aux machines à laver et mettre le bazar ?

Vous voulez un exemple d'erreur qu'on peut obtenir en faisant n'importe quoi ? Regardez le code ci-dessous.

```
1 int main()
2 {
3     Fraction f1 { 4, 2 };
4     std::cout << "4/2 = " << f1.valeur_reelle() << std::endl;
5
6     // Erreur !
7     f1.denominateur = 0;
8     // Ça va faire boom !
9     std::cout << "4/0 = " << f1.valeur_reelle() << std::endl;
10
11     return 0;
12 }
```

```
1 4/2 = 2
2 4/0 = inf
```

Rappelons que diviser un entier par 0 est mathématiquement indéfini. Je peux donc obtenir tout et n'importe quoi comme résultat. Ici, j'obtiens l'infini, mais ça aurait très bien pu être autre chose (d'où le résultat peut-être différent que vous avez chez vous).

Le vrai problème, c'est qu'en modifiant directement un attribut, je m'attribue des droits qui ne sont pas les miens et je peux **potentiellement tout casser**. Si on part du principe que ma classe doit représenter fidèlement le concept mathématique de fraction, alors **on est face à un problème**, puisque mon objet n'est désormais **plus valide**.

V.4.1.2. Tout est question d'invariant

Justement, qu'est-ce qui me permet de dire que mon objet n'est plus valide? C'est qu'il **ne respecte plus une condition qui devrait toujours être vraie**, celle que le dénominateur ne doit jamais valoir zéro. Cette condition, qu'on appelle **invariant**, c'est moi, le concepteur de la classe, qui l'ai définie.

J'aurais très bien pu dire que je m'en fiche et que ce n'est pas une erreur de diviser par zéro. Ce sont des **choix de conception**. Je veux implémenter une classe qui offre des services proches de ceux d'une fraction mathématiquement parlant, donc je décide qu'il y a toujours un invariant à respecter, celui que le dénominateur ne soit jamais nul.

Tant que cet invariant tient toujours, chaque instance de ma classe **Fraction** est supposée rendre les services qu'elle expose (simplification, puissance, etc). Par contre, s'il vient à être violé, alors tout peut arriver. C'est typiquement de la **programmation par contrat**, déjà abordée dans la partie II.

Souvenez-vous de ce que nous avons parlé dans le chapitre consacré aux fonctions. Nous avons vu qu'il existe les **préconditions**, qui doivent être respectées avant l'appel à une fonction. Nous avons aussi parlé des **postconditions**, qu'une fonction doit garantir quand elle termine de s'exécuter. Le principe est le même ici.

Vous voulez d'autres exemples? Revenons à notre blanchisserie. Son invariant est que votre linge soit toujours en bon état. Si vous vous amusez à changer la lessive ou la programmation de la machine à laver vous-mêmes, sans rien y connaître, il ne faudra pas s'étonner que le linge soit décoloré ou rétréci.

L'invariant d'un carré, c'est que la longueur soit égale à la largeur. L'invariant d'un jeu de tarot, c'est qu'il y ait 56 cartes. L'invariant d'une date, c'est que les jours, les mois et les années renseignés soient valides. L'invariant d'une machine à café, c'est qu'elle ne fasse pas de court-jus. En fait, dès qu'il y a **des conditions à respecter quand on veut créer un objet**, c'est qu'il y a un ou plusieurs invariants.

V.4.1.3. Encapsulons tout ça

La solution pour respecter nos invariants s'appelle **l'encapsulation**. Plutôt que de laisser nos attributs librement modifiables par le premier venu, nous allons les protéger, les enrober d'une couche protectrice, d'une capsule. Seuls les services seront publics et seuls eux seront à même de modifier les attributs. Il faut pour cela **changer leur visibilité**. Il en existe trois, qui peuvent s'appliquer tant aux attributs qu'aux fonctions membres.

- Ils peuvent être **publics** et accessibles depuis le reste du monde, par tout le monde.
- Ils peuvent être **privés** et seule les instances de la classe peuvent les manipuler et les utiliser.
- Ils peuvent être d'un troisième type, que nous verrons en temps voulu.



Un mot sur la visibilité

Ce concept de visibilité est à mettre en lien avec le découpage en fichiers d'en-têtes et fichiers sources. Dans les premiers, on trouve les fonctions, les classes, etc qu'un code peut appeler et utiliser. Ce sont les **interfaces publiques**, ce que le reste du monde voit. Au contraire, dans les fichiers sources, qui contiennent les détails d'implémentations, tout est **masqué**.

De la même manière, **nos services seront publiquement exposés**, alors que **les détails d'implémentations seront cachés et inaccessibles** par le reste du monde.

Par défaut, une structure a tous ses éléments publics, d'où le bug illustré précédemment. Heureusement, la visibilité n'est pas figée dans le marbre.

V.4.1.4. Modifier sa visibilité

Il existe deux mots-clés, **public** et **private**, qui permettent respectivement de rendre un élément public ou privé. C'est justement ce dont nous avons besoin. Grâce à **public**, nous pouvons permettre au reste du monde de faire appel à nos services. Grâce à **private**, on se protège en interdisant toute modification directe inappropriée.

Chaque mot-clé s'applique jusqu'à ce qu'on rencontre un autre modificateur de visibilité, ou jusqu'à la fin de la classe s'il n'y en a pas d'autres.

```
1  class Fraction
2  {
3      // Tout ce qui est en-dessous est maintenant public.
4  public:
5      double valeur_reelle() const noexcept;
6      void simplification() noexcept;
7
8      // Tout ce qui est en-dessous est maintenant privé et caché.
9  private:
10     int m_numerateur;
11     int m_denominateur;
12 };
```

- Les attributs sont maintenant **préfixés par m_**. Cela permet d'un coup d'œil de voir qu'on a affaire à un attribut privé. Cette convention n'est pas obligatoire, mais est très répandue dans le monde C++. Nous l'utiliserons dans la suite de ce cours.
- Le mot-clé **struct** a été remplacé par **class**. Les deux sont interchangeables, la seule différence étant que, par défaut, tout est public avec **struct**, alors que tout est privé avec **class**.



`struct` ou `class`?

Les deux sont très proches et on peut sans problème adapter la visibilité dans les deux cas. Il y a néanmoins une convention répandue, qui stipule d'utiliser **`struct`** dans le cas d'une simple agrégation de données et de réserver **`class`** quand il y a présence d'au moins un invariant.

Par exemple, une structure `Point`, regroupant des coordonnées $(x; y; z)$, n'a pas vraiment d'invariant et peut être laissée comme telle. Au contraire, une fraction peut présenter au moins un invariant, celui de ne jamais avoir de dénominateur à zéro. C'est donc adapté pour une classe.

V.4.2. On cherche un constructeur

Si vous essayez de compiler le code ci-dessous, qui reprend toutes les modifications que nous venons de faire, vous allez obtenir une erreur. En effet, nous ne pouvons plus initialiser notre objet avec les valeurs que l'on souhaite.

```

1  #include <cmath>
2  #include <iostream>
3  #include <numeric>
4
5  class Fraction
6  {
7  public:
8      double valeur_reelle() const noexcept;
9      void simplification() noexcept;
10
11 private:
12     int m_numerateur;
13     int m_denominateur;
14 };
15
16 double Fraction::valeur_reelle() const noexcept
17 {
18     return static_cast<double>(m_numerateur) / m_denominateur;
19 }
20
21 void Fraction::simplification() noexcept
22 {
23     int const pgcd { std::gcd(m_numerateur, m_denominateur) };
24     m_numerateur /= pgcd;
25     m_denominateur /= pgcd;
26 }
27
28 int main()
```

```

29 {
30     Fraction const f1 { 5, 2 };
31     std::cout << "5/2 = " << f1.valeur_reelle() << "\n";
32     return 0;
33 }

```

```

1 [Visual Studio]
2 Erreur C2440 'initialisation' : impossible de convertir de
   'initializer list' en 'Fraction' TestCpp.cpp 30
3 Erreur E0146 trop de valeurs d'initialiseur TestCpp.cpp 30
4
5 -----
6
7 [GCC]
8 prog.cc: In function 'int main()':
9 prog.cc:44:30: error: no matching function for call to
   'Fraction::Fraction(<brace-enclosed initializer list>)'
10    44 |         Fraction const f1 { 5, 2 };
11        |         ^
12 prog.cc:5:7: note: candidate: 'Fraction::Fraction()'
13    5 | class Fraction
14        |     ^~~~~~
15 prog.cc:5:7: note: candidate expects 0 arguments, 2 provided
16 prog.cc:5:7: note: candidate: 'constexpr Fraction::Fraction(const
   Fraction&)'
17 prog.cc:5:7: note: candidate expects 1 argument, 2 provided
18 prog.cc:5:7: note: candidate: 'constexpr
   Fraction::Fraction(Fraction&&)'
19 prog.cc:5:7: note: candidate expects 1 argument, 2 provided
20
21 -----
22
23 [Clang]
24 prog.cc:44:20: error: no matching constructor for initialization of
   'const Fraction'
25    44 |         Fraction const f1 { 5, 2 };
26        |         ^ ~~~~~~
27 prog.cc:5:7: note: candidate constructor (the implicit copy
   constructor) not viable: requires 1 argument, but 2 were
   provided
28 class Fraction
29     ^
30 prog.cc:5:7: note: candidate constructor (the implicit move
   constructor) not viable: requires 1 argument, but 2 were
   provided
31 prog.cc:5:7: note: candidate constructor (the implicit default
   constructor) not viable: requires 0 arguments, but 2 were
   provided

```

```
32 1 error generated.
```

Le message d'erreur de Clang est plus clair que celui de GCC, puisqu'il nous indique clairement qu'il ne trouve pas le bon **constructeur**. Cette fonction-membre spéciale est appelée **dès l'instanciation d'une classe** et s'occupe d'**initialiser l'objet**. Comme il s'agit du point d'entrée depuis l'extérieur, c'est ici qu'on va pouvoir **définir nos invariants**.

- Si tous les invariants sont respectés, alors l'objet est **correctement initialisé** et utilisable.
- Si un des invariants est violé, alors on a **une erreur**, qu'on va signaler à l'utilisateur de notre code.

Notez que certains constructeurs sont définis implicitement par le compilateur.

- Le **constructeur par défaut**, que nous allons voir tout de suite. C'est celui que GCC désigne par `Fraction::Fraction()`.
- Le **constructeur par copie**, que nous verrons dans le chapitre suivant, de la forme `constexpr Fraction::Fraction(const Fraction&)`.
- Le **constructeur par mouvement**, que nous verrons également, mais plus tard. C'est celui qui reste, `constexpr Fraction::Fraction(Fraction&&)`.

V.4.2.1. La syntaxe

Tous les constructeurs d'une classe partagent la même syntaxe, ce qui va simplifier les choses pour vous. En fait, seuls les paramètres et leur nombre changent. Examinons ce que nous apprend GCC.

Déjà, aucun constructeur n'a de **type de retour**, pas même `void`. Le prototype du constructeur est donc constitué uniquement du **nom de la classe**, plus les éventuels arguments. Et comme pour les autres fonctions membres, on retrouve le nom de la classe et l'opérateur de résolution de portée `::`, comme nous l'avons vu juste à l'instant avec le message de GCC. Enfin, on peut retrouver `constexpr` et/ou `noexcept`, si l'on le désire.

L'autre point surprenant, c'est que l'initialisation des attributs se fait directement en-dessous de sa signature. On appelle cette syntaxe **la liste d'initialisation des membres** (en anglais *member initializer lists*). Voyez par vous-même l'application concrète à notre classe `Fraction`.

```
1 #include <cassert>
2 #include <cmath>
3 #include <iostream>
4 #include <numeric>
5
6 class Fraction
7 {
8 public:
9     // Pas d'exception car on ne fait qu'initialiser deux entiers.
10    Fraction(int numérateur, int dénominateur) noexcept;
```



```

11     double valeur_reelle() const noexcept;
12     void simplification() noexcept;
13
14 private:
15     int m_numérateur;
16     int m_dénominateur;
17 };
18
19 // Le qualificateur noexcept se répète aussi ici.
20 Fraction::Fraction(int numérateur, int dénominateur) noexcept
21     : m_numérateur(numérateur), m_dénominateur(dénominateur)
22 {
23     assert(m_dénominateur != 0 &&
24           "Le dénominateur ne peut pas valoir 0.");
25 }
26
27 double Fraction::valeur_reelle() const noexcept
28 {
29     return static_cast<double>(m_numérateur) / m_dénominateur;
30 }
31
32 void Fraction::simplification() noexcept
33 {
34     int const pgcd { std::gcd(m_numérateur, m_dénominateur) };
35     m_numérateur /= pgcd;
36     m_dénominateur /= pgcd;
37 }
38
39 int main()
40 {
41     Fraction const f1 { 5, 2 };
42     std::cout << "5/2 = " << f1.valeur_reelle() << "\n";
43     return 0;
44 }

```

L'avantage de passer ainsi par un constructeur pour faire ça, c'est qu'on peut **vérifier nos contrats AVANT de créer l'objet**. Ainsi, si on reçoit un argument incorrect ou si une erreur externe survient, il est possible de réagir immédiatement, avec une assertion ou une exception.

Tout comme nous avons des `assert` pour vérifier les préconditions de nos fonctions libres, nous avons les constructeurs qui, en plus d'initialiser l'objet et de définir ses invariants, vont **vérifier qu'il est possible de le créer à l'aide de préconditions**.

V.4.3. Constructeur par défaut

Appelé en anglais *default constructor*, c'est lui qui est appelé quand on instancie une classe sans fournir d'argument. Par exemple, quand nous écrivons `std::string chaine {}` ou

`std::vector` `tableau {}`, nos instances sont initialisées par défaut. Dans le premier cas, on obtient une chaîne de caractères vide, dans l'autre un tableau vide.

Par défaut, le compilateur en génère automatiquement un, sauf si des constructeurs personnalisés sont définis. Le problème, c'est qu'il est **très basique**. Pour tous les attributs objets, comme `std::string` ou `std::vector`, il se contente d'appeler leur constructeur par défaut. Par contre, dans le cas des types natifs, rien n'est fait, leur valeur est **indéterminée**, elle peut valoir n'importe quoi. Cela peut mener à quelques surprises, comme dans le cas de la classe `Fraction`.

```
1 int main()
2 {
3     Fraction const f1 {};
4     std::cout << "Valeur réelle par défaut = " <<
5         f1.valeur_reelle() << "\n";
6     return 0;
}
```

```
1 [Visual Studio]
2 Valeur réelle par défaut = -nan(ind)
3
4 -----
5
6 [GCC]
7 Valeur réelle par défaut = -nan
8
9 -----
10
11 [Clang]
12 Valeur réelle par défaut = -nan
```

Le code ci-dessus est buggué parce que `m_numérateur` et `m_dénominateur` peuvent potentiellement **valoir n'importe quoi**, d'où le résultat étrange. Notre programme n'est donc pas très robuste, puisqu'il est **impossible de prévoir le résultat**.

Moralité, le constructeur par défaut ne nous plaît pas. Nous devons donc réfléchir et déterminer si créer une «fraction par défaut» a, ou non, du sens. À partir de cette réflexion, on part dans **deux directions**.

V.4.3.1. Initialiser une instance par défaut a du sens

On peut décider, à bon droit, qu'une fraction sans plus autre détail vaut 0, ou 1, ou 42, ou encore 97455. C'est **notre choix** en tant que concepteur. Pour cet exemple, nous déciderons qu'une fraction vaut par défaut zéro. On peut directement **initialiser nos attributs** et nous résolvons le problème sans même avoir à écrire nous-mêmes le constructeur par défaut.

```
1 class Fraction
2 {
3 private:
4     int m_numerator { 0 };
5     int m_denominator { 1 };
6 }
```

Dans le cas où le constructeur par défaut fait son travail, pas la peine de le redéfinir, cela ne ferait que rajouter de la lourdeur au code. On peut simplement demander explicitement au compilateur de le générer, ce qui rend le code plus clair. Cela se fait en rajoutant `= default` après la signature.

```
1 class Fraction
2 {
3 public:
4     Fraction() = default;
5
6 private:
7     int m_numerator;
8     int m_denominator;
9 }
```

V.4.3.2. Initialiser une instance par défaut n'a pas de sens

On peut aussi très bien décider, à l'opposé, que l'utilisateur doit explicitement initialiser toute fraction qu'il souhaite créer. C'est aussi un choix valable. Dans ce cas, on va lui **interdire l'utilisation du constructeur par défaut**.

Si la classe contient d'autres constructeurs, le constructeur par défaut n'est pas implicitement créé, donc impossible de l'utiliser. Mais comme la clarté est une bonne pratique que nous nous efforçons de vous transmettre, nous allons **dire explicitement au compilateur que nous n'en voulons pas**. Cela se fait en ajoutant `= delete` à la fin du prototype du constructeur par défaut.

```
1 class Fraction
2 {
3 public:
4     Fraction() = delete;
5
6 private:
7     int m_numerator;
8     int m_denominator;
9 };
```

Voici l'erreur que vous obtiendrez si vous essayez quand même d'utiliser le constructeur par défaut.

```
1 [Visual Studio]
2 Erreur C2280 'Fraction::Fraction(void)' : tentative de
  référencement d'une fonction supprimée
3 Erreur (active)      E1790 impossible de faire référence au
  constructeur par défaut de "Fraction"
4
5 -----
6
7 [GCC]
8 main.cpp: In function 'int main()':
9 main.cpp:26:18: error: use of deleted function
   'Fraction::Fraction()'
10    26 |         Fraction f1 {};
11        |                ^
12 main.cpp:10:5: note: declared here
13    10 |         Fraction() = delete;
14        |         ^~~~~~
15
16 -----
17
18 [Clang]
19 prog.cc:26:14: error: call to deleted constructor of 'Fraction'
20     Fraction f1 {};
21           ^  ~~
22 prog.cc:10:5: note: 'Fraction' has been explicitly marked deleted
   here
23     Fraction() = delete;
24     ^
25 1 error generated.
```



Pour ou contre

Le choix de proposer ou non un constructeur par défaut **dépend de vos choix de conception**. Dans certains cas, ça a du sens, comme une chaîne de caractères vide. Dans d'autres pas forcément. Nous aborderons de nouveau ce point plus tard.

V.4.4. Soyons explicites

Pour pouvoir obtenir le nombre réel correspondant à la division de deux entiers, nous avons utilisé l'opérateur `static_cast`, qui permet de faire une **conversion explicite** d'un type vers un autre, ici de `int` vers `double`. Mais qui dit explicite dit souvent implicite. En effet, C++ permet de nombreuses **conversions implicites**. C'est par exemple le cas dans la ligne suivante, où un caractère est implicitement converti en entier.

```
1 int const entier { 'A' };
```

De telles conversions peuvent aussi se produire lors de l'appel d'un constructeur à un argument (ou plusieurs, mais avec des valeurs par défaut). Parfois, ça nous rend service, comme lorsqu'on manipule des chaînes de caractères avec `std::string`.

```
1 #include <string>
2
3 class Mot
4 {
5 public:
6     Mot(std::string const & mot);
7 private:
8     std::string m_mot;
9 };
10
11 Mot::Mot(std::string const & mot)
12     : m_mot(mot)
13 {
14 }
15
16 using namespace std::string_literals;
17
18 int main()
19 {
20     // Conversion implicite d'une chaîne de caractères C en
21     // std::string au moment de l'appel au constructeur.
22     Mot const arbre { "arbre" };
23
24     // Pas de conversion car argument de type std::string.
25     Mot const agrume { "agrume"s };
26
27     return 0;
28 }
```

L'exemple ci-dessus montre que les conversions implicites peuvent être utiles et même désirées. Ici, l'utilisateur peut passer au choix un littéral comme en C, ou bien un littéral de type `std::string`. On lui offre un plus grand choix, une plus grande liberté, sans rendre le code plus compliqué ou moins clair.

Mais d'autres fois, on veut empêcher cette conversion implicite. Imaginez par exemple une classe représentant un colis. En fonction de son poids et de plein d'autres paramètres, comme l'adresse, le mode de livraison, etc, le prix à payer varie. Modélisons tout ça (le strict minimum).

```

1  #include <cassert>
2  #include <iostream>
3
4  enum class ModeLivraison
5  {
6      Standard,
7      Express,
8      Urgent,
9      Economique
10 };
11
12 class Adresse
13 {
14     // Juste pour l'exemple.
15 };
16
17 class Colis
18 {
19 public:
20     Colis(double poids);
21 private:
22     double m_poids { 0. };
23 };
24
25 Colis::Colis(double poids)
26     : m_poids(poids)
27 {
28     assert(poids >= 0.1 &&
29         "Le poids ne peut pas être inférieur à 100 g.");
30 }
31
32 double calcul_prix(Colis const & colis, Adresse const & adresse,
33     ModeLivraison mode)
34 {
35     // Plein d'opérations complexes pour calculer le prix à payer
36     // pour le colis.
37     return 10;
38 }
39
40 int main()
41 {
42     Adresse adresse_livraison {};
43     std::cout << "Voici le prix à payer : " << calcul_prix(4,
44         adresse_livraison, ModeLivraison::Express) << "\n";
45     return 0;
46 }

```

```
1 Voici le prix à payer : 10
```

Notez l'appel à la fonction libre `calcul_prix(4, adresse_livraison, ModeLivraison::Express)`. C'est ici que se niche le problème. En effet, le premier argument est sensé être une référence sur un objet `Colis const`, mais pourtant **le compilateur ne bronche pas qu'on lui ait donné un entier**. Normal, il fait appel à un **constructeur de conversion**.

Dans notre cas, la conversion implicite qu'il fait d'un entier vers un `Colis` nuit à la lisibilité du code. On ne comprend pas tout de suite à quoi on a affaire en voyant ce littéral `4`. Mieux vaut interdire la conversion implicite, ce qui se fait en utilisant le mot-clé `explicit` devant le prototype du constructeur (attention, il est absent de devant l'implémentation).

```
1 explicit Colis(double poids);
```

Maintenant, le compilateur va nous signaler que passer un entier est invalide.

```
1 [Visual Studio]
2 Erreur C2664 'double calcul_prix(const Colis &,const Adresse
   &,ModeLivraison)' : impossible de convertir l'argument 1 de
   'int' en 'const Colis &'
3 Erreur (active) E0415 il n'existe aucun constructeur approprié pour
   la conversion de "int" en "Colis"
4
5 -----
6
7 [GCC]
8 main.cpp: In function 'int main()':
9 main.cpp:41:61: error: invalid initialization of reference of type
   'const Colis&' from expression of type 'int'
10    41 |         std::cout << "Voici le prix à payer : " <<
       |         calcul_prix(4, adresse_livraison, ModeLivraison::Express) <<
       |         "\n";
11    ^
12 main.cpp:32:34: note: in passing argument 1 of 'double
   calcul_prix(const Colis&, const Adresse&, ModeLivraison)'
13    32 | double calcul_prix(Colis const & colis, Adresse const &
       |         ^~~~~~
       |         ~~~~~~^~~~~~
14
15 -----
16
17 [Clang]
18 prog.cc:41:49: error: no matching function for call to
   'calcul_prix'
```

```

20     std::cout << "Voici le prix à payer : " << calcul_prix(4,
21         adresse_livraison, Modelivraison::Express) << "\n";
22     prog.cc:32:8: note: candidate function not viable: no known
23         conversion from 'int' to 'const Colis' for 1st argument
24     double calcul_prix(Colis const & colis, Adresse const & adresse,
25         Modelivraison mode)
26         ^
27 1 error generated.

```



explicit ou pas?

Là encore, le choix de déclarer un constructeur comme étant explicite ne dépend que de vous et de vos choix de conception. On recommande souvent de **définir comme `explicit` tout constructeur prenant un seul argument** (ce qui inclut ceux ayant plusieurs paramètres optionnels), car les conversions implicites peuvent nuire à la compréhension du code.

Néanmoins, dans certains cas, elles sont appréciables. Par exemple, on peut très bien passer un entier `5` à une fonction attendant un nombre complexe (de type `std::complex` [↗](#)) car un entier est un cas particulier de nombre complexe (avec une partie imaginaire nulle). Il faut donc **peser à chaque fois le pour et le contre**.

V.4.5. En toute amitié

Avez-vous noté que, depuis le début de ce chapitre, j'ai volontairement enlevé la fonction `pow`? C'est que, maintenant que nous avons encapsulé nos attributs, **nous n'y avons plus accès en dehors de la classe**. Comme `pow` est une fonction libre et ne fait donc pas partie de la classe `Fraction`, elle ne peut plus accéder aux attributs `m_numérateur` et `m_dénominateur`.

```

1  #include <cassert>
2  #include <cmath>
3  #include <iostream>
4  #include <numeric>
5
6  class Fraction
7  {
8  public:
9      Fraction(int numérateur, int dénominateur) noexcept;
10     double valeur_reelle() const noexcept;
11     void simplification() noexcept;
12     Fraction pow(Fraction const & fraction, int puissance);
13
14 private:
15     int m_numérateur;

```



```

16     int m_denominateur;
17 };
18
19 Fraction::Fraction(int numerateur, int denominateur) noexcept
20     : m_numerateur(numerateur), m_denominateur(denominateur)
21 {
22     assert(m_denominateur != 0 &&
23           "Le dénominateur ne peut pas valoir 0.");
24 }
25
26 double Fraction::valeur_reelle() const noexcept
27 {
28     return static_cast<double>(m_numerateur) / m_denominateur;
29 }
30
31 void Fraction::simplification() noexcept
32 {
33     int const pgcd { std::gcd(m_numerateur, m_denominateur) };
34     m_numerateur /= pgcd;
35     m_denominateur /= pgcd;
36 }
37
38 Fraction pow(Fraction const & fraction, int puissance)
39 {
40     Fraction copie { fraction };
41     copie.m_numerateur = std::pow(fraction.m_numerateur,
42                                   puissance);
43     copie.m_denominateur = std::pow(fraction.m_denominateur,
44                                    puissance);
45     return copie;
46 }
47
48 int main()
49 {
50     Fraction const f1 { 5, 2 };
51     Fraction const f2 { pow(f1, 3) };
52     return 0;
53 }

```

```

1 [Visual Studio]
2 Erreur (active)      E0265 membre "Fraction::m_numerateur"
   (déclaré à la ligne 15) est inaccessible
3 Erreur (active)      E0265 membre "Fraction::m_denominateur"
   (déclaré à la ligne 16) est inaccessible
4 Erreur (active)      E0265 membre "Fraction::m_denominateur"
   (déclaré à la ligne 16) est inaccessible
5 Erreur C2248 'Fraction::m_numerateur' : impossible d'accéder à
   private membre déclaré(e) dans la classe 'Fraction'

```

```

6  Erreur C2248 'Fraction::m_numerator' : impossible d'accéder à
   private membre déclaré(e) dans la classe 'Fraction'
7
8  -----
9
10 [GCC]
11 prog.cc: In function 'Fraction pow(const Fraction&, int)':
12 prog.cc:40:11: error: 'int Fraction::m_numerator' is private
   within this context
13     40 |         copie.m_numerator = std::pow(fraction.m_numerator,
        |         ^~~~~~
14         |         ^~~~~~
15 prog.cc:15:9: note: declared private here
16     15 |         int m_numerator;
        |         ^~~~~~
17         |         ^~~~~~
18 prog.cc:40:44: error: 'int Fraction::m_numerator' is private
   within this context
19     40 |         copie.m_numerator = std::pow(fraction.m_numerator,
        |         ^~~~~~
20         |         ^~~~~~
21 prog.cc:15:9: note: declared private here
22     15 |         int m_numerator;
        |         ^~~~~~
23         |         ^~~~~~
24 prog.cc:41:11: error: 'int Fraction::m_denominateur' is private
   within this context
25     41 |         copie.m_denominateur =
        |         ^~~~~~
26         |         ^~~~~~
27 prog.cc:16:9: note: declared private here
28     16 |         int m_denominateur;
        |         ^~~~~~
29         |         ^~~~~~
30 prog.cc:41:46: error: 'int Fraction::m_denominateur' is private
   within this context
31     41 |         copie.m_denominateur =
        |         ^~~~~~
32         |         ^~~~~~
33 prog.cc:16:9: note: declared private here
34     16 |         int m_denominateur;
        |         ^~~~~~
35         |         ^~~~~~
36
37  -----
38
39 [Clang]
40 prog.cc:40:11: error: 'm_numerator' is a private member of
   'Fraction'
41     copie.m_numerator = std::pow(fraction.m_numerator,
        |         ^
42         puissance);
43 prog.cc:15:9: note: declared private here
44     int m_numerator;

```

```
45      ^
46 prog.cc:40:44: error: 'm_numerator' is a private member of
   'Fraction'
47     copie.m_numerator = std::pow(fraction.m_numerator,
   puissance);
48                                ^
49 prog.cc:15:9: note: declared private here
50     int m_numerator;
51     ^
52 prog.cc:41:11: error: 'm_denominateur' is a private member of
   'Fraction'
53     copie.m_denominateur = std::pow(fraction.m_denominateur,
   puissance);
54                                ^
55 prog.cc:16:9: note: declared private here
56     int m_denominateur;
57     ^
58 prog.cc:41:46: error: 'm_denominateur' is a private member of
   'Fraction'
59     copie.m_denominateur = std::pow(fraction.m_denominateur,
   puissance);
60                                ^
61 prog.cc:16:9: note: declared private here
62     int m_denominateur;
63     ^
64
65 4 errors generated.
```

Pour résoudre ce problème, on peut partir sur plusieurs solutions.

1. On peut **rendre les attributs publics**.
2. On peut **ajouter des services `numérateur` et `denominateur`**, qui retournent les attributs respectifs, ainsi que des services **`changer_numérateur` et `changer_denominateur`**
3. On peut utiliser **l'amitié**.

V.4.5.1. Attributs publics

Concernant la première solution, autant pour `m_numerator`, cela ne causerait pas de problème, autant pour `m_denominateur`, nous prendrions **le risque que l'invariant puisse être violé**.

V.4.5.2. Accesseurs

La deuxième solution peut sembler bonne, à première vue. Seulement voilà, il faut pour cela ajouter quatre nouveaux services à notre classe: deux **accesseurs** / ***getters*** (de l'anglais *to*

get, qui veut dire «obtenir»), qui permettent d'obtenir un accès en lecture aux attributs `m_nu`
`merateur` et `m_denominateur`, ainsi que deux **mutateurs** / **setters** (de l'anglais *to set*, qui
signifie «mettre à jour, modifier»), qui permettent eux de les modifier.

Déjà, on sait que plus on écrit de code, plus on a de chance d'**introduire des bugs** par
inadvertance. Et plus une classe contient de services, plus elle sera **difficilement maintenable**
et demandera plus d'efforts pour s'assurer qu'elle est correcte.

De plus, en utilisant cette solution, **on dévoile des détails d'implémentation**, puisque
chaque attribut fait l'objet d'un accesseur et d'un mutateur. On ne pense plus en terme de
comportements, de services, mais de nouveau en simple agrégat de données.

Enfin, les services que l'on est amené à écrire n'ont pas toujours du sens. Prenons l'exemple
d'une classe `Lampe`. On peut imaginer récupérer la puissance de la lampe, c'est un service qui
peut être attendu. Par contre, **que dire d'un mutateur qui permettrait de modifier sa**
puissance? Soit le *setter* ne fait aucune vérification, auquel cas pourquoi ne pas rendre l'attribut
public? Soit on a des préconditions, mais qui auront vraisemblablement déjà été vérifiées dans le
constructeur, ce qui veut dire duplication de code, avec plus de risque de bugs, d'erreurs, etc.

Et parfois, cette solution n'est tout simplement **pas envisageable**. C'est le cas si on a plusieurs
invariants à respecter. Si je rajoute à ma classe `Fraction` l'invariant «La fraction doit toujours
être sous sa forme simplifiée» alors le code suivant ne permet pas de faire ce qu'on veut.

```
1 Fraction f1 { 3, 2 };
2 // On obtient 4/2 qui se simplifie tout de suite en 2/1.
3 f1.setNumerator(4);
4
5 Fraction f2 { 3, 2 };
6 // On obtient 3/3 qui se simplifie tout de suite en 1/1.
7 f2.setDenominateur(3);
8
9 Fraction f3 { 3, 2 };
10 // On obtient 2/3.
11 f3.setNumerator(4);
12 f3.setDenominateur(3);
13
14 Fraction f4 { 3, 2 };
15 // On obtient 4/1.
16 f4.setDenominateur(3);
17 f4.setNumerator(4);
18
19 // Le seul moyen possible de passer à 4/3.
20 f1 = Fraction { 4, 3 };
```

V.4.5.3. La troisième, c'est la bonne

Il nous reste donc la troisième solution, l'amitié. Dire d'une fonction qu'elle est amie de la nôtre
signifie qu'elle a un **droit d'accès privilégié à ladite classe**, tant en lecture qu'en écriture.

Elle fait partie intégrante de l'interface, des services offerts par cette classe.

L'amitié se traduit en C++ avec le mot-clé `friend`, qui s'écrit devant le prototype, que l'on place **à l'intérieur de la classe** et non plus à l'extérieur, comme pour une fonction libre non-amie. Ensuite, il n'y a plus qu'à implémenter la fonction de la même manière qu'avant.

```
1 class Fraction
2 {
3 public:
4     friend Fraction pow(Fraction const & fraction, int puissance);
5 };
6
7 // Aucun problème maintenant.
8 Fraction pow(Fraction const & fraction, int puissance)
9 {
10     Fraction copie { fraction };
11     copie.numerateur = std::pow(fraction.numerateur, puissance);
12     copie.denominateur = std::pow(fraction.denominateur,
13                                   puissance);
13     return copie;
14 }
```

Je le répète, mais les fonctions amies ne sont pas des fonctions externes venues «squatter» notre classe. Bien au contraire, elles sont les services que rend la classe, au même titre que les fonctions membres. Elles font donc partie de ce qu'on a défini comme l'interface de la classe, tout en gardant «l'aspect syntaxique» d'une fonction libre.

Bien entendu, avec de grands pouvoirs viennent de grandes responsabilités. Nous avons certes un accès privilégié à la classe, mais cela implique que nous devons **respecter ses invariants**, comme n'importe quelle autre fonction membre.

V.4.6. Exercices

V.4.6.1. Un autre constructeur pour la classe `Fraction`

On pourrait très bien vouloir instancier une fraction à partir d'une chaîne de caractères, comme `"35/8"`. Il nous suffit d'écrire un autre constructeur. Demandons-nous simplement quels seront les cas d'erreurs possibles.

- Pas de numérateur.
- Pas de dénominateur.
- Pas de séparateur, ou caractère invalide.
- Trop de séparateur.
- Dénominateur qui vaut 0.

Traduit en code, on obtient ceci.

```

1  int main()
2  {
3      // Doit compiler.
4      Fraction f1 { "4/2" };
5      assert(f1.valeur_reelle() == 2 &&
6             "La valeur réelle de f1 doit être 2.");
7
8      // Doit générer une erreur.
9      //Fraction f2 { "4\2" };
10
11     // Doit générer une erreur.
12     //Fraction f3 { "4//2" };
13
14     // Doit générer une erreur.
15     //Fraction f4 { "4/" };
16
17     // Doit générer une erreur.
18     //Fraction f5 { "/2" };
19
20     // Doit générer une erreur.
21     //Fraction f6 { "4/0" };
22
23     return 0;
24 }

```

👁 Correction

V.4.6.2. Pile

Une [pile](#) est une structure de données qui fonctionne sur le principe du «**dernier arrivé, premier parti**.» On peut lui ajouter ou supprimer des éléments, mais à chaque fois c'est le dernier élément qui est modifié. On ne peut pas directement supprimer le premier élément, par exemple. Il faut d'abord supprimer tous les autres.

On peut par exemple modéliser l'ordre des appels à des fonctions en C++ avec une pile. D'abord, il y aura `main`, puis une autre fonction appelée dans `main`, puis une autre, etc. Une fois qu'une fonction a fini son exécution, on la retire de la pile et on revient à la fonction précédente, jusqu'à ce que la pile soit vide, quand le programme est terminé.

Voici une liste des opérations que nous allons implémenter.

- Ajouter un élément.
- Retirer le dernier élément.
- Obtenir un accès au premier élément.
- Obtenir le nombre total d'éléments.
- Déterminer si la pile est vide ou non.

Avec ces spécifications, nous pouvons maintenant définir un jeu de tests, qui nous servira à tester que notre code est correct et fonctionnel.

```
1  int main()
2  {
3      Pile pile {};
4      assert(pile.est_vide() && "La pile est vide.");
5      assert(pile.taille() == 0 &&
6              "Au début, la pile n'a aucun élément.");
7
8      pile.ajouter(5);
9      pile.ajouter(10);
10     pile.ajouter(15);
11     assert(pile.taille() == 3 &&
12             "Maintenant, la pile a trois éléments.");
13
14     int const premier { pile.premier() };
15     assert(premier == 5 && "Le premier élément vaut 5.");
16     assert(pile.taille() == 3 &&
17             "La pile a toujours trois éléments.");
18
19     pile.supprimer();
20     assert(pile.taille() == 2 &&
21             "Maintenant, la pile n'a plus que deux éléments.");
22     assert(!pile.est_vide() && "La pile n'est pas vide.");
23
24     pile.supprimer();
25     pile.supprimer();
26     assert(pile.est_vide() && "Maintenant la pile est vide.");
27
28     return 0;
29 }
```

👁 Correction

V.4.6.3. En résumé

- Les classes peuvent posséder des invariants, des contrats, qui doivent être respectés.
- Grâce à l'encapsulation et aux différentes visibilités, il est possible de n'exposer que les services que l'on veut, tout en protégeant les invariants de la classe.
- Une convention C++ répandue incite à préfixer les attributs privés avec `m_`.
- Le constructeur permet d'initialiser l'objet, tout en définissant quels seront les invariants de la classe.
- Si aucun constructeur n'est défini, le compilateur implémente un constructeur par défaut.

- On peut initialiser directement des attributs, ce qui permet dans certains cas de se passer du constructeur par défaut.
- On peut demander explicitement au compilateur de supprimer le constructeur par défaut.

Contenu masqué

Contenu masqué n°76 : Correction

Voici une façon de faire parmi d'autres. Notez que je n'utilise pas la liste d'initialisation, car le code est plus complexe qu'une simple affectation et qu'il y a un peu de travail avant d'extraire le numérateur et le dénominateur.

Comme certaines fonctions de la bibliothèque standard n'ont pas la garantie `noexcept` (comme `std::begin` [↗](#)) et que je n'essaie pas de rattraper une potentielle exception, je ne peux pas garantir que mon constructeur est `noexcept`.

```
1 Fraction::Fraction(std::string const & fraction)
2 {
3     // Vérification que la chaîne n'est pas vide.
4     assert(!std::empty(fraction) &&
5           "La chaîne ne peut pas être vide.");
6
7     // Vérification de la présence du séparateur /.
8     auto debut = std::begin(fraction);
9     auto fin = std::end(fraction);
10    assert(std::count(debut, fin, '/') == 1 &&
11          "Il ne doit y avoir qu'un seul séparateur '/'.");
12    auto itérateur = std::find(debut, fin, '/');
13
14    // Récupération du numérateur.
15    std::string const n { debut, debut + itérateur };
16    assert(!std::empty(n) && "Le numérateur doit être renseigné.");
17    m_numérateur = std::stoi(n);
18
19    // Pour sauter le caractère /.
20    ++itérateur;
21    // Récupération du dénominateur.
22    std::string const d { itérateur, fin };
23    assert(!std::empty(d) &&
24          "Le dénominateur doit être renseigné.");
25    m_dénominateur = std::stoi(d);
26    assert(m_dénominateur != 0 &&
27          "Le dénominateur ne peut pas être nul.");
28 }
```

[Retourner au texte.](#)

Contenu masqué n°77 : Correction

En soit, le code n'est pas très compliqué car nous nous basons entièrement sur `std::vector`. Nous nous contentons juste de rajouter des contraintes, celles inhérentes aux piles.

Enfin, sachez que les piles existent déjà en C++, avec `std::stack` [↗](#). Donc, à l'avenir, quand vous utiliserez des piles, ne reprenez pas cet exercice, utilisez la bibliothèque standard. 🍊

Pile.hpp

```
1  #ifndef PILE_HPP
2  #define PILE_HPP
3
4  #include <vector>
5
6  class Pile
7  {
8  public:
9      void ajouter(int valeur);
10     void supprimer();
11     // Pas de noexcept car en interne, std::vector ne fournit
12     // pas la garantie que récupérer un élément ne lancera
13     // pas d'exception.
14     int premier() const;
15     bool est_vide() const noexcept;
16     std::size_t taille() const noexcept;
17
18 private:
19     std::vector<int> m_pile;
20 };
21
22 #endif
```

Pile.cpp

```
1  #include <cassert>
2  #include "Pile.hpp"
3
4  void Pile::ajouter(int valeur)
5  {
6      m_pile.push_back(valeur);
7  }
8
9  void Pile::supprimer()
10 {
11     assert(m_pile.size() > 0 &&
12           "Impossible de supprimer un élément d'une pile vide.");
13     m_pile.pop_back();
14 }
15
16 int Pile::premier() const
17 {
18     assert(m_pile.size() > 0 &&
19           "Impossible de récupérer le premier élément d'une pile vide.");
20     return m_pile[0];
21 }
22
23 bool Pile::est_vide() const noexcept
24 {
25     return m_pile.empty();
26 }
27
28 std::size_t Pile::taille() const noexcept
29 {
30     return m_pile.size();
31 }
```

[Retourner au texte.](#)

V.5. Une classe de grande valeur

Dans le chapitre sur les opérateurs, nous avons dit, sans plus détailler, que ce n'est pas parce que l'on peut surcharger des opérateurs qu'on doit forcément le faire. En effet, le faire n'a de sens que pour les classes à **sémantique de valeur**. Et nous allons voir que ce n'est pas la seule implication qu'a cette sémantique.

Ce chapitre va présenter la **sémantique de valeur** et tout ce qui lui est lié.

V.5.1. Une histoire de sémantique

Depuis le début du cours, nous avons manipulé de nombreux types, comme des entiers, des réels, des chaînes de caractères, des tableaux, etc. Mais vous êtes-vous déjà demandé **pourquoi y'en a t-il autant**? Puisqu'au final, le processeur ne travaille que sur des nombres, on pourrait se dire que c'est un peu inutile. Pourtant, s'il existe tant de types différents, c'est parce qu'ils ont des **sémantiques différentes**.

La sémantique, c'est **l'étude du sens**. Étudier la sémantique d'un type, c'est donc regarder ce qui a du sens de faire et ce qui n'en a pas. Le type `int` est un type simple, avec une sémantique simple, celle de manipuler un entier. L'exemple suivant est donc parfaitement compilable et exécutable.

```
1 int const temperature { 37 };
2 int const poids { 70 };
3 auto const calcul_idiot { poids + temperature };
```

Pourtant, on sait bien qu'additionner un poids et une température n'a pas de sens. D'ailleurs, quelle unité est-on sensé obtenir? Le compilateur lui ne le sait pas. On lui donne deux entiers et on lui demande de les additionner, il n'y voit aucun problème. Alors que si on définit des types plus spécialisés, avec une sémantique plus précise, on empêche ce comportement.

```
1 Temperature const temperature { 37 };
2 Poids const poids { 70 };
3 // Erreur à la compilation.
4 auto const calcul_idiot { poids + temperature };
```

Définir une sémantique permet donc d'**éviter des bugs**, tout en **rendant le code plus clair**, puisque celui qui vous relit ne se pose pas de question sur vos intentions et sur le sens de votre code. Par exemple, quel code vous semble le plus clair, parmi les deux ci-dessous?

```

1  std::tuple<int, int, int> date_jour()
2  {
3  }
4
5  auto const date { date_jour() };
6  std::cout << "Aujourd'hui, nous sommes le " << std::get<0>(date) <<
    "/" << std::get<1>(date) << "/" << std::get<2>(date) << "\n";
7
8  // -----
9  // VS
10 // -----
11
12 struct Date
13 {
14     int jour;
15     int mois;
16     int annee;
17 }
18
19 Date date_jour()
20 {
21 }
22
23 auto const date { date_jour() };
24 std::cout << "Aujourd'hui, nous sommes le " << date.jour << "/" <<
    date.mois << "/" << date.annee << "\n";

```

Dans le premier, il faut se souvenir constamment de l'ordre des éléments du tuple: est-ce que c'est jour, mois puis année, ou l'inverse? Dans le deuxième, tout est transparent et clair, on sait directement qu'on manipule une date. Le code est donc meilleur.

Ce qu'il convient de retenir, c'est que **définir la sémantique d'un type que l'on crée a des conséquences** sur son utilisation, car certaines opérations auront du sens et d'autres non. Créer des objets, on sait faire. Maintenant, nous allons voir plusieurs sémantiques possibles et ce qu'elles vont impliquer dans le design de nos classes.

V.5.2. La sémantique de valeur, c'est quoi?

On dit d'une classe qu'elle a une sémantique de valeur si **deux objets au contenu identique peuvent être considérés comme égaux**. Par exemple, une classe modélisant une fraction est une classe à sémantique de valeur. Si deux fractions valent $\frac{1}{2}$ chacune, elles sont égales. On a bien deux objets en mémoire, mais leur valeur est égale et donc on considère que ces objets sont égaux.

i

À retenir

Si deux objets, ayant la même valeur, peuvent être considérés comme égaux, on est typiquement dans de la sémantique de valeur.

On peut aussi soustraire plusieurs valeurs entre elles, les multiplier, les diviser, etc. D'où l'intérêt de la surcharge d'opérateurs, qui nous permet d'écrire ça sous forme élégante.

i

À retenir

La surcharge d'opérateurs arithmétiques prend tout son sens dans le cas de la sémantique de valeur.

Prenons un autre exemple, une somme d'argent. C'est bien une valeur. Mais que mes 100€ soient sous forme de liquide, de chèque ou d'une ligne sur mon compte en banque, c'est toujours la même valeur. Je peux ajouter une somme d'argent à une autre et dans ce cas, j'obtiens une nouvelle valeur, une nouvelle somme.

i

À retenir

Modifier une valeur revient à en créer une nouvelle.

V.5.3. Égalité

Commençons par le premier point caractéristiques des valeurs, **l'égalité**. Derrière ce mot simple se cachent des concepts plus complexes, qu'on aborde notamment en [théorie des langages](#) [↗](#). En C++, pour être en mesure de définir l'égalité, on doit respecter les conditions suivantes.

- Pour n'importe quel objet **a**, la comparaison **a == a** est toujours vraie. C'est ce qu'on appelle la **réflexivité**.
- Pour deux objets **a** et **b** de même type, si **a == b** alors **b == a**. La comparaison doit être **commutative** et **symétrique**.
- Pour des objets **a**, **b** et **c** de même type, si on a **a == b** et **b == c**, alors on doit avoir **a == c**. On appelle ça la **transitivité**.

Dans le chapitre 6 de la partie II, nous avons découvert les opérateurs de comparaison **==**, **!=**, **>**, **<**, **>=** et **<=**. C'est sur eux que nous allons nous pencher de nouveau maintenant.

V.5.3.1. Libre ou membre ?

Nous avons appris à les écrire sous **forme libre**, mais ces opérateurs existent aussi sous la forme de **fonctions membres**. C'est ce qu'on apprend en [lisant la documentation](#) [↗](#). On peut donc se demander laquelle choisir?

```

1  class Classe
2  {
3      bool operator==(Classe a);
4  };
5
6  Classe a {};
7  Classe b {};
8  a == b; // Appel de == membre.
9
10 class Classe
11 {
12 };
13
14 bool operator==(Classe a, Classe b);
15
16 Classe a {};
17 Classe b {};
18 a == b; // Appel de == libre.

```

Pourtant, entre les deux formes, il y a une grande différence. Dans le premier cas, l'appel `a == b` va être remplacé par `a.operator==(b)`. Cela signifie que le premier paramètre sera toujours du même type que la classe. Dans le deuxième cas, ce n'est pas obligatoire, car il peut y avoir une conversion implicite réalisée.

Imaginons que nous voulions comparer une fraction avec un entier. Cette opération a du sens, puisqu'un entier est un cas particulier de fraction, avec le dénominateur valant 1. Avec la forme membre, l'instruction `entier == fraction` ne compile pas, car le premier argument doit obligatoirement être une instance de `Fraction`.

```

1  #include <cassert>
2  #include <iostream>
3
4  class Fraction
5  {
6  public:
7      // Constructeur par conversion car un seul paramètre
8      // obligatoire.
9      Fraction(int numerateur, int denominateur = 1) noexcept;
10
11     // Version membre.
12     bool operator==(Fraction const & fraction) const noexcept;
13
14 private:
15     int m_numerateur { 0 };
16     int m_denominateur { 1 };
17 };

```

```

18 Fraction::Fraction(int numérateur, int dénominateur) noexcept
19     : m_numérateur(numérateur), m_dénominateur(dénominateur)
20 {
21     assert(dénominateur != 0 &&
22            "Le dénominateur ne peut pas valoir 0.");
23 }
24 // Implémentation de la version membre.
25 bool Fraction::operator==(Fraction const & fraction) const noexcept
26 {
27     return m_numérateur == fraction.m_numérateur && m_dénominateur
28            == fraction.m_dénominateur;
29 }
30 int main()
31 {
32     Fraction const fraction { 4, 2 };
33     int const entier { 2 };
34
35     if (entier == fraction)
36     {
37         std::cout << "4/2 est égal à 2.\n";
38     }
39     else
40     {
41         std::cout << "4/2 est différent de 2.\n";
42     }
43
44     return 0;
45 }

```

```

1 [Visual Studio]
2 Erreur (active)          E0349 aucun opérateur "==" ne correspond à
   ces opérandes
3 Erreur                  C2677 '==' binaire : aucun opérateur global trouvé
   qui accepte le type 'const Fraction' (ou il n'existe aucune
   conversion acceptable)
4
5 -----
6
7 [GCC]
8 main.cpp: In function 'int main()':
9 main.cpp:34:16: error: no match for 'operator==' (operand types are
   'const int' and 'const Fraction')
10    34 |         if (entier == fraction)
11        |             ~~~~~ ^~ ~~~~~
12        |             |         |
13        |             const int const Fraction

```

```

14
15 -----
16
17 [Clang]
18 prog.cc:34:16: error: invalid operands to binary expression ('const
    int' and 'const Fraction')
19     if (entier == fraction)
20         ~~~~~ ^ ~~~~~
21 1 error generated.

```

Avec la version libre, le problème ne se pose plus, puisqu'il y a conversion implicite de l'entier en fraction. Notez au passage l'utilisation de l'amitié pour surcharger l'opérateur `==` libre.

```

1  #include <cassert>
2  #include <iostream>
3
4  class Fraction
5  {
6  public:
7      // Constructeur par conversion car un seul paramètre
        obligatoire.
8      Fraction(int numérateur, int dénominateur = 1) noexcept;
9
10     // Version libre amie.
11     friend bool operator==(Fraction const & lhs, Fraction const &
        rhs);
12
13 private:
14     int m_numérateur { 0 };
15     int m_dénominateur { 1 };
16 };
17
18 Fraction::Fraction(int numérateur, int dénominateur) noexcept
19     : m_numérateur(numérateur), m_dénominateur(dénominateur)
20 {
21     assert(dénominateur != 0 &&
22         "Le dénominateur ne peut pas valoir 0.");
23 }
24
25 bool operator==(Fraction const & lhs, Fraction const & rhs)
26 {
27     return lhs.m_numérateur == rhs.m_numérateur &&
28         lhs.m_dénominateur == rhs.m_dénominateur;
29 }
30
31 int main()
32 {
33     Fraction const fraction { 4, 2 };

```



```

32     int const entier { 2 };
33
34     if (entier == fraction)
35     {
36         std::cout << "4/2 est égal à 2.\n";
37     }
38     else
39     {
40         std::cout << "4/2 est différent de 2.\n";
41     }
42
43     return 0;
44 }

```

```

1 4/2 est égal à 2.

```

?

Quelle version dois-je utiliser? Laquelle est la meilleure?

Il faut en effet en choisir une des deux, car il est **impossible d'utiliser les deux en même temps**. En effet, en écrivant `fraction == autre_fraction`, le compilateur ne sait pas si vous cherchez à utiliser la version libre ou la version membre. Cela se voit dans les erreurs qu'on obtient.

```

1 [Visual Studio]
2 Erreur (active)          E0350 plusieurs opérateurs "=="
   correspondent à ces opérandes
3 Erreur                  C2593 'operator ==' est ambigu
4
5 -----
6
7 [GCC]
8 main.cpp: In function 'int main()':
9 main.cpp:42:18: error: ambiguous overload for 'operator==' (operand
   types are 'const Fraction' and 'const Fraction')
10    42 |         if (fraction == autre_fraction)
11        |             ~~~~~ ^~ ~~~~~
12        |             |           |
13        |             |           const Fraction
14        |             const Fraction
15 main.cpp:27:6: note: candidate: 'bool Fraction::operator==(const
   Fraction&) const'
16    27 | bool Fraction::operator==(Fraction const & fraction) const
   noexcept
17        |             ^~~~~~

```

```

18 main.cpp:32:6: note: candidate: 'bool operator==(const Fraction&,
    const Fraction&)'
19     32 | bool operator==(Fraction const & lhs, Fraction const & rhs)
20         |          ^~~~~~
21
22 -----
23
24 [Clang]
25 prog.cc:42:18: error: use of overloaded operator '==' is ambiguous
    (with operand types 'const Fraction' and 'const Fraction')
26     if (fraction == autre_fraction)
27         ~~~~~ ^ ~~~~~
28 prog.cc:27:16: note: candidate function
29 bool Fraction::operator==(Fraction const & fraction) const noexcept
30         ^
31 prog.cc:32:6: note: candidate function
32 bool operator==(Fraction const & lhs, Fraction const & rhs)
33     ^
34 1 error generated.

```

La forme la plus pratique et la plus simple reste la forme libre, éventuellement **friend** si besoin d'accéder à des attributs privés. La forme membre est permise, mais est plus limitative.

V.5.3.2. Soyons intelligents, réutilisons

Ce conseil vous a déjà été donné lors de la découverte de la surcharge d'opérateurs, mais il est bon de se le rappeler. En effet, réutiliser le code permet de réduire le risque de bugs, évite la duplication, rend le code plus léger. L'habitude que beaucoup prennent est de définir les opérateurs `==` et `<`, puis de définir les autres en fonction de ces deux-là.

Dans le cas de notre classe **Fraction**, rien de bien compliqué, il suffit de reprendre ce que nous avons déjà écrit il y a quelques temps.

```

1  #include <cassert>
2
3  class Fraction
4  {
5  public:
6      Fraction(int numérateur, int dénominateur) noexcept;
7
8      // Utilisation de l'amitié car == et < ont besoin d'accéder aux
        attributs.
9      // Notez que si la classe fournit les services 'numérateur()'
        et 'dénumérateur()', alors l'amitié n'est plus nécessaire.
10     friend bool operator==(Fraction const & lhs, Fraction const &
        rhs) noexcept;

```

```

11     friend bool operator<(Fraction const & lhs, Fraction const &
12         rhs) noexcept;
13 private:
14     int m_numérateur { 0 };
15     int m_dénominateur { 1 };
16 };
17
18 Fraction::Fraction(int numérateur, int dénominateur) noexcept
19     : m_numérateur(numérateur), m_dénominateur(dénominateur)
20 {
21     assert(dénominateur != 0 &&
22         "Le dénominateur ne peut pas valoir 0.");
23 }
24 // Opérateurs d'égalité.
25
26 bool operator==(Fraction const & lhs, Fraction const & rhs)
27     noexcept
28 {
29     return lhs.m_numérateur == rhs.m_numérateur &&
30         lhs.m_dénominateur == rhs.m_dénominateur;
31 }
32
33 bool operator!=(Fraction const & lhs, Fraction const & rhs)
34 {
35     return !(lhs == rhs);
36 }
37
38 // Opérateurs de comparaisons.
39
40 bool operator<(Fraction const & lhs, Fraction const & rhs) noexcept
41 {
42     if (lhs.m_numérateur == rhs.m_numérateur)
43     {
44         return lhs.m_dénominateur > rhs.m_dénominateur;
45     }
46     else if (lhs.m_dénominateur == rhs.m_dénominateur)
47     {
48         return lhs.m_numérateur < rhs.m_numérateur;
49     }
50     else
51     {
52         Fraction const gauche { lhs.m_numérateur *
53             rhs.m_dénominateur, lhs.m_dénominateur *
54             rhs.m_dénominateur };
55         Fraction const droite { rhs.m_numérateur *
56             lhs.m_dénominateur, rhs.m_dénominateur *
57             lhs.m_dénominateur };
58         return gauche.m_numérateur < droite.m_numérateur;
59     }
60 }

```

```

53     }
54 }
55
56 // Pour simplifier, il suffit de vérifier que lhs n'est ni
    inférieur, ni égal à rhs.
57 bool operator>(Fraction const & lhs, Fraction const & rhs)
58 {
59     return !(lhs < rhs) && lhs != rhs;
60 }
61
62 // Pour simplifier, il suffit de vérifier que lhs est inférieur ou
    égal à rhs.
63 bool operator<=(Fraction const & lhs, Fraction const & rhs)
64 {
65     return lhs < rhs || lhs == rhs;
66 }
67
68 // Pour simplifier, il suffit de vérifier que lhs est supérieur ou
    égal à rhs.
69 bool operator>=(Fraction const & lhs, Fraction const & rhs)
70 {
71     return lhs > rhs || lhs == rhs;
72 }

```

En fait, dès que l'on définit un opérateur d'égalité ou de comparaison, il fait sens de **définir les autres**. Si vous fournissez `<` et `>`, l'utilisateur s'attend aussi à pouvoir utiliser `<=` et `>=`. Heureusement, comme on l'a vu, on peut définir les autres en fonction du premier, ce qui fait gagner du temps et réduit la possibilité de bugs.

V.5.4. Le retour des opérateurs

Dans le chapitre 6 de la deuxième partie, nous avons vu une première introduction à la surcharge d'opérateur, ce qui nous a permis de définir l'addition, la soustraction, etc. Maintenant, nous allons en aborder d'autres, grâce à nos nouvelles connaissances sur les objets.



Sous-ensemble

La surcharge d'opération n'est pas une caractéristique essentielle des classes à sémantique de valeur, mais plutôt un **sous-ensemble** qu'on rencontre souvent, mais pas systématiquement.

V.5.4.1. Les opérateurs d'affectation intégrés

Sous ce terme se cachent les opérateurs `+=`, `-=`, `*=`, `/=` et `%=`, qui permettent les raccourcis du type `f1 += f2`, ou bien encore `f3 -= f4`. On les appelle en anglais les *compound assignment*

operator.

Notez que, comme pour des types natifs, ils modifient l'instance qui les appelle. La forme canonique voudra donc qu'on les écrive sous la forme membre. Voici à quoi l'opérateur `+=` de la classe `Fraction` doit ressembler. La forme est la même pour tous les autres, hormis le signe.

```
1 class Fraction
2 {
3 public:
4     Fraction& operator+=(Fraction const & fraction) noexcept;
5 };
6
7 Fraction& Fraction::operator+=(Fraction const & fraction) noexcept
8 {
9     // Implémentation de l'addition.
10    return *this;
11 }
```

Normalement, l'instruction `return *this` devrait vous surprendre. En effet, nous ne l'avons jamais rencontrée encore. De plus, la fonction renvoie une référence. Or, nous avons vu que cela peut être dangereux dans les cas où celle-ci pointerait sur un objet qui n'existe plus. Que de mystères...

L'explication est simple. L'expression `return *this` permet de **renvoyer l'objet courant**. Sans cela, il serait impossible d'écrire quelque chose comme ce qui suit.

```
1 Fraction const f1 { 5, 4 };
2 Fraction f2 { 1, 1 };
3 Fraction f3 { 1, 1 };
4
5 // On additionne f1 à f2, puis on soustrait ce nouveau f2 à f3.
6 f3 -= f2 += f1;
```

Comme la ligne `f3 -= f2 += f1` revient à écrire `f3.operator-=(f2.operator+=(f1))`, si la fonction ne renvoyait rien, nous ne pourrions pas **chaîner ainsi les modifications** (bien que cet exemple soit très peu lisible). Il faut donc renvoyer l'objet après modification. Comme celui-ci est toujours valide, aucun risque à renvoyer une référence sur cet objet, ce qui est moins long qu'une recopie.

Cela nous permet d'avoir le même comportement pour notre objet que s'il s'agissait d'un type natif comme `int` (pas de surprise), tout en gardant une compatibilité / similitude avec le langage C.

Le reste de l'implémentation ne devrait pas vous surprendre, il ne s'agit que d'une simple addition de fractions.

```

1  class Fraction
2  {
3  public:
4      Fraction& operator+=(Fraction const & fraction) noexcept;
5
6  private:
7      int m_numerateur { 0 };
8      int m_denominateur { 1 };
9  };
10
11 Fraction& Fraction::operator+=(Fraction const & fraction) noexcept
12 {
13     m_numerateur = m_numerateur * fraction.m_denominateur +
14                     fraction.m_numerateur * m_denominateur;
15     m_denominateur *= fraction.m_denominateur;
16     return *this;
17 }

```



Exercice

Je vous invite à implémenter de la même manière `-=`, `*=` et `/=` pour vous entraîner.

V.5.4.2. Du deux en un

Pour chaque opérateur intégré, il existe l'équivalent binaire, qui prend deux arguments et renvoie une copie. Ainsi, pour `+=`, il existe `+`, pour `-=` il existe `-`, etc. Logiquement, si l'on fournit l'une des deux formes, on s'attend à l'autre. Dans un esprit de réutilisation du code déjà écrit et validé, on va en **implémenter un sous forme de l'autre**.

L'usage en C++ est d'écrire l'implémentation dans l'opérateur intégré, puis de définir l'opérateur libre associé en fonction du premier. Dans le cas de la classe `Fraction`, cela veut dire écrire la logique de l'addition dans `operator+=` puis d'appeler celui-ci dans `operator+`. Voyez par vous-mêmes l'exemple suivant.

```

1  class Fraction
2  {
3  public:
4      Fraction& operator+=(Fraction const & fraction) noexcept;
5
6  private:
7      int m_numerateur { 0 };
8      int m_denominateur { 1 };
9  };
10

```

```

11 Fraction& Fraction::operator+=(Fraction const & fraction) noexcept
12 {
13     m_numerator = m_numerator * fraction.m_denominateur +
14         fraction.m_numerator * m_denominateur;
15     m_denominateur *= fraction.m_denominateur;
16     return *this;
17 }
18 // L'argument lhs est recopié, donc on peut le modifier sans
19 // problème.
20 Fraction operator+(Fraction lhs, Fraction const & rhs) noexcept
21 {
22     lhs += rhs;
23     return lhs;
24 }

```

On utilise le **principe de la recopie**. Puisque le paramètre `lhs` est une copie, on peut le modifier en utilisant `+=` sans risque. On obtient un opérateur `+` correct et non redondant avec `+=`.



Exercice

Je vous invite à implémenter de la même manière `-`, `*` et `/` pour vous entraîner.

V.5.4.3. Les opérateurs de manipulation des flux

Ce sont eux qui nous permettent d'afficher un objet sur la sortie standard, ou d'en créer un depuis un fichier, par exemple. En [lisant la documentation](#) [↗](#), on note que leur forme canonique est la suivante.

```

1 std::ostream& operator<<(std::ostream & os, T const & obj)
2 {
3     // Écriture sur le flux de sortie.
4     return os;
5 }
6
7 std::istream& operator>>(std::istream & is, T & obj)
8 {
9     // Lecture depuis le flux d'entrée.
10    if( /* Erreur, impossible de construire T. */ )
11    {
12        is.setstate(std::ios::failbit);
13    }
14    return is;
15 }

```

Ces opérateurs s'écrivent toujours sous la forme libre car **leur premier argument est toujours un flux**. Ils sont souvent `friend`, car ils doivent accéder aux attributs de l'instance. Enfin, notez qu'on signale que l'entrée est invalide **en mettant le flux dans un état invalide** avec `std::ios::failbit` [↗](#), ce qui permet à l'utilisateur de faire `if (std::cin.fail())`, comme nous l'avons appris lors de l'écriture de la fonction de saisie sécurisée.

L'exemple suivant nous montre une implémentation possible pour la classe `Fraction`. Notez que j'ai extrait dans une nouvelle fonction l'extraction des données depuis une chaîne de caractères. Ça permet de gérer de plusieurs manières les erreurs et de ne pas dupliquer la logique et le code entre le constructeur et `operator>`.

Fraction.hpp

```
1  #ifndef FRACTION_HPP
2  #define FRACTION_HPP
3
4  #include <istream>
5  #include <ostream>
6  #include <string>
7
8  class Fraction
9  {
10 public:
11     Fraction(int numérateur, int dénominateur);
12     Fraction(std::string const & chaîne);
13
14     friend std::ostream& operator<<(std::ostream & os,
15                                     Fraction const & fraction);
16     friend std::istream& operator>>(std::istream & is,
17                                     Fraction & fraction);
18
19 private:
20     int m_numérateur { 1 };
21     int m_dénominateur { 1 };
22 };
23
24 #endif
```


Fraction.cpp

```

1  #include <algorithm>
2  #include <cassert>
3  #include <exception>
4  #include "Fraction.hpp"
5
6  /**
7   * @brief Extrait le numérateur et le dénominateur d'une
8   *        chaîne, si possible.
9   * @param[in] chaine La chaîne de caractères à analyser.
10  * @param[out] numerateur Le numérateur extrait.
11  * @param[out] denominateur Le dénominateur extrait.
12  * @exception Lance une std::invalid_argument si la chaîne
13  *        n'est pas correcte.
14  */
15 void extraction_depuis_chaine(std::string const & chaine, int
16     & numerateur, int & denominateur)
17 {
18     if (std::empty(chaine))
19     {
20         throw
21             std::invalid_argument("La chaîne ne peut pas être vide.");
22     }
23
24     auto const debut = std::begin(chaine);
25     auto const fin = std::end(chaine);
26     if (std::count(debut, fin, '/') != 1)
27     {
28         throw
29             std::invalid_argument("La chaîne ne peut contenir qu'un seul sép
30     }
31     auto itérateur = std::find(debut, fin, '/');
32
33     // Récupération du numérateur.
34     std::string const n { debut, debut + std::distance(debut,
35         itérateur) };
36     if (std::empty(n))
37     {
38         throw
39             std::invalid_argument("Le numérateur ne peut pas être vide.");
40     }
41
42     // Pour sauter le caractère /.
43     ++itérateur;
44     // Récupération du dénominateur.
45     std::string const d { itérateur, fin };
46     if (std::empty(d) || std::stoi(d) == 0)
47     {
48         throw
49             std::invalid_argument("La dénominateur ne peut pas être vide ou
50     }
51
52     // Mise à jour uniquement si tout c'est bien passé.

```

main.cpp

```

1  #include <iostream>
2  #include "Fraction.hpp"
3
4  template<typename T>
5  void entree_securisee(T & variable)
6  {
7      while (!(std::cin >> variable))
8      {
9          if (std::cin.eof())
10         {
11             break;
12         }
13         else if (std::cin.fail())
14         {
15             std::cout << "Entrée invalide. Recommence." <<
16                 std::endl;
17             std::cin.clear();
18
19             std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
20                 '\n');
21         }
22         else
23         {
24             std::cout << "Le prédicat n'est pas respecté !" <<
25                 std::endl;
26         }
27     }
28 }
29
30 int main()
31 {
32     Fraction const f1 { "5/2" };
33     std::cout << f1 << "\n";
34
35     Fraction f2 { 1, 1 };
36     // Grâce à notre écriture de operator>> conforme aux
37     // normes, on peut
38     // réutiliser la fonction entree_securisee sans plus
39     // d'effort.
40     entree_securisee(f2);
41     std::cout << f2 << "\n";
42     return 0;
43 }

```

V.5.4.4. Quelques bonnes pratiques

Finissons cette section en détaillant trois bonnes pratiques que vous devez garder à l'esprit chaque fois que vous surchargerez des opérateurs. Comme d'habitude, ces principes n'ont rien d'obligatoire, mais ils rendront votre code de meilleure qualité.

D'abord, **respectez toujours la sémantique d'un opérateur**. Si vous utilisez l'opérateur `+`, ne calculez pas une division. C++ vous y autorise, mais les utilisateurs de votre code risquent fort de ne pas apprécier. Si un opérateur a une sémantique déjà définie pour un domaine, tenez vous-y. Que votre `+` signifie toujours `+`!

Ensuite, **si la signification d'un opérateur n'est pas claire et indiscutable, vous ne devriez pas l'utiliser**. Par exemple, bien qu'on puisse surcharger l'opérateur `*`, ne vous en servez pas pour retourner l'inverse d'une fraction, ça surprendrait tout le monde car personne ne s'attend à ce que cet opérateur retourne un inverse. Si la signification de l'opérateur n'est pas clairement comprise et largement acceptée pour l'opération que vous voulez réaliser, mieux vaut **utiliser une fonction bien nommée** à la place.



Point d'histoire

Par exemple, le choix de `<` et `>` pour les flux d'entrée ne respecte pas ce principe, car cet opérateur n'a pas la même sémantique ici que celle qu'il a en C. Aujourd'hui, tout le monde est habitué, mais ce n'est pas une raison pour faire de même.

Enfin, certains opérateurs viennent par groupe et **si vous implémentez un opérateur d'une catégorie, implémentez aussi tous les autres de cette catégorie**. Je le répète, mais si vous implémentez `<`, les utilisateurs s'attendront à pouvoir utiliser aussi `>`, `<=` et `>=`. Si vous définissez `+=`, on peut vraisemblablement penser qu'on pourra utiliser aussi `+`.

V.5.4.5. Les autres opérateurs

Faire le tour de tous les opérateurs serait long et peu instructif. En effet, maintenant que nous avons vu plus en détail comment procéder, ainsi que les formes canoniques, **le principe est le même pour tous les autres**. Vous serez donc capables de vous débrouiller seuls avec les opérateurs que nous n'avons pas abordé. 🍊

De plus, hormi les opérateurs `()` et `[]`, ainsi que `++` et `--`, ceux qui restent sont assez rarement surchargés. Dans ce cours, nous ne verrons pas de situation où l'on aurait besoin de surcharger d'autres opérateurs que tous ceux cités, ainsi que `()` et `[]`.

V.5.5. Copier des objets

On peut créer autant de «duplicatas» du même objet, s'il a une sémantique de valeur, ils seront tous égaux. On peut copier, par exemple, notre fraction 1/2 dix fois, à la fin il ne reste que la même valeur. Mais justement, comment on copie les objets en C++? Avec deux choses.

- Le constructeur de copie.

— L'opérateur d'affectation par copie.

V.5.5.1. Constructeur de copie

C'est ce constructeur qui est appelé quand on crée une copie d'une autre instance. C'est lui que nous avons utilisé, sans le savoir, en écrivant la fonction libre `pow` pour la classe `Fraction`.

```
1 Fraction copie { fraction };
```

Par défaut, le compilateur en génère un si l'on ne fait rien. Ce constructeur par recopie s'occupe comme un grand de copier chaque attribut. Dans notre cas, nos attributs sont de simples entiers. On peut donc **laisser le compilateur faire son travail** sans problème. Notez qu'on peut utiliser le mot-clé `default` si l'on veut être plus explicite.

```
1 class Fraction
2 {
3     public:
4         Fraction(Fraction const & fraction) = default;
5 };
```

V.5.5.2. Opérateur d'affectation par copie

Ce que nous venons de voir fonctionne lors de l'instanciation d'une classe, puisque c'est à ce moment précis que le constructeur est appelé. Or, il se peut que l'on veuille **affecter une nouvelle valeur à notre objet en dupliquant celle d'un autre**, alors que celui-là est déjà créé. Pour cela, il faut implémenter l'**opérateur d'affectation par copie**, de manière à rendre compilable le code suivant.

```
1 Fraction const fraction { 3, 4 };
2 Fraction copie { 1, 1 };
3
4 // Du code.
5
6 copie = fraction;
```

Comme pour le constructeur de copie, son implémentation est faite implicitement par le compilateur. On peut là encore choisir de la rendre explicite.

```
1 class Fraction
2 {
3 public:
4     Fraction & operator=(Fraction const & fraction) = default;
5 };
```

Comme pour les opérateurs d’affectation intégrés, cet opérateur renvoie une référence sur l’objet courant, ce qui permet de chaîner les affectations, comme le montre l’exemple ci-dessous.

```
1 Fraction const f1 { 87, 9 };
2 Fraction f2 { 1, 1 };
3 Fraction f3 { 1, 1 };
4
5 f3 = f2 = f1;
```

V.5.5.3. En résumé

- Certaines classes ont une sémantique de valeur.
- L’égalité de deux objets est définie en C++ par des règles précises.
- Le constructeur de copie et l’opérateur d’affectation par recopie peuvent être générés automatiquement par le compilateur, soit implicitement en n’écrivant rien, soit explicitement en utilisant `default`.
- On peut utiliser la surcharge d’opérateurs pour faciliter la manipulation de nos valeurs.
- En réutilisant intelligemment certains opérateurs, on peut écrire le reste plus simplement et de façon plus concise.

V.6. [T.P] Entrons dans la matrice !

Les objets mathématiques se prêtent bien à la sémantique de valeur. Alors, pour ne pas déroger à la tradition, nous allons implémenter un nouveau concept, **les matrices**. Si vous ne connaissez pas, aucun problème, nous allons aborder rapidement le sujet. Sinon, vous n'avez plus qu'à foncer coder. 🍌

Ce chapitre a pour but de nous faire **pratiquer toutes les notions vues précédemment**.

V.6.1. Qu'est-ce qu'une matrice ?

i

Un mot pour les mathématiciens

Veuillez nous pardonner si les explications et notations qui suivent ne sont pas parfaitement rigoureuses, le but étant de faire simple.

On peut définir une matrice de manière simple comme étant un tableau de nombres à n lignes et m colonnes. Ces nombres sont appelés **dimensions** de la matrice. Ci-dessous, vous avez un exemple de matrice de dimension 2×3 .

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 6 \end{pmatrix}$$

Si une matrice n'est composée que d'une ligne, on parle de **vecteur-ligne**. De même, s'il n'y a qu'une colonne, on parle de **vecteur-colonne**.

Comme pour beaucoup d'objets mathématiques, il est possible d'appliquer un certain nombre d'opérations élémentaires, comme l'**addition** ou la **multiplication**. Il existe aussi quelques opérations plus spécifiques aux matrices, comme la **transposition**.

V.6.1.1. Addition

L'addition de deux matrices se fait **terme à terme**. Les deux matrices additionnées doivent être de même dimension.

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 6 \end{pmatrix} + \begin{pmatrix} 0 & 4 & 9 \\ 4 & -7 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 6 & 12 \\ 7 & -3 & 7 \end{pmatrix}$$

Le même principe s'applique à la soustraction.

V.6.1.2. Multiplication par un nombre

Multiplier une matrice par un nombre revient à multiplier chaque terme par ce nombre.

$$2 \times \begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 6 \end{pmatrix} = \begin{pmatrix} 2 & 4 & 6 \\ 6 & 8 & 12 \end{pmatrix}$$

V.6.1.3. Transposition

Transposer une matrice revient à échanger les lignes et les colonnes.

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 6 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 6 \end{pmatrix}$$

V.6.1.4. Multiplication de deux matrices



Pour définir la multiplication dans le cas général, intéressons-nous d'abord à la multiplication d'un vecteur-colonne par un vecteur-ligne, qu'on appelle **produit scalaire**. Pour deux matrices de même dimension, calculer leur produit scalaire revient à faire l'opération suivante.

$$\begin{pmatrix} x_1 & x_2 & \dots & x_n \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = x_1y_1 + x_2y_2 + \dots + x_ny_n = \sum_{i=1}^n x_iy_i$$

De manière plus générale, la multiplication d'une matrice $A(n \times m)$ par une matrice $B(m \times p)$ donne une matrice $C(n \times p)$ dont chaque élément C_{ij} est le produit scalaire de la ligne i de la matrice A avec la colonne j de la matrice B . L'exemple ci-dessous va vous aider à saisir le principe.

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 6 \end{pmatrix} \times \begin{pmatrix} 0 & 4 \\ 4 & -7 \\ 9 & 1 \end{pmatrix} = \begin{pmatrix} 1 \times 0 + 2 \times 4 + 3 \times 9 & 1 \times 4 + 2 \times -7 + 3 \times 1 \\ 3 \times 0 + 4 \times 4 + 6 \times 9 & 3 \times 4 + 4 \times -7 + 6 \times 1 \end{pmatrix} = \begin{pmatrix} 35 & -7 \\ 70 & -10 \end{pmatrix}$$

V.6.1.5. D'autres propriétés

Les matrices sont un vaste sujet et il y aurait encore beaucoup à dire. Vous pouvez continuer votre découverte, si le cœur vous en dit, sur de nombreux sites. Vous pouvez par exemple jeter un œil à [celui-ci](#) , ou bien encore à la page [Wikipédia](#)  associée.


Quant à nous, nous nous contenterons de ces notions dans le cadre du T.P. 🍊

V.6.2. L'énoncé

Comme vous l'avez compris, le but est de pouvoir manipuler des matrices en C++. Mais avant de vous lancer corps et âme dans ce TP, prenons le temps de réfléchir à la conception de notre programme, notamment aux services que l'on souhaite offrir.

V.6.2.1. Les services

Le premier est bien évidemment **la création d'une matrice**. Sans cette partie, pas de TP. Il serait bien de pouvoir créer une matrice de **dimension quelconque**, avec éventuellement tous ses éléments ayant une valeur par défaut, mais il faut aussi pouvoir **la copier**.

Ensuite, on aimerait **l'accès à un élément de la matrice**, tant en lecture qu'en écriture. Cette opération est possible avec `std::vector` ou `std::string` par exemple. Cependant, l'opérateur `[]` n'accepte qu'un seul argument, alors que nous aimerions pouvoir indiquer la ligne et la colonne de l'élément voulu. Heureusement, [il existe un opérateur](#)  qui peut nous aider.

👁 Opérateur à utiliser

Tant qu'on parle des opérateurs, il semble évident que l'on va en implémenter plusieurs autres pour **gérer l'addition et la multiplication**. Vous devez maintenant avoir l'habitude. Néanmoins, n'oubliez pas que l'on veut proposer la multiplication tant par un entier que par une autre matrice.

Enfin, en vrac, proposons de récupérer les **dimensions de la matrice**, de **l'afficher** et puis d'extraire une ligne ou une colonne spécifique, pour en faire respectivement un vecteur-ligne et un vecteur colonne.

V.6.2.2. Détails d'implémentation

Maintenant que nous avons bien fixé nos services, il est temps de se plonger dans des détails d'implémentations. La grande question est de savoir **comment stocker les différents nombres** qui composent une matrice, en n'oubliant pas que nous ne connaissons pas sa dimension à l'avance.

Je vais vous aider en vous indiquant qu'il **suffit d'utiliser un tableau**, de taille `lignes * colonnes`, tout simplement. Une petite astuce vous aidera à accéder directement au bon élément. Essayez de la trouver sans regarder l'indice.

👁 Astuce secrète

V.6.2.3. Les tests

Bien entendu, nous voulons nous assurer que notre code fonctionne bien. Nous avons vu quelques exemples mathématiques pour illustrer chacune des opérations, donc nous pouvons nous baser sur les valeurs obtenues pour écrire quelques tests.

```

1 void test_addition()
2 {
3     Matrice m1 { 2, 3 };
4     m1(0, 0) = 1;
5     m1(0, 1) = 2;
6     m1(0, 2) = 3;
7     m1(1, 0) = 3;
8     m1(1, 1) = 4;
9     m1(1, 2) = 6;
10
11    Matrice m2 { 2, 3 };
12    m2(0, 0) = 0;
13    m2(0, 1) = 4;
14    m2(0, 2) = 9;
15    m2(1, 0) = 4;
16    m2(1, 1) = -7;
17    m2(1, 2) = 1;
18
19    Matrice resultat { 2, 3 };
20    resultat(0, 0) = 1;
21    resultat(0, 1) = 6;
22    resultat(0, 2) = 12;
23    resultat(1, 0) = 7;
24    resultat(1, 1) = -3;
25    resultat(1, 2) = 7;
26
27    Matrice const addition { m1 + m2 };
28
29    assert(addition.nb_lignes() == resultat.nb_lignes() &&
30           "L'addition n'est pas correcte.");
31    assert(addition.nb_colonnes() == resultat.nb_colonnes() &&
32           "L'addition n'est pas correcte.");
33    for (int i { 0 }; i < addition.nb_lignes(); ++i)
34    {
35        for (int j { 0 }; j < addition.nb_colonnes(); ++j)
36        {
37            assert(resultat(i, j) == addition(i, j) &&
38                   "L'addition n'est pas correcte.");
39        }
40    }
41 }

```

```

36     }
37 }
38 }
39
40 void test_multiplication_entier()
41 {
42     Matrice m1 { 2, 3 };
43     m1(0, 0) = 1;
44     m1(0, 1) = 2;
45     m1(0, 2) = 3;
46     m1(1, 0) = 3;
47     m1(1, 1) = 4;
48     m1(1, 2) = 6;
49
50     Matrice resultat { 2, 3 };
51     resultat(0, 0) = 2;
52     resultat(0, 1) = 4;
53     resultat(0, 2) = 6;
54     resultat(1, 0) = 6;
55     resultat(1, 1) = 8;
56     resultat(1, 2) = 12;
57
58     Matrice const multiplication { m1 * 2 };
59
60     assert(resultat.nb_lignes() == multiplication.nb_lignes() &&
61           "La multiplication par un entier n'est pas correcte.");
62     assert(resultat.nb_colonnes() == multiplication.nb_colonnes()
63           && "La multiplication par un entier n'est pas correcte.");
64     for (int i { 0 }; i < multiplication.nb_lignes(); ++i)
65     {
66         for (int j { 0 }; j < multiplication.nb_colonnes(); ++j)
67         {
68             assert(resultat(i, j) == multiplication(i, j) &&
69                   "La multiplication par un entier n'est pas correcte.");
70         }
71     }
72 }
73
74 void test_multiplication_matrice()
75 {
76     Matrice m1 { 2, 3 };
77     m1(0, 0) = 1;
78     m1(0, 1) = 2;
79     m1(0, 2) = 3;
80     m1(1, 0) = 3;
81     m1(1, 1) = 4;
82     m1(1, 2) = 6;
83
84     Matrice m2 { 3, 2 };
85     m2(0, 0) = 0;

```

```

83     m2(0, 1) = 4;
84     m2(1, 0) = 4;
85     m2(1, 1) = -7;
86     m2(2, 0) = 9;
87     m2(2, 1) = 1;
88
89     Matrice resultat { 2, 2 };
90     resultat(0, 0) = 35;
91     resultat(0, 1) = -7;
92     resultat(1, 0) = 70;
93     resultat(1, 1) = -10;
94
95     Matrice const multiplication { m1 * m2 };
96
97     assert(resultat.nb_lignes() == multiplication.nb_lignes() &&
98           "La multiplication par une matrice n'est pas correcte.");
99     assert(resultat.nb_colonnes() == multiplication.nb_colonnes()
100           &&
101           "La multiplication par une matrice n'est pas correcte.");
102     for (int i { 0 }; i < multiplication.nb_lignes(); ++i)
103     {
104         for (int j { 0 }; j < multiplication.nb_colonnes(); ++j)
105         {
106             assert(resultat(i, j) == multiplication(i, j) &&
107                   "La multiplication par une matrice n'est pas correcte.");
108         }
109     }
110 }
111
112 void test_transposition()
113 {
114     Matrice m1 { 2, 3 };
115     m1(0, 0) = 1;
116     m1(0, 1) = 2;
117     m1(0, 2) = 3;
118     m1(1, 0) = 3;
119     m1(1, 1) = 4;
120     m1(1, 2) = 6;
121
122     Matrice resultat { 3, 2 };
123     resultat(0, 0) = 1;
124     resultat(0, 1) = 3;
125     resultat(1, 0) = 2;
126     resultat(1, 1) = 4;
127     resultat(2, 0) = 3;
128     resultat(2, 1) = 6;
129
130     Matrice const transposee { m1.transpose() };

```

```

128     assert(resultat.nb_lignes() == transposee.nb_lignes() &&
129           "La transposition n'est pas correcte.");
129     assert(resultat.nb_colonnes() == transposee.nb_colonnes() &&
130           "La transposition n'est pas correcte.");
130     for (int i { 0 }; i < transposee.nb_lignes(); ++i)
131     {
132         for (int j { 0 }; j < transposee.nb_colonnes(); ++j)
133         {
134             assert(resultat(i, j) == transposee(i, j) &&
135                   "La transposition n'est pas correcte.");
136         }
137     }
138
139 void test_affichage()
140 {
141     Matrice m1 { 2, 2 };
142     m1(0, 0) = 1;
143     m1(0, 1) = 2;
144     m1(1, 0) = 3;
145     m1(1, 1) = 3;
146
147     Matrice m2 { 2, 2 };
148     m2(0, 0) = 0;
149     m2(0, 1) = 4;
150     m2(1, 0) = 4;
151     m2(1, 1) = -7;
152
153     std::cout << m1 << "\n";
154     std::cout << m1 + m2 << "\n";
155     std::cout << m1 + m2 + m2 << "\n";
156 }
157
158 int main()
159 {
160     test_addition();
161     test_multiplication_entier();
162     test_multiplication_matrice();
163     test_transposition();
164     test_affichage();
165
166     return 0;
167 }

```

Vous avez maintenant tout ce qu'il vous faut pour faire ce T.P. Bon courage. 🍊

V.6.3. Correction détaillée

Comment le TP s'est-il passé? Si vous avez tout fini, ou même pris de l'avance, félicitations. Si vous avez bloqué ou n'avez pas pu tout faire, pas d'inquiétude. La correction est justement là pour vous aider à voir une façon de faire. Elle se trouve ci-dessous au complet.

👁 Contenu masqué n°80

👁 Contenu masqué n°81

V.6.3.1. Les services

Contenu masqué

Contenu masqué n°78 : Opérateur à utiliser

Il s'agit de `()`. On peut surcharger cet opérateur en fournissant autant d'arguments que l'on veut. Dans notre cas, il s'agit de le surcharger en lui donnant un indice de ligne et un indice de colonne. Il nous permettra d'écrire `matrice(0, 0)`, par exemple. [Retourner au texte.](#)

Contenu masqué n°79 : Astuce secrète

Pour comprendre l'astuce, dites vous que nous avons une matrice de taille 2×3 , soit une matrice de deux lignes par trois colonnes. Cela fait un tableau de 6 éléments. Un moyen de stocker les éléments est de le faire **par colonne**, donc d'avoir l'élément $M_{0,0}$ à l'indice `0`, l'élément $M_{1,0}$ à l'indice `1`, l'élément $M_{0,1}$ à l'indice `2`, etc.

Supposons que je veuille l'élément $M_{1,1}$. Avec ce système, il faut que j'accède à l'élément d'indice `3`. Or, si on regarde, ce chiffre s'obtient en multipliant le nombre de lignes, qui vaut `2` par l'indice `y` voulu, soit `1`, puis en rajoutant l'indice `x` voulu, soit `1` encore.

L'astuce se répète pour n'importe quel indice $(x; y)$. On obtient donc la formule suivante:
 $y \times \text{lignes} + x$ [Retourner au texte.](#)

Contenu masqué n°80

```

1  #ifndef MATRICE_HPP
2  #define MATRICE_HPP
3
4  #include <iosfwd>
5  #include <vector>
6
7  /**
8   * @brief Manipulation de matrices.
9   */
10 class Matrice
11 {
12 public:
13     /**
14      * @brief Créé une nouvelle instance de Matrice.
15      * @param[in] lignes Le nombre de lignes de la matrice à créer.
16      * @param[in] colonne Le nombre de colonnes de la matrice à
17      créer.
18      * @pre Le nombre de lignes et de colonnes doit être supérieur
19      à 0.
20      * @param[in] valeur_initiale La valeur à donner à chaque
21      élément de la matrice. Vaut 0 par défaut.
22      */
23     Matrice(std::size_t lignes, std::size_t colonne, int
24             valeur_initiale = 0);
25
26     /**
27      * @brief Constructeur de copie par défaut.
28      * @param[in] matrice La matrice à copier.
29      */
30     Matrice(Matrice const & matrice) = default;
31
32     /**
33      * @brief Opérateur d'affectation par copie par défaut.
34      * @param[in] matrice La matrice à copier.
35      * @return La matrice actuelle après copie.
36      */
37     Matrice& operator=(Matrice const & matrice) = default;
38
39     /**
40      * @brief Opérateur d'accès à un élément de la matrice, version
41      constante.
42      * @param[in] x L'indice de la ligne voulue.
43      * @param[in] y L'indice de la colonne voulue.
44      * @pre Les indices de ligne et de colonne doivent être
45      inférieurs aux dimensions de la matrice.
46      * @return Une référence constante sur l'indice trouvé.
47      */

```

```

42  int const & operator()(std::size_t x, std::size_t y) const;
43
44  /**
45   * @brief Opérateur d'accès à un élément de la matrice.
46   * @param[in] x L'indice de la ligne voulue.
47   * @param[in] y L'indice de la colonne voulue.
48   * @pre Les indices de ligne et de colonne doivent être
49   inférieurs aux dimensions de la matrice.
50   * @return Une référence sur l'indice trouvé.
51   */
52  int& operator()(std::size_t x, std::size_t y);
53
54  /**
55   * @brief Additionne une matrice à la matrice courante.
56   * @param matrice La matrice à additionner à la matrice
57   courante.
58   * @pre La matrice passée en paramètre doit être de même
59   dimension que la matrice courante.
60   * @return Matrice& L'objet modifié après addition.
61   */
62  Matrice& operator+=(Matrice const & matrice);
63
64  /**
65   * @brief Opérateur libre d'addition de deux matrices.
66   * @param[in] lhs La matrice de gauche.
67   * @param[in] rhs La matrice de droite.
68   * @pre Les deux matrices doivent être de même dimension.
69   * @return Le résultat de l'addition des deux matrices.
70   */
71  friend Matrice operator+(Matrice lhs, Matrice const & rhs);
72
73  /**
74   * @brief Multiplie la matrice courante par un entier.
75   * @param[in] multiplicateur Un entier, par lequel la matrice
76   est multipliée.
77   * @return L'objet modifié après multiplication.
78   */
79  Matrice& operator*=(int multiplicateur);
80
81  /**
82   * @brief Opérateur libre de multiplication d'une matrice par
83   un entier.
84   * @param[in] m La matrice à multiplier.
85   * @param[in] multiplicateur L'entier multiplicateur.
86   * @return Le résultat de la multiplication d'une matrice par
87   un entier.
88   */
89  friend Matrice operator*(Matrice m, int multiplicateur);
90
91  /**

```

```

86      * @brief Opérateur libre de multiplication d'une matrice par
      un entier.
87      * @param[in] multiplicateur L'entier multiplicateur.
88      * @param[in] m La matrice à multiplier.
89      * @return Le résultat de la multiplication d'une matrice par
      un entier.
90      */
91      friend Matrice operator*(int multiplicateur, Matrice m);
92
93      /**
94      * @brief Multiplie la matrice courante par une autre matrice.
95      * @param[in] matrice La matrice par laquelle la matrice
      courante sera multipliée.
96      * @return L'objet modifié après multiplication.
97      */
98      Matrice& operator*=(Matrice const & matrice);
99
100     /**
101     * @brief Opérateur libre de multiplications de matrices.
102     * @param[in] lhs La matrice de gauche.
103     * @param[in] rhs La matrice de droite.
104     * @return Le résultat de la multiplication d'une matrice par
      une autre.
105     */
106     friend Matrice operator*(Matrice const & lhs, Matrice const &
      rhs);
107
108     /**
109     * @brief Affiche une matrice sur un flux de sortie.
110     * @param[in,out] stream Le flux sur lequel sera affichée la
      matrice.
111     * @param[in] matrice La matrice à afficher.
112     * @return Le flux de sortie utilisé.
113     */
114     friend std::ostream& operator<<(std::ostream & stream, Matrice
      const & matrice);
115
116     /**
117     * @brief Retourne un vecteur ligne, selon l'indice demandé.
118     * @param[in] index_ligne L'index de la ligne désirée.
119     * @pre L'index de la ligne désirée doit être strictement
      inférieur au nombre de lignes de la matrice.
120     * @return Un vecteur ligne de type Matrice, correspondant à la
      ligne demandée.
121     */
122     Matrice ligne(std::size_t index_ligne) const;
123
124     /**
125     * @brief Retourne un vecteur colonne, selon l'indice demandé.
126     * @param[in] index_colonne L'index de la colonne désirée.

```



```

127     * @pre L'index de la colonne désirée doit être strictement
128     * @return Un vecteur colonne de type Matrice, correspondant à
129     * la colonne demandée.
130     */
131     Matrice colonne(std::size_t index_colonne) const;
132
133     /**
134     * @brief Transpose la matrice courante.
135     * @return La matrice courante une fois transposée.
136     */
137     Matrice transpose() const;
138
139     /**
140     * @brief Donne le nombre de lignes de la matrice actuelle.
141     * @return Le nombre de lignes de la matrice actuelle.
142     */
143     std::size_t nb_lignes() const noexcept;
144
145     /**
146     * @brief Donne le nombre de colonnes de la matrice actuelle.
147     * @return Le nombre de colonnes de la matrice actuelle.
148     */
149     std::size_t nb_colonnes() const noexcept;
150 private:
151     /**
152     * Permet de calculer la position dans le tableau de l'élément
153     * qui se trouve à la ligne et à la colonne demandées.
154     * @param[in] ligne La ligne demandée.
155     * @param[in] colonne La colonne demandée.
156     * @pre La ligne et la colonne doivent être inférieures aux
157     * dimensions de la matrice.
158     * @returns L'index dans le tableau auquel se trouve l'élément.
159     */
160     std::size_t offset(std::size_t ligne, std::size_t colonne)
161         const noexcept;
162     std::size_t m_nb_lignes;
163     std::size_t m_nb_colonnes;
164     std::vector<int> m_matrice;
165 };
166 #endif

```

Listing 1 – Matrice.hpp

[Retourner au texte.](#)

Contenu masqué n°81

```

1  #include <algorithm>
2  #include <cassert>
3  #include <ostream>
4  #include <numeric>
5  #include "Matrice.hpp"
6
7  Matrice::Matrice(std::size_t lignes, std::size_t colonnes, int
   valeur_initiale)
8      : m_nb_lignes(lignes), m_nb_colonnes(colonnes), m_matrice(lignes
   * colonnes, valeur_initiale)
9  {
10     assert(m_nb_lignes > 0 && m_nb_colonnes > 0 &&
   "On ne peut pas avoir une matrice de dimension 0.");
11 }
12
13 std::size_t Matrice::offset(std::size_t ligne, std::size_t colonne)
   const noexcept
14 {
15     assert(ligne < m_nb_lignes && "Ligne demandée invalide.");
16     assert(colonne < m_nb_colonnes &&
   "Colonne demandée invalide.");
17     return colonne * m_nb_lignes + ligne;
18 }
19
20 int const & Matrice::operator()(std::size_t x, std::size_t y) const
21 {
22     return m_matrice[offset(x, y)];
23 }
24
25 int& Matrice::operator()(std::size_t x, std::size_t y)
26 {
27     return m_matrice[offset(x, y)];
28 }
29
30 Matrice& Matrice::operator+=(Matrice const & matrice)
31 {
32     assert(nb_lignes() == matrice.nb_lignes() && nb_colonnes() ==
   matrice.nb_colonnes() &&
   "Impossible d'additionner deux matrices de dimensions différentes.");
33     for (std::size_t i { 0 }; i < nb_lignes(); ++i)
34     {
35         for (std::size_t j { 0 }; j < nb_colonnes(); ++j)
36         {
37             (*this)(i, j) += matrice(i, j);
38         }
39     }
40     return *this;

```

```

41 }
42
43 Matrice operator+(Matrice lhs, Matrice const & rhs)
44 {
45     lhs += rhs;
46     return lhs;
47 }
48
49 Matrice& Matrice::operator*=(int multiplicateur)
50 {
51     for (std::size_t i { 0 }; i < nb_lignes(); ++i)
52     {
53         for (std::size_t j { 0 }; j < nb_colonnes(); ++j)
54         {
55             (*this)(i, j) *= multiplicateur;
56         }
57     }
58     return *this;
59 }
60
61 Matrice operator*(Matrice matrice, int multiplicateur)
62 {
63     matrice *= multiplicateur;
64     return matrice;
65 }
66
67 Matrice operator*(int multiplicateur, Matrice matrice)
68 {
69     return matrice * multiplicateur;
70 }
71
72 Matrice& Matrice::operator*=(Matrice const & matrice)
73 {
74     Matrice copie { *this * matrice };
75     std::swap(*this, copie);
76     return *this;
77 }
78
79 Matrice operator*(Matrice const & lhs, Matrice const & rhs)
80 {
81     assert(lhs.nb_colonnes() == rhs.nb_lignes() &&
82           "Le nombre de colonnes de la matrice multipliée doit être égal au nombre de lignes de la matrice multipliée");
83     Matrice copie { lhs.nb_lignes(), rhs.nb_colonnes() };
84     for (std::size_t i { 0 }; i < copie.nb_lignes(); ++i)
85     {
86         auto const ligne_courante { lhs.ligne(i).m_matrice };
87         for (std::size_t j { 0 }; j < copie.nb_colonnes(); ++j)
88         {

```

```

89         auto const colonne_courante { rhs.colonne(j).m_matrice
90             };
91         const int valeur {
92             std::inner_product(std::begin(ligne_courante),
93                 std::end(ligne_courante),
94                 std::begin(colonne_courante), 0) };
95         copie(i, j) = valeur;
96     }
97 }
98
99 std::ostream& operator<<(std::ostream & stream, Matrice const &
100     matrice)
101 {
102     for (std::size_t i { 0 }; i < matrice.nb_lignes(); ++i)
103     {
104         for (std::size_t j { 0 }; j < matrice.nb_colonnes(); ++j)
105         {
106             stream << matrice(i, j) << " ";
107         }
108         stream << "\n";
109     }
110     return stream;
111 }
112
113 Matrice Matrice::ligne(std::size_t index_ligne) const
114 {
115     assert(index_ligne < nb_lignes() &&
116         "L'index demandé pour la ligne est plus grand que la dimension de la m
117
118     Matrice matrice_ligne { 1, nb_colonnes() };
119     for (std::size_t j { 0 }; j < nb_colonnes(); ++j)
120     {
121         matrice_ligne(0, j) = (*this)(index_ligne, j);
122     }
123     return matrice_ligne;
124 }
125
126 Matrice Matrice::colonne(std::size_t index_colonne) const
127 {
128     assert(index_colonne < nb_colonnes() &&
129         "L'index demandé pour la colonne est plus grand que la dimension de la
130
131     Matrice matrice_colonne { nb_lignes(), 1 };
132     for (std::size_t i { 0 }; i < nb_lignes(); ++i)
133     {
134         matrice_colonne(i, 0) = (*this)(i, index_colonne);
135     }

```

```
132     return matrice_colonne;
133 }
134
135 Matrice Matrice::transpose() const
136 {
137     Matrice transposee { nb_colonnes(), nb_lignes() };
138     for (std::size_t i { 0 }; i < nb_lignes(); ++i)
139     {
140         for (std::size_t j { 0 }; j < nb_colonnes(); ++j)
141         {
142             transposee(j, i) = (*this)(i, j);
143         }
144     }
145
146     return transposee;
147 }
148
149 std::size_t Matrice::nb_lignes() const noexcept
150 {
151     return m_nb_lignes;
152 }
153
154 std::size_t Matrice::nb_colonnes() const noexcept
155 {
156     return m_nb_colonnes;
157 }
```

Listing 2 – Matrice.cpp

[Retourner au texte.](#)

V.7. Constructeurs

Commençons par réfléchir à la façon dont nous voulons créer des matrices. Il paraît évident de **renseigner le nombre de lignes et de colonnes** quand on crée une matrice. On peut vouloir aussi initialiser toute la matrice à une valeur précise.

De plus, comme nous sommes en plein dans la sémantique de valeur, on aimerait pouvoir **copier une matrice**, ce qui signifie que nous avons besoin d'un constructeur par copie et un opérateur d'affectation par recopie. On en déduit donc le code suivant.

```
1 Matrice(std::size_t lignes, std::size_t colonne, int
    valeur_initiale = 0);
2 // Constructeur par copie.
3 Matrice(Matrice const & matrice) = default;
4 // Opérateur d'affectation par recopie
5 Matrice& operator=(Matrice const & matrice) = default;
```

V.7.0.0.1. Accès aux éléments d'une matrice

De manière similaire à `std::vector` ou encore `std::string`, il faut qu'on puisse **accéder à un élément de la matrice**, tant en lecture qu'en écriture. On doit donc penser à gérer une matrice constante. Puisqu'on sait qu'on doit utiliser l'opérateur `()`, on obtient les services suivants.

```
1 // Version d'accès en lecture, marche avec une matrice const.
2 int const & operator()(std::size_t x, std::size_t y) const;
3 // Version d'accès en écriture.
4 int& operator()(std::size_t x, std::size_t y);
```

Remarquez l'usage d'assertions, car nous avons des préconditions à respecter sur les index fournis.

V.7.0.0.2. Surcharges des opérateurs d'addition et de multiplication

On a vu dans l'énoncé qu'on peut additionner deux matrices de même dimension, multiplier une matrice par un entier ou par une autre matrice. On en déduit donc plusieurs **surcharges d'opérateurs**, en n'oubliant bien sûr pas leurs versions libres.

```
1 // Additionne une matrice à l'objet courant.
2 Matrice& operator+=(Matrice const & matrice);
3 // Opérateur libre d'addition.
4 friend Matrice operator+(Matrice lhs, Matrice const & rhs);
```

```
5
6 // Multiplie la matrice courante par un entier.
7 Matrice& operator*=(int multiplicateur);
8 // Opérateur libre de multiplication par un entier.
9 friend Matrice operator*(Matrice matrice, int multiplicateur);
10
11 // Multiplie l'objet courant par une autre matrice.
12 Matrice& operator*=(Matrice const & matrice);
13 // Opérateur libre de multiplication de deux matrices.
14 friend Matrice operator*(Matrice lhs, Matrice const & rhs);
```

V.7.0.0.3. Autres fonctions utiles

Terminons la définition de notre classe `Matrice` en parlant de quelques services supplémentaires qu'elle peut être amenée à fournir. On a déjà la fonction permettant de faire une transposition, qui apparaît dans l'énoncé. J'ai personnellement choisi de faire **une fonction membre retournant une copie de l'objet courant**, après transposition, mais d'autres pourraient considérer que `transposition` s'applique à l'objet courant. C'est une question de choix.

```
1 Matrice transposition() const;
```

On peut aussi se dire qu'il serait bien d'accéder à une colonne ou une ligne précise de la matrice, les fameux **vecteurs lignes** et **vecteurs colonnes**. Cela ajoute deux nouvelles fonctions membres à la liste de nos services.

```
1 // Pour obtenir un vecteur ligne.
2 Matrice ligne(std::size_t index_ligne) const;
3 // Pour obtenir un vecteur colonne.
4 Matrice colonne(std::size_t index_colonne) const;
```

Enfin, il paraît important de connaître les dimensions de notre matrice, ce qui nous amène aux deux dernières fonctions membres à définir. Celles-ci sont très courtes et donc sont définies directement dans le fichier d'en-tête.

```
1 // Connaître le nombre de lignes de la matrice courante.
2 std::size_t nb_lignes() const noexcept { return _lignes; }
3 // Connaître le nombre de colonnes de la matrice courante.
4 std::size_t nb_colonnes() const noexcept { return _colonnes; }
```

Notez qu'on voit ici des détails d'implémentation, ce qui fait une excellente transition avec la partie suivante.

V.7.0.1. Les détails d'implémentation

V.8. Fichiers d'en-tête standards

Notre fichier d'en-tête `Matrice.hpp` utilise deux composants de la bibliothèque standard, à savoir `std::vector` et `std::ostream`. Il faut donc **inclure les fichiers d'en-tête correspondants**, pour que le nôtre soit valide. On s'assure ainsi qu'il est **auto-suffisant**, qu'il n'y a pas besoin d'ajouts externes ou de manipulations supplémentaires pour pouvoir l'utiliser.

Le fichier `<vector>` ne devrait pas vous surprendre. Par contre, l'autre `<iosfwd>` est nouveau. On l'utilise à la place de `<iostream>` quand on a **simplement besoin de déclarer des types I/O**, comme `std::ostream`, `std::istream`, `std::stringstream`, etc. Ce fichier d'en-tête est très léger car il ne contient que ça. À l'opposé, `<iostream>` contient toutes les implémentations et est donc **très lourd**.

V.8.0.0.1. Constructeurs

En plus des lignes et des colonnes, il faut un **tableau** dans lequel stocker les nombres de la matrice. Comme nous ne savons pas à l'avance sa taille, un `std::vector` est tout désigné pour ça. Et comme un de ses [constructeurs](#) permet d'initialiser tous les éléments à une certaine valeur, on peut déjà écrire le nôtre.

```
1 Matrice::Matrice(std::size_t lignes, std::size_t colonnes, int
   valeur_initiale)
2   : _lignes(lignes), _colonnes(colonnes), _matrice(lignes *
   colonnes, valeur_initiale)
3 {
4   assert(_lignes > 0 && _colonnes > 0 &&
   "On ne peut pas avoir une matrice de dimension 0.");
5 }
```

Notez qu'on utilise ici une assertion pour s'assurer qu'on fournit bien des dimensions correctes. Une matrice de dimension 0 ou moins n'ayant pas de sens, ce serait une erreur de programmation que de l'accepter.

Le constructeur par recopie et l'opérateur d'affectation par recopie seront très bien gérés par le compilateur, donc on se contente de les définir avec `default`.

V.8.0.0.2. Accès aux éléments d'une matrice

L'accès à un élément d'une matrice n'est pas très compliqué à écrire. Il faut simplement penser à une petite astuce pour savoir comment récupérer le bon élément dans le tableau. Si quelque chose vous perturbe, n'hésitez pas à relire les explications que j'ai données dans la partie énoncé.

Notez qu'encore une fois, nous utilisons la programmation par contrat pour nous assurer qu'on ne demande pas à accéder à un élément inexistant de la matrice. Et, afin de ne pas répéter dans deux endroits cette vérification, j'utilise une fonction membre privée qui s'occupe de ça, qui est ensuite appelée dans les deux opérateurs.


```

1  std::size_t Matrice::offset(std::size_t ligne, std::size_t colonne)
   const noexcept
2  {
3      assert(ligne < m_nb_lignes && "Ligne demandée invalide.");
4      assert(colonne < m_nb_colonnes &&
5              "Colonne demandée invalide.");
6      return colonne * m_nb_lignes + ligne;
7  }
8  int const & Matrice::operator()(std::size_t x, std::size_t y) const
9  {
10     return m_matrice[offset(x, y)];
11 }
12
13 int& Matrice::operator()(std::size_t x, std::size_t y)
14 {
15     return m_matrice[offset(x, y)];
16 }

```

V.8.0.0.3. Surcharges des opérateurs d'addition et de multiplication

Parlons maintenant d'un plus gros morceau. Commençons simplement et parlons de l'addition. On rappelle que celle-ci ne peut se faire que si les deux matrices ont la même dimension, d'où le `assert` pour s'en assurer. Dans une logique de réutilisation, on fait appel à `operator()` en l'appliquant sur l'objet courant (d'où le `*this`), ce qui nous permet de ne pas dupliquer la logique d'accès à un élément.

```

1  Matrice& Matrice::operator+=(Matrice const & matrice)
2  {
3      assert(nb_lignes() == matrice.nb_lignes() && nb_colonnes() ==
4              matrice.nb_colonnes() &&
5              "Impossible d'additionner deux matrices de dimensions différentes.");
6      for (std::size_t i { 0 }; i < nb_lignes(); ++i)
7      {
8          for (std::size_t j { 0 }; j < nb_colonnes(); ++j)
9          {
10             (*this)(i, j) += matrice(i, j);
11          }
12      }
13      return *this;
14 }

```

La version libre de l'addition est elle aussi très simple, puisqu'elle réutilise `operator+=`.

```

1 Matrice operator+(Matrice lhs, Matrice const & rhs)
2 {
3     return lhs += rhs;
4 }

```

Passons maintenant à la multiplication par un entier. Là encore, l'algorithme est simple puisqu'on doit simplement multiplier chaque élément de la matrice par l'entier en question. Il n'y a pas de précaution ni de vérification particulière à faire, donc pas de `assert` cette fois-ci.

```

1 Matrice& Matrice::operator*=(int multiplicateur)
2 {
3     for (std::size_t i { 0 }; i < nb_lignes(); ++i)
4     {
5         for (std::size_t j { 0 }; j < nb_colonnes(); ++j)
6         {
7             (*this)(i, j) *= multiplicateur;
8         }
9     }
10    return *this;
11 }

```

La version libre, pareil, très simple.

```

1 Matrice operator*(Matrice matrice, int multiplicateur)
2 {
3     matrice *= multiplicateur;
4     return matrice;
5 }

```

Maintenant, nous abordons le plus gros morceau, la **multiplication de deux matrices**. On doit commencer par vérifier que la précondition sur les dimensions des matrices multipliées est respectée. Ensuite, on applique l'algorithme, à savoir multiplier chaque ligne de la matrice actuelle par chaque colonne de la matrice passée en paramètre.

Sauf que voilà, si on modifie la matrice courante, cela va fausser les résultats puisque des éléments auront été modifiés en cours de route puis réutilisés dans les calculs suivants. On doit donc **utiliser une copie**. C'est justement ce que fait l'opérateur libre de multiplication. Cela veut dire que, de manière exceptionnelle, nous allons écrire l'opérateur intégré `*=` en fonction du libre `*`.

Sur cette copie, on va pouvoir faire la multiplication de chaque vecteur-ligne par chaque vecteur-colonne, qu'on peut obtenir avec les fonctions membres `lignes` et `colonnes`, que l'on définira juste après.

Cette multiplication revient à faire une somme de produits. Un algorithme comme celui-ci est assez commun. Alors commençons par regarder dans la bibliothèque standard. Ça tombe bien, la fonction `std::inner_product` [↗](#) sert justement à ça. C'est ainsi qu'on peut obtenir l'entier résultat de la multiplication d'un vecteur-ligne par un vecteur-colonne.

```

1  const int valeur { std::inner_product(std::begin(ligne_courante),
    std::end(ligne_courante), std::begin(colonne_courante), 0) };
2  copie(i, j) = valeur;

```

On obtient au final le code suivant pour l'opérateur libre.

```

1  Matrice operator*(Matrice const & lhs, Matrice const & rhs)
2  {
3      assert(lhs.nb_colonnes() == rhs.nb_lignes() &&
4          "Le nombre de colonnes de la matrice multipliée doit être égal au nombre de lignes de la matrice multipliée");
5      Matrice copie { lhs.nb_lignes(), rhs.nb_colonnes() };
6      for (std::size_t i { 0 }; i < copie.nb_lignes(); ++i)
7      {
8          auto const ligne_courante { lhs.ligne(i).m_matrice };
9          for (std::size_t j { 0 }; j < copie.nb_colonnes(); ++j)
10         {
11             auto const colonne_courante { rhs.colonne(j).m_matrice };
12             const int valeur {
13                 std::inner_product(std::begin(ligne_courante),
14                     std::end(ligne_courante),
15                     std::begin(colonne_courante), 0) };
16             copie(i, j) = valeur;
17         }
18     }
19     return copie;
20 }

```

Reste à écrire l'opérateur intégré. Comment mettre à jour notre objet avec le nouveau résultat? En l'échangeant. Encore une fois, la bibliothèque standard contient ce qu'il faut, avec `std::swap` [↗](#).

```

1  Matrice& Matrice::operator*=(Matrice const & matrice)
2  {
3      Matrice copie { *this * matrice };
4      // On échange l'objet courant avec la copie.
5      std::swap(*this, copie);
6      // L'objet courant est à jour et contient bien le résultat
7      // attendu, on peut donc le renvoyer.
8      return *this;
9  }

```

V.8.0.0.4. Autres fonctions utiles

Pour que la surcharge de l'opérateur `*=` puisse fonctionner, nous devons définir les fonctions membres `lignes` et `colonnes`. Celles-ci sont simples et ne consistent qu'en un simple tour de boucle.

```

1  Matrice Matrice::ligne(std::size_t index_ligne) const
2  {
3      assert(index_ligne < nb_lignes() &&
4              "L'index demandé pour la ligne est plus grand que la dimension de la m
5
6      Matrice matrice_ligne { 1, nb_colonnes() };
7      for (std::size_t j { 0 }; j < nb_colonnes(); ++j)
8      {
9          matrice_ligne(0, j) = (*this)(index_ligne, j);
10     }
11     return matrice_ligne;
12 }
13 Matrice Matrice::colonne(std::size_t index_colonne) const
14 {
15     assert(index_colonne < nb_colonnes() &&
16             "L'index demandé pour la colonne est plus grand que la dimension de la
17
18     Matrice matrice_colonne { nb_lignes(), 1 };
19     for (std::size_t i { 0 }; i < nb_lignes(); ++i)
20     {
21         matrice_colonne(i, 0) = (*this)(i, index_colonne);
22     }
23     return matrice_colonne;
24 }
```

Ensuite, la transposition ne devrait pas vous causer trop de problème, puisqu'il s'agit simplement d'inverser les indices de lignes et de colonnes.

```

1  Matrice Matrice::transposition() const
2  {
3      Matrice transposee { nb_colonnes(), nb_lignes() };
4      for (std::size_t i { 0 }; i < nb_lignes(); ++i)
5      {
6          for (std::size_t j { 0 }; j < nb_colonnes(); ++j)
7          {
8              transposee(j, i) = (*this)(i, j);
9          }
10     }
11
12     return transposee;
13 }
```

Enfin, il ne reste que l'affichage, qui nous permettra de visualiser les résultats. Dans une classe

plus complète, destinée à être utilisée par d'autres, nous ne l'écrivons pas, car le formatage et l'affichage des données dépend beaucoup du contexte et des besoins de l'utilisateur, donc **nous ne voudrions pas le forcer à afficher ses matrices d'une certaine façon**. Pour le T.P, on se le permet. 🍊

```
1 std::ostream& operator<<(std::ostream & stream, Matrice const &
   matrice)
2 {
3     for (std::size_t i { 0 }; i < matrice.nb_lignes(); ++i)
4     {
5         for (std::size_t j { 0 }; j < matrice.nb_colonnes(); ++j)
6         {
7             stream << matrice(i, j) << " ";
8         }
9         stream << "\n";
10    }
11    return stream;
12 }
```

V.8.1. Aller plus loin

Travailler sur les matrices vous a mis l'eau à la bouche? Vous voulez d'autres idées pour pratiquer la sémantique de valeur? Les exercices suivants vont vous plaire.

V.8.1.1. Le plus grand nombre

Comme vous le savez, les types comme `int` ou `double` ont une taille maximale, que vous pouvez retrouver dans l'en-tête `<limits>`. Et si jamais on veut manipuler des entiers encore plus grands? Il faut coder.

Le but de cet exercice est de pouvoir manipuler des entiers très, très grands. Pour cela, on peut utiliser des chaînes de caractères. Mais plutôt que de les manipuler nous-mêmes, nous allons **encapsuler tout ça** dans une belle classe, qui nous fournira des services simples à utiliser, comme l'addition, la soustraction, ou encore la valeur absolue.

Vous pouvez vous servir du code suivant comme base, pour vous donner une idée de comment la classe est utilisée.

```
1 int main()
2 {
3     BigInt a { 420 };
4     BigInt b { "48879554321858363547485554545557454555445" };
5
6     // Doit afficher 48879554321858363547485554545557454555865.
7     std::cout << a + b << std::endl;
8
9     // Doit afficher 48879554321858363547485554545557454555025.
10    std::cout << a - b << std::endl;
```

```

11
12 // Doit afficher 20529412815180512689943932909134130913286900.
13 std::cout << a * b << std::endl;
14
15 // Doit afficher 116379891242519913208298939394184415608.
16 std::cout << a / b << std::endl;
17
18 return 0;
19 }

```

V.8.1.2. Des polynômes

Continuons dans notre voyage mathématique en parlant maintenant des polynômes, ces outils d'algèbre qu'on commence à découvrir au lycée et qui ont la forme suivante.

$$P(x) = \sum_{i=0}^n a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n, \quad n \geq 0$$

De même que pour des réels ou des entiers, il est possible de leur appliquer plusieurs opérations, dont les quatre élémentaires. Le but de l'exercice est donc de créer une classe encapsulant ces différents services.

i

Si les maths et vous, ça fait 4...

Si vous ne connaissez pas les polynômes ou n'êtes pas très à l'aise en mathématiques, ne vous en faites pas. Vous pouvez vous contenter d'implémenter l'addition et la soustraction, qui sont les opérations les plus simples.

Voici un exemple d'utilisation, pour vous inspirer.

```

1 int main()
2 {
3     Polynome p1 { "3x^2+x+7" };
4     Polynome p2 { "4x^3-3x^2+x+1" };
5
6     // Doit afficher 4x^3+2x+8.
7     std::cout << p1 + p2 << std::endl;
8     // Doit afficher -4x^3+6x^2+6.
9     std::cout << p1 - p2 << std::endl;
10    // Doit afficher 12x^5-5x^4+28x^3-17x^2+8x+7.
11    std::cout << p1 * p2 << std::endl;
12
13    return 0;
14 }

```

V.8.1.3. En résumé

- Nous avons pu pratiquer toutes les notions vues précédemment.
- Nous avons revu l'importance de chercher au préalable si une fonction ou un algorithme n'existe pas déjà dans la bibliothèque standard.
- Nous avons pu voir l'importance de d'abord réfléchir aux services que propose une classe et seulement après de penser aux détails d'implémentation.

V.9. La sémantique de déplacement

V.10. Classes à sémantique d'entités

V.11. Le besoin de handles

V.12. [T.P] Le Javaquarium

V.13. Oh, le bel héritage

V.14. Les piliers de la POO

V.15. Classes templates


V.16. [T.P] Des polynômes génériques

Sixième partie

Annexes

VI.1. Index

Vous avez besoin de retrouver une notion rapidement? Tout ce que nous avons vu dans ce cours est ici. Et s'il manque quelque chose, n'hésitez pas à nous en faire part pour que nous puissions mettre à jour l'index.

Ce cours s'achève ici, mais votre apprentissage du C++ est loin d'être terminé. Il reste encore beaucoup de choses à apprendre. Vous pouvez commencer par les [contenus C++ disponibles sur Zeste de Savoir](#) . Certains approfondissent des concepts que nous avons introduits dans ce tutoriel, d'autres vous présenteront des notions nouvelles. Enfin, pour continuer votre apprentissage, voici quelques derniers conseils.

- **Soyez curieux:** fouillez sur Internet pour découvrir de nouvelles méthodes, approfondissez celles que vous connaissez, renseignez-vous, testez de nouveaux outils, lisez des bons livres, etc.
- **Entraînez-vous:** c'est le meilleur moyen de progresser. Faites des projets qui vous tiennent à cœur, mettez en œuvre des algorithmes connus, réalisez des exercices, participez au code des autres, etc.
- **Lisez des codes produits par d'autres personnes:** découvrez comment elles procèdent, apprenez d'elles de nouvelles techniques ou façons de faire et progressez en suivant leurs conseils. Vous pouvez par exemple commencer en visitant les forums de ce site.
- Enfin, le plus important: **amusez-vous!** 🍊

Liste des abréviations

ASCII American Standard Code for Information Interchange. 157

ISO International Organization for Standardization. 21, 27

PGCD Plus Grand Commun Diviseur. 2, 87, 96, 304

UB Undefined behavior. 105