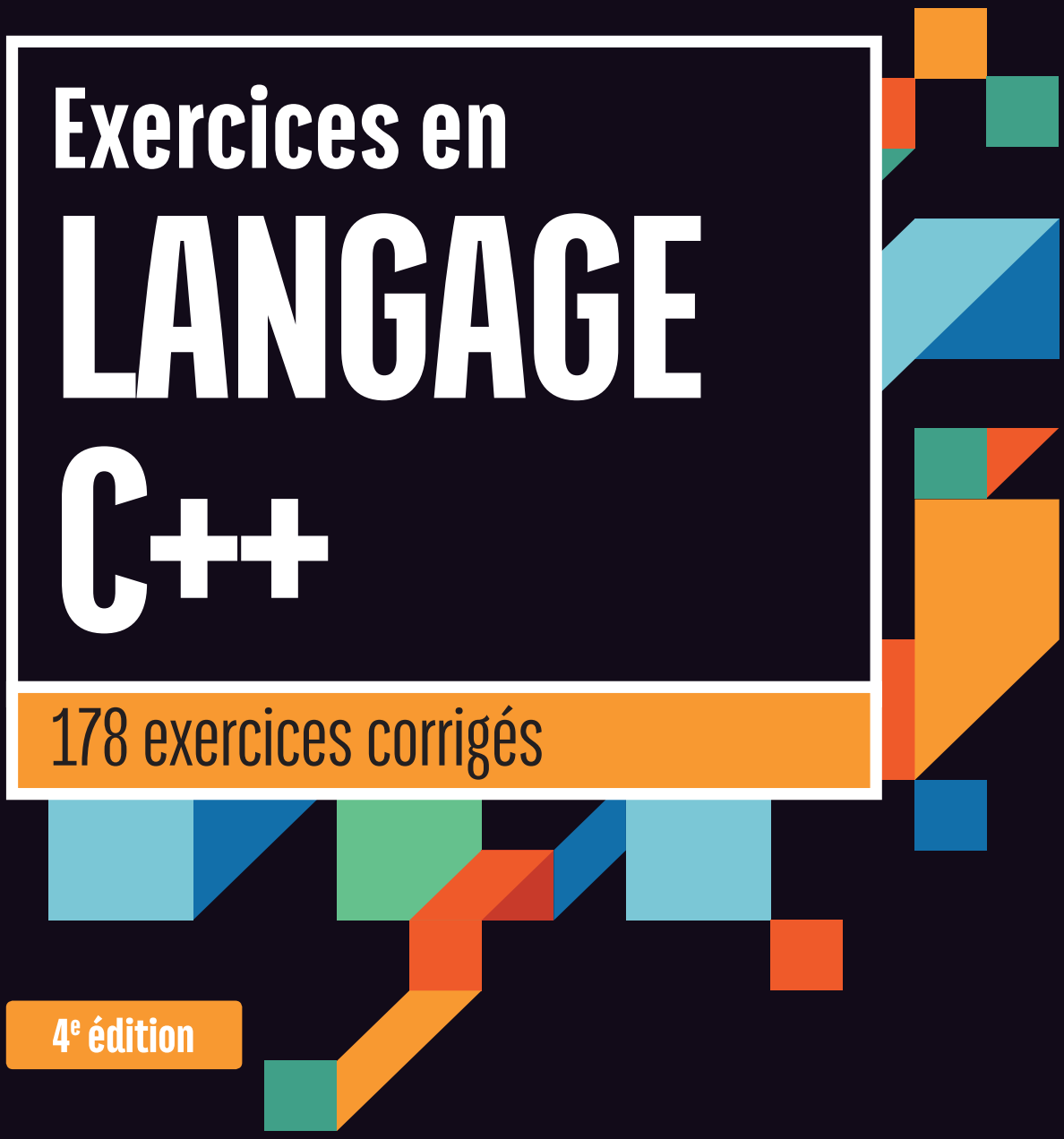
Claude Delannoy

Exercices en LANGAGE C++

178 exercices corrigés

4e édition



EYROLLES

178 exercices corrigés pour maîtriser le langage C++

Conçu pour les étudiants en informatique (DUT, licence, master, écoles d'ingénieur), ce recueil d'exercices corrigés et commentés est le complément idéal de *Programmer en langage C++* du même auteur ou de tout autre ouvrage d'initiation au langage C++. L'ouvrage propose 178 exercices pour mieux assimiler la syntaxe de base du langage (types et opérateurs, instructions de contrôle, fonctions, tableaux, pointeurs...) et les concepts objet du C++.

Les exercices vous permettront de vous forger une véritable méthodologie de conception de vos propres classes C++. Vous saurez notamment décider du bien-fondé de la surdéfinition de l'opérateur d'affectation ou du constructeur par recopie, tirer parti de l'héritage (simple ou multiple), créer vos propres bibliothèques de classes, exploiter les possibilités offertes par les patrons de fonctions et de classes, etc.

Cette 4e édition inclut 20 nouveaux exercices portant notamment sur les pointeurs intelligents et sur la nouvelle sémantique de déplacement introduits par les versions C++11 et C++14 de la norme.

Chaque chapitre débute par un rappel de cours suivi de plusieurs exercices de difficulté croissante. Les corrigés sont tous présentés suivant le même canevas : analyse détaillée du problème, solution sous forme de programme avec exemple de résultat d'exécution, justification des choix opérés – car il n'y a jamais de solution unique à un problème donné ! – et, si besoin, commentaires sur les points délicats et suggestions sur les extensions possibles du programme.

Le code source des corrigés est fourni sur le site *www.editions-eyrolles.com.*

Au sommaire

Ingénieur informaticien au CNRS, **Claude Delannoy** possède une grande pratique de la formation continue et de l'enseignement supérieur. Réputés pour la qualité de leur démarche pédagogique, ses ouvrages sur les langages et la programmation totalisent plus de 500 000 exemplaires vendus.

www.editions-eyrolles.com

Exercices en langage C++

AUX ÉDITIONS EYROLLES

Du même auteur

C. Delannoy. – **Exercices en Java**.
N°67385, 4e édition, 2014, 360 pages (réédition avec nouvelle présentation, 2016).

C. Delannoy. – **Programmer en langage C++** *(couvre C++11, C++14 et C++17).*
N°67386, 9e édition, 2017, 886 pages.

C. Delannoy. – **Le guide complet du langage C.**
N°14012, 2014, 844 pages.

C. Delannoy. – **S'initier à la programmation et à l'orienté objet.**
Avec des exemples en C, C++, C#, Python, Java et PHP.
N°11826, 2e édition, 2014, 360 pages.

Autres ouvrages

H. Bersini, I. Wellesz. – **La programmation orientée objet**.
Cours et exercices en UML 2 avec Java, C#, C++, Python, PHP et LinQ.
N°67399, 7e édition, 2017, 696 pages.

P. Roques. – **UML 2 par la pratique.**
N°67565, 8e édition, février 2018, 400 pages environ.

G. Swinnen. – **Apprendre à programmer avec Python 3.**
N°13434, 3e édition, 2012, 435 pages.

J. Engels. – **PHP 7.** *Cours et exercices*.
N°67360, 2017, 608 pages.

P. Martin, J. Pauli, C. Pierre de Geyer et E. Daspet. – **PHP 7 avancé.**
N°14357, 2016, 728 pages.

R. Rimelé. – **HTML 5.** *Une référence pour le développeur web.*
N°14365, 3e édition, 2017, 820 pages.

H. Giraudel, R. Goetter. – **CSS 3 : pratique du design web.**
N°14023, 2015, 372 pages.

R. Goetter. – **CSS 3 Flexbox.**
N°14363, 2016, 152 pages.

E. Biernat, M. Lutz. – **Data science : fondamentaux et études de cas.** *Machine Learning avec Python et R.*
N°14243, 2e édition, mars 2018, 400 pages environ.

Claude Delannoy

Exercices en langage C++

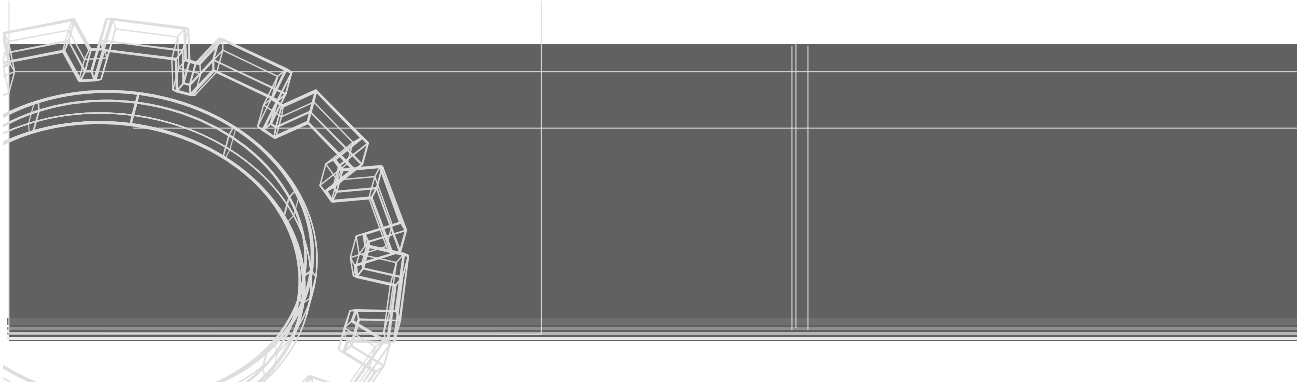
4e édition

EYROLLES

© Groupe EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

ISBN : 978-2-212-67663-1

Avant-propos



La maîtrise d'un langage de programmation passe obligatoirement par la pratique, c'est-à-dire la recherche personnelle d'une solution à un problème donné, et cette affirmation reste vraie pour le programmeur chevronné qui étudie un nouveau langage. C'est dans cette situation que se trouve généralement une personne qui aborde le C++ :

- soit elle connaît déjà un langage procédural classique autre que le C (Java, C#, PHP),
- soit elle connaît déjà le langage C sur lequel s'appuie effectivement C++ ; toutefois, ce dernier langage introduit suffisamment de possibilités supplémentaires et surtout de nouveaux concepts (en particulier ceux de la Programmation Orientée Objet) pour que son apprentissage s'apparente à celui d'un nouveau langage.

Cet ouvrage vous propose d'accompagner votre étude du C++ et de la prolonger à l'aide d'exercices appropriés, variés et de difficulté croissante, et ceci quelles que soient vos connaissances préalables. Il comporte :

- 5 chapitres destinés à ceux d'entre vous qui ne connaissent pas le C : types de base, opérateurs et expressions ; instructions de contrôle ; fonctions ; tableaux, pointeurs et chaînes de style C ;
- 1 chapitre pour assurer la transition de C à C++, destiné à ceux qui connaissent déjà le langage C ;
- 15 chapitres destinés à tous : les notions de classe, constructeur et destructeur ; les propriétés des fonctions membre ; la construction, la destruction et l'initialisation des objets ; les fonctions amies ; la surdéfinition d'opérateurs ; les conversions de type définies par l'utilisateur ; la technique de l'héritage ; les fonctions virtuelles ; les flots d'entrée et de sortie, les patrons de fonctions et les patrons de classes ; la gestion des exceptions.

- 1 chapitre, le 20, qui propose des exercices de synthèse.

Chaque chapitre débute par un rappel détaillé des connaissances nécessaires pour aborder les exercices correspondants (naturellement, un exercice d'un chapitre donné peut faire intervenir des points résumés dans les chapitres précédents).

Le cours complet correspondant à ces résumés se trouve dans l'un de nos ouvrages consacrés au C++, du même auteur, aux éditions Eyrolles.

Au sein de chaque chapitre, les exercices proposés vont d'une application immédiate du cours à des réalisations de classes relativement complètes. Au fil de votre progression dans l'ouvrage, vous réaliserez des classes de plus en plus réalistes et opérationnelles, et ayant un intérêt général.

Citons, par exemple :

- les ensembles ;
- les vecteurs dynamiques ;
- les tableaux dynamiques à plusieurs dimensions ;
- les listes chaînées ;
- les tableaux de bits ;
- les (vraies) chaînes de caractères ;
- les piles ;
- les complexes.

Naturellement, tous les exercices sont corrigés. Pour la plupart, la solution proposée ne se limite pas à une simple liste de programme (laquelle ne représente finalement qu'une rédaction possible parmi d'autres). Vous y trouverez une analyse détaillée du problème et, si besoin, les justifications de certains choix. Des commentaires viennent, le cas échéant, éclairer les parties quelque peu délicates. Fréquemment, vous trouverez des suggestions de prolongement ou de généralisation du problème abordé.

Outre la maîtrise du langage C++ proprement dit, les exercices proposés vous permettront de vous forger une méthodologie de conception de vos propres classes. Notamment, vous saurez :

- décider du bien-fondé de la surdéfinition de l'opérateur d'affectation ou du constructeur par recopie ;
- exploiter, lorsque vous jugerez que cela est opportun, les possibilités de « conversions implicites » que le compilateur peut mettre en place ;
- tirer parti de l'héritage (simple ou multiple) et déterminer quels avantages présente la création d'une bibliothèque de classes, notamment par le biais du typage dynamique des objets qui découle de l'emploi des fonctions virtuelles ;

- mettre en œuvre les possibilités de fonctions génériques (patrons de fonctions) et de classes génériques (patrons de classes).

Cette nouvelle édition comporte deux chapitres supplémentaires consacrés aux fonctionnalités les plus importantes introduites par C++11 et C++14, à savoir :

- les pointeurs intelligents
- la sémantique dite « de déplacement »

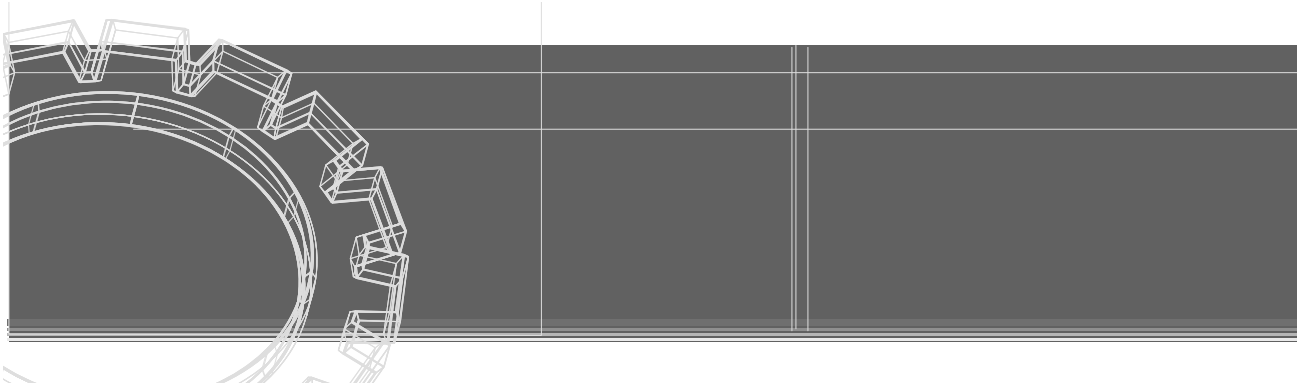
Remarque

Quelques exercices trouvent une solution évidente en faisant appel aux composants standard introduits par C++98. Nous continuons de les proposer, dans la mesure où la recherche d'une solution ne faisant pas appel aux composants standard conserve un intérêt didactique manifeste. De surcroît, le chapitre 21 montre comment résoudre les exercices lorsqu'on accepte, cette fois, de recourir à ces composants standard.

Table des matières

Chapitre 1

Généralités, types de base, opérateurs et expressions



Rappels

Généralités

Le canevas minimal à utiliser pour réaliser un programme en C++ se présente ainsi :

```
#include <iostream>
using namespace std ;
main()          // en-tête
{ .....         // corps du programme
}
```

Toute variable utilisée dans un programme doit avoir fait l'objet d'une déclaration en précisant le type et, éventuellement, la valeur initiale. Voici des exemples de déclarations :

```
int i ;   // i est une variable de type int nommée i
float x = 5.25 ;   // x est une variable de type float nommée x
                   // initialisée avec la valeur 5.25
const int NFOIS = 5 ;  // NFOIS est une variable de type int dont la
                       // valeur, fixée à 5, ne peut plus être modifiée
```

L'affichage d'informations à l'écran est réalisé en envoyant des valeurs sur le « flot *cout* », comme dans :

```
cout << n << 2*p ;  // affiche les valeurs de n et de 2*p sur l'écran
```

La lecture d'informations au clavier est réalisée en extrayant des valeurs du « flot *cin* », comme dans :

```
cin >> x >> y ;      // lit deux valeurs au clavier et les affecte à x et à y
```

Types de base

Les types de base sont ceux à partir desquels seront construits tous les autres, dits dérivés (il s'agira des types structurés comme les tableaux, les structures, les unions et les classes, ou d'autres types simples comme les pointeurs ou les énumérations).

Il existe trois types entiers : `short int` (ou `short`), `int` et `long int` (ou `long`). Les limitations correspondantes dépendent de l'implémentation. On peut également définir des types entiers non signés : `unsigned short int` (ou `unsigned short`), `unsigned int` et `unsigned long int` (ou `unsigned long`). Ces derniers sont essentiellement destinés à la manipulation de motifs binaires.

Les constantes entières peuvent être écrites en notation hexadécimale (comme 0xF54B) ou octale (comme 014). On peut ajouter le « suffixe » `u` pour un entier non signé et le suffixe `l` pour un entier de type `long`.

Il existe trois types flottants : `float`, `double` et `long double`. La précision et le « domaine représentable » dépendent de l'implémentation.

Le type « caractère » permet de manipuler des caractères codés sur un octet. Le code utilisé dépend de l'implémentation. Il existe trois types caractère : `signed char`, `unsigned char` et `char` (la norme ne précise pas s'il correspond à `signed char` ou `unsigned char`).

Les constantes de type caractère, lorsqu'elles correspondent à des « caractères imprimables », se notent en plaçant le caractère correspondant entre apostrophes.

Certains caractères disposent d'une représentation conventionnelle utilisant le caractère « \ », et notamment `'\n'` qui désigne un saut de ligne. De même, `'\''` représente le caractère `'` et `'\"'` désigne le caractère ". On peut également utiliser la notation hexadécimale (comme dans `'\x41'`) ou octale (comme dans `'\07'`).

Le type `bool` permet de manipuler des « booléens ». Il dispose de deux constantes notées `true` et `false`.

Les opérateurs de C++

Voici un tableau présentant l'ensemble des opérateurs de C++ (certains ne seront exploités que dans les chapitres ultérieurs) :

Catégorie	Opérateurs	Associativité
Résolution de portée	:: (portée globale - unaire) :: (portée de classe - binaire)	<-- -->
Référence	() [] -> .	-->
Unaire	+ - ++ -- ! ~ * & sizeof cast dynamic_cast static_cast reinterpret_cast const_cast new new[] delete delete[]	<---
Sélection	->* .*	<--
Arithmétique	* / %	--->
Arithmétique	+ -	--->
Décalage	<< >>	--->
Relationnels	< <= > >=	--->
Relationnels	== !=	--->
Manipulation de bits	&	--->
Manipulation de bits	^	--->
Manipulation de bits	|	--->
Logique	&&	--->
Logique	||	--->
Conditionnel (ternaire)	? :	--->
Affectation	= += -= *= /= %= &= ^= |= <<= >>=	<---
Séquentiel	,	--->

Les opérateurs arithmétiques et les opérateurs relationnels

Les opérateurs arithmétiques binaires (+, -, * et /) et les opérateurs relationnels ne sont définis que pour des opérandes d'un même type parmi `int`, `long int` (et leurs variantes non signées), `float`, `double` et `long double`. Mais on peut constituer des expressions mixtes (opérandes

de types différents) ou contenant des opérandes d'autres types (`bool`, `char` et `short`), grâce à l'existence de deux sortes de conversions implicites :

- les conversions d'ajustement de type, selon l'une des hiérarchies :

```
int -> long -> float -> double -> long double
unsinged int -> unsigned long -> float -> double -> long double
```

- les promotions numériques, à savoir des conversions systématiques de `char` (avec ou sans attribut de signe), `bool` et `short` en `int`.

Les opérateurs logiques

Les opérateurs logiques `&&` (et), `||` (ou) et `!` (non) acceptent n'importe quel opérande **numérique** (entier ou flottant) ou pointeur, en considérant que tout opérande de valeur non nulle correspond à « faux » :

Opérande 1	Opérateur	Opérande 2	Résultat
0	&&	0	faux
0	&&	non nul	faux
non nul	&&	0	faux
non nul	&&	non nul	vrai
0	||	0	faux
0	||	non nul	vrai
non nul	||	0	vrai
non nul	||	non nul	vrai
	!	0	vrai
	!	non nul	faux

Les deux opérateurs `&&` et `||` sont « à court-circuit » : le second opérande n'est évalué que si la connaissance de sa valeur est indispensable.

Opérateurs d'affectation

L'opérande de gauche d'un opérateur d'affectation doit être une `lvalue`, c'est-à-dire la référence à quelque chose de modifiable.

Les opérateurs d'affectation (=, -=, += ...), appliqués à des valeurs de type numérique, provoquent la conversion de leur opérande de droite dans le type de leur opérande de gauche. Cette conversion « forcée » peut être « dégradante ».

Opérateurs d'incrémentation et de décrémentation

Les opérateurs unaires d'incrémentation (++) et de décrémentation (--) agissent sur la valeur de leur unique opérande (qui doit être une `lvalue`) et fournissent la valeur après modification lorsqu'ils sont placés à gauche (comme dans `++n`) ou avant modification lorsqu'ils sont placés à droite (comme dans `n--`).

Opérateur de cast

Il est possible de forcer la conversion d'une expression quelconque dans un type de son choix, grâce à l'opérateur dit de « cast ». Par exemple, si `n` et `p` sont des variables entières, l'expression :

```
(double) n / p          // ou : static_cast<double> (n/p)
```

aura comme valeur celle de l'expression entière `n/p` convertie en `double`.

Opérateur conditionnel

Cet opérateur ternaire fournit comme résultat la valeur de son deuxième opérande si la condition mentionnée en premier opérande est non nulle (vraie pour une expression booléenne), et la valeur de son troisième opérande dans le cas contraire. Par exemple, avec cette affectation :

```
max = a>b ? a : b ;
```

on obtiendra dans la variable `max` la valeur de `a` si la condition `a>b` est vraie, la valeur de `b` dans le cas contraire. Avec :

```
valeur = 2*n-1 ? a : b ;
```

on obtiendra dans la variable `valeur` la valeur de `a` si l'expression `2*n-1` est non nulle, la valeur de `b` dans le cas contraire.

Exercice 1

Énoncé

Éliminer les parenthèses superflues dans les expressions suivantes :

```
a = (x+5)            /* expression 1 */
a = (x=y) + 2        /* expression 2 */
a = (x==y)           /* expression 3 */
(a<b) && (c<d)       /* expression 4 */
(i++) * (n+p)        /* expression 5 */
```

Solution

```
a = x+5              /* expression 1 */
```

L'opérateur + est prioritaire sur l'opérateur d'affectation =.

```
a = (x=y) + 2        /* expression 2 */
```

Ici, l'opérateur + étant prioritaire sur =, les parenthèses sont indispensables.

```
a = x==y             /* expression 3 */
```

L'opérateur == est prioritaire sur =.

```
a<b && c<d           /* expression 4 */
```

L'opérateur && est prioritaire sur l'opérateur <.

```
i++ * (n+p)          /* expression 5 */
```

L'opérateur ++ est prioritaire sur * ; en revanche, * est prioritaire sur +, de sorte qu'on ne peut éliminer les dernières parenthèses.

Exercice 2

Énoncé

Soient les déclarations :

```
char c = '\x01' ;
short int p = 10 ;
```

Quels sont le type et la valeur de chacune des expressions suivantes :

```
p + 3            /* 1 */
c + 1            /* 2 */
p + c            /* 3 */
3 * p + 5 * c    /* 4 */
```

Solution

1. p est d'abord soumis à la conversion « systématique » `short -> int`, avant d'être ajouté à la valeur 3 (`int`). Le résultat 13 est de type `int`.
2. c est d'abord soumis à la conversion « systématique » `char -> int` (ce qui aboutit à la valeur 1), avant d'être ajouté à la valeur 1 (`int`). Le résultat 2 est de type `int`.
3. p est d'abord soumis à la conversion systématique `short -> int`, tandis que c est soumis à la conversion systématique `char -> int` ; les résultats sont alors additionnés pour aboutir à la valeur 11 de type `int`.
4. p et c sont d'abord soumis aux mêmes conversions systématiques que ci-dessus ; le résultat 35 est de type `int`.

Exercice 3

Énoncé

Soient les déclarations :

```
char c = '\x05' ;
int n = 5 ;
long p = 1000 ;
float x = 1.25 ;
double z = 5.5 ;
```

Quels sont le type et la valeur de chacune des expressions suivantes :

```
n + c + p              /* 1 */
2 * x + c              /* 2 */
(char) n + c           /* 3 */
(float) z + n / 2      /* 4 */
```

Solutions

1. `c` est tout d'abord converti en `int`, avant d'être ajouté à `n`. Le résultat (`10`), de type `int`, est alors converti en `long`, avant d'être ajouté à `p`. On obtient finalement la valeur `1010`, de type `long`.

2. On évalue d'abord la valeur de `2*x`, en convertissant `2` (`int`) en `float`, ce qui fournit la valeur `2.5` (de type `float`). Par ailleurs, `c` est converti en `int` (conversion systématique). On évalue ensuite la valeur de `2*x`, en convertissant `2` (`int`) en `float`, ce qui fournit la valeur `2.5` (de type `float`). Pour effectuer l'addition, on convertit alors la valeur entière `5` (`c`) en `float`, avant de l'ajouter au résultat précédent. On obtient finalement la valeur `7.75`, de type `float`.

3. `n` est tout d'abord converti en `char` (à cause de l'opérateur de « cast »), tandis que `c` est converti (conversion systématique) en `int`. Puis, pour procéder à l'addition, il est nécessaire de reconvertir la valeur de `(char) n` en `int`. Finalement, on obtient la valeur `10`, de type `int`.

4. `z` est d'abord converti en `float`, ce qui fournit la valeur `5.5` (approximative, car, en fait, on obtient une valeur un peu moins précise que ne le serait `5.5` exprimé en `double`). Par ailleurs, on procède à la division entière de `n` par `2`, ce qui fournit la valeur entière `2`. Cette dernière est ensuite convertie en `float`, avant d'être ajoutée à `5.5`, ce qui fournit le résultat `7.5`, de type `float`.

Exercice 4

Énoncé

Soient les déclarations suivantes :

```
int n = 5, p = 9 ;
int q ;
float x ;
```

Quelle est la valeur affectée aux différentes variables concernées par chacune des instructions suivantes ?

```
q = n < p ;                  /* 1 */
q = n == p ;                 /* 2 */
q = p % n + p > n ;          /* 3 */
x = p / n ;                  /* 4 */
x = (float) p / n ;          /* 5 */
x = (p + 0.5) / n ;          /* 6 */
x = (int) (p + 0.5) / n ;    /* 7 */
q = n * (p > n ? n : p) ;    /* 8 */
q = n * (p < n ? n : p) ;    /* 9 */
```

Solution

1. `1`

2. `0`

3. `5` (`p%n` vaut `4`, tandis que `p>n` vaut `1`).

4. `1` (`p/n` est d'abord évalué en `int`, ce qui fournit `1` ; puis le résultat est converti en `float`, avant d'être affecté à `x`).

5. `1.8` (`p` est converti en `float`, avant d'être divisé par le résultat de la conversion de `n` en `float`).

6. `1.9` (`p` est converti en `float`, avant d'être ajouté à `0.5` ; le résultat est divisé par le résultat de la conversion de `n` en `float`).

7. `1` (`p` est converti en `float`, avant d'être ajouté à `0.5` ; le résultat (`5.5`) est alors converti en `int` avant d'être divisé par `n`).

8. `25`

9. `45`

Exercice 5

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
main ()
{
   int i, j, n ;
   i = 0 ; n = i++ ;
   cout << "A : i = " << i << " n = " << n << "\n" ;

   i = 10 ; n = ++ i ;
   cout << "B : i = " << i << " n = " << n << "\n" ;
   i = 20 ; j = 5 ; n = i++ * ++ j ;
   cout << "C : i = " << i << " j = " << j << " n = " << n << "\n" ;
   i = 15 ; n = i += 3 ;
   cout << "D : i = " << i << " n = " << n << "\n" ;

   i = 3 ; j = 5 ; n = i *= --j ;
   cout << "E : i = " << i << " j = " << j << " n = " << n << "\n" ;
}
```

Solution

```
A : i = 1 n = 0
B : i = 11 n = 11
C : i = 21 j = 6 n = 120
D : i = 18 n = 18
E : i = 12 j = 4 n = 12
```

Exercice 6

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
main()
{
   int n=10, p=5, q=10, r ;

   r = n == (p = q) ;
   cout << "A : n = " << n << "  p = " << p << " q = " << q
                      << " r = " << r << "\n" ;

   n = p = q = 5 ;
   n += p += q ;
   cout << "B : n = " << n << " p = " << p << " q = " << q << "\n" ;

   q = n < p ? n++ : p++ ;
   cout << "C : n = " << n << " p = " << p << " q = " << q << "\n" ;

   q = n > p ? n++ : p++ ;
   cout << "D : n = " << n << " p = " << p << " q = " << q << "\n" ;
}
```

Solution

```
A : n = 10  p = 10  q = 10  r = 1
B : n = 15  p = 10  q = 5
C : n = 15  p = 11  q = 10
D : n = 16  p = 11  q = 15
```

Exercice 7

Énoncé

Quels résultats fournira ce programme :

```
#include <iostream>
using namespace std ;
main()
{  int n, p, q ;

   n = 5 ; p = 2 ;                                  /* cas 1 */
   q = n++ >p || p++ != 3 ;
   cout << "A : n = " << n << " p = " << p << " q = " << q << "\n" ;

   n = 5 ; p = 2 ;                                  /* cas 2 */
   q = n++<p || p++ != 3 ;
   cout << "B : n = " << n << " p = " << p << " q = " << q << "\n" ;

   n = 5 ; p = 2 ;                                  /* cas 3 */
   q = ++n == 3 && ++p == 3 ;
   cout << "C : n = " << n << " p = " << p << " q = " << q << "\n" ;

   n = 5 ; p = 2 ;                                  /* cas 4 */
   q = ++n == 6 && ++p == 3 ;
   cout << "D : n = " << n << " p = " << p << " q = " << q << "\n" ;
}
```

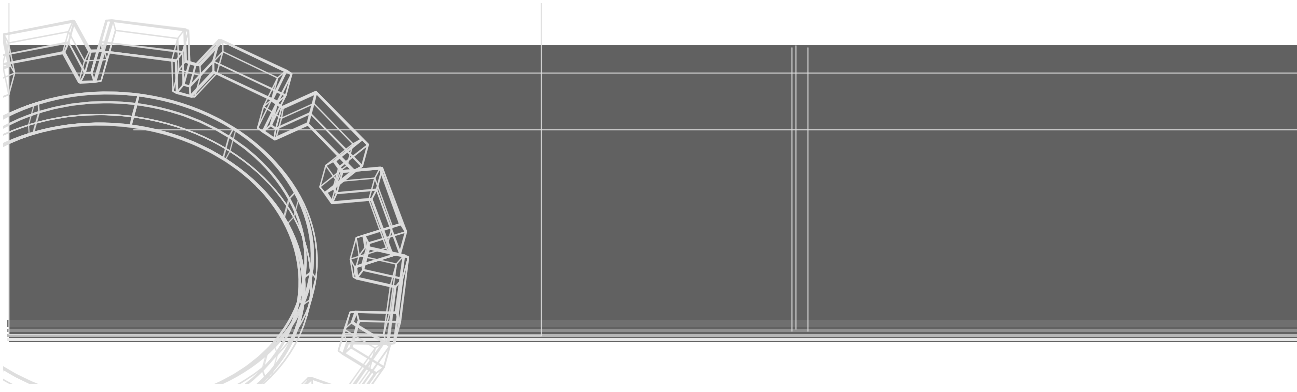
Solution

Il ne faut pas oublier que les opérateurs `&&` et `||` n'évaluent leur second opérande que lorsque cela est nécessaire. Ainsi, ici, il n'est pas évalué dans les cas 1 et 3. Voici les résultats fournis par ce programme :

```
A : n = 6  p = 2  q = 1
B : n = 6  p = 3  q = 1
C : n = 6  p = 2  q = 0
D : n = 6  p = 3  q = 1
```

Chapitre 2

Les instructions de contrôle



Rappels

Le terme *instruction* désignera indifféremment : une instruction simple (terminée par un point-virgule), une instruction structurée (choix, boucle) ou un bloc (instructions entre { et }).

Instruction if

Elle possède deux formes :

```
if (expression) instruction_1
           else instruction_2
if (expression) instruction_1
```

Lorsque des instructions `if` sont imbriquées, un `else` se rapporte toujours au dernier `if` auquel un `else` n'a pas encore été attribué.

Instruction switch

```
switch (expression) { bloc_d_instructions ]
```

Cette instruction évalue la valeur de l'expression entière mentionnée, puis recherche dans le bloc qui suit s'il existe une *étiquette* de la forme `case x` (`x` étant une expression constante, c'est-à-dire calculable par le compilateur) correspondant à cette valeur. Si c'est le cas, il y a branchement à l'instruction figurant à la suite de cette étiquette. Dans le cas contraire, on passe à l'instruction suivant le bloc. L'`expression` peut être de type `char`, auquel cas elle sera convertie en entier.

Une instruction `switch` peut contenir une ou plusieurs instructions `break` qui provoquent la sortie du bloc. Il est possible d'utiliser le mot `default` comme étiquette à laquelle le programme se branche lorsque aucune valeur satisfaisante n'a été rencontrée auparavant.

Instructions do... while et while

```
do instruction while (expression) ;

while (expression) instruction
```

L'expression gouvernant la boucle peut être d'un type quelconque ; elle sera convertie en *bool* selon la règle : non nul devient vrai, nul devient faux.

Instruction for

```
for ([expression_déclaration_1] ; [expression_2] ; [expression_3] )
       instruction
```

Les crochets signifient que leur contenu est facultatif.

- `expression_déclaration_1` est soit une expression, soit une déclaration d'une ou plusieurs variables d'un même type, initialisées ou non ;
- `expression_2` est une expression quelconque (qui sera éventuellement convertie en `bool`) ;
- `expression_3` est une expression quelconque.

Cette instruction est équivalente à :

```
{ expression_déclaration_1 ;
  while (expression_2)  { instruction ;
                          expression_3 ;
                        }
}
```

Note Depuis C++17, l'expression figurant dans une instruction `if` ou `switch` peut également être une « expression déclaration ».

Instructions break , continue et goto

Une boucle (`do... while`, `while` ou `for`) peut contenir une ou plusieurs instructions `break` dont le rôle est d'interrompre le déroulement de la boucle, en passant à l'instruction qui suit cette boucle. En cas de boucles imbriquées, `break` fait sortir de la boucle la plus interne. Si `break` apparaît dans un `switch` imbriqué dans une boucle, elle ne fait sortir que du `switch`.

L'instruction `continue` s'emploie uniquement dans une boucle. Elle permet de passer prématurément au tour de boucle suivant.

L'instruction `goto` permet le branchement en un emplacement quelconque du programme, repéré par une *étiquette*, comme dans cet exemple où, lorsqu'une certaine condition est vraie, on se branche à l'étiquette `erreur` :

```
for (...) { .....
            if (...) goto erreur ;
            .....
          }
   erreur : .....
```

Exercice 8

Énoncé

Quelles erreurs ont été commises dans chacun des groupes d'instructions suivants :

1.

```
if (a<b) cout << "ascendant"
    else cout << "non ascendant" ;
```

2.

```
int n ;
  ...
switch (2*n+1)
{ case 1 : cout << "petit" ;
  case n : cout << "moyen" ;
}
```

3.

```
const int LIMITE=100
int n ;
  ...
switch (n)
{ case LIMITE-1 : cout << "un peu moins" ;
  case LIMITE   : cout << "juste" ;
  case LIMITE+1 : cout << "un peu plus" ;
}
```

Solutions

1. Il manque un point-virgule à la fin de la première ligne :

```
if (a<b) cout << "ascendant" ;
    else cout << "non ascendant" ;
```

2. Les valeurs suivant le mot `case` doivent obligatoirement être des « expressions constantes », c'est-à-dire des expressions calculables par le compilateur lui-même. Ce n'est pas le cas de `n`.

3. Aucune erreur, les expressions telles que `LIMITE-1` étant bien des expressions constantes (ce qui n'était pas le cas en langage C).

Exercice 9

Énoncé

Soit le programme suivant :

```
#include <iostream>
main()
{   int n ;
    cin >> n ;
    switch (n)
    { case 0 : cout << "Nul\n" ;
      case 1 :
      case 2 : cout << "Petit\n" ;
               break ;
      case 3 :
      case 4 :
      case 5 : cout << "Moyen\n" ;
      default : cout << "Grand\n" ;
    }
}
```

Quels résultats affiche-t-il lorsqu'on lui fournit en donnée :

a. 0

b. 1

c. 4

d. 10

e. -5

Solutions

a.

```
Nul
Petit
```

b.

```
Petit
```

c.

```
Moyen
Grand
```

d.

```
Grand
```

e.

```
Grand
```

Exercice 10

Énoncé

Quelles erreurs ont été commises dans chacune des instructions suivantes :

a.

```
do cin >> c while (c != '\n') ;
```

b.

```
do while ( cin >> c, c != '\n') ;
```

c.

```
do {} while (1) ;
```

Solutions

a. Il manque un point-virgule :

```
do cin >> c ; while (c != '\n') ;
```

b. Il manque une instruction (éventuellement « vide ») après le mot `do`. On pourrait écrire, par exemple :

```
do {} while ( (cin >> c, c != '\n') ;
```

ou :

```
do ; while ( cin >> c, c != '\n') ;
```

c. Il n'y aura pas d'erreur de compilation (la valeur entière 1 est convertie en booléen, ce qui fournit la valeur vrai) ; toutefois, il s'agit d'une « boucle infinie ».

Exercice 11

Énoncé

Écrire plus lisiblement :

```
do {} while (cout << "donnez un nombre >0 ", cin >> n, n<=0) ;
```

Solution

Plusieurs possibilités existent, puisqu'il « suffit » de reporter, dans le corps de la boucle, des instructions figurant « artificiellement » sous forme d'expressions dans la condition de poursuite :

```
do
   cout << donnez un nombre >0 " ;
while (cin >> n, n<=0) ;
```

ou, mieux :

```
do
  { cout << "donnez un nombre >0 " ;
    cin >> n ;
  }
while (n<=0) ;
```

Exercice 12

Énoncé

Soit le petit programme suivant :

```
#include <iostream>
using namespace std ;
main()
{  int i, n, som ;
   som = 0 ;
   for (i=0 ; i<4 ; i++)
      { cout << "donnez un entier " ;
        cin >> n ;
        som += n ;
      }
   cout << "Somme : " << som ;
}
```

Écrire un programme réalisant exactement la même chose, en employant, à la place de l'instruction `for` :

a. une instruction `while` ;

b. une instruction `do ... while`.

Solutions

a.

```
#include <iostream>
using namespace std ;
main()
{  int i, n, som ;
   som = 0 ;
   i = 0 ;                    /* ne pas oublier cette "initialisation" */
   while (i<4)
      { cout << "donnez un entier " ;
        cin >> n ;
        som += n ;
        i++ ;                 /* ni cette "incrémentation" */
      }
   cout << "Somme : " << som ;
}
```

b.

```
#include <iostream>
using namespace std ;
main()
{  int i, n, som ;
   som = 0 ;
   i = 0 ;                  /* ne pas oublier cette "initialisation" */
   do
      { cout << "donnez un entier " ;
        cin >> n ;
        som += n ;
        i++ ;               /* ni cette "incrémentation" */
      }
   while (i<4) ;            /* attention, ici, toujours <4 */
   cout << "Somme : " << som ;
}
```

Exercice 13

Énoncé

Quels résultats fournit le programme suivant :

```
#include <iostream>
using namespace std ;
main()
{  int n=0 ;
   do
     { if (n%2==0) { cout << n << " est pair\n" ;
                     n += 3 ;
                     continue ;
                   }
       if (n%3==0) { cout << n << " est multiple de 3\n" ;
                     n += 5 ;
                   }
       if (n%5==0) { cout << n << " est multiple de 5\n" ;
                     break ;
                   }
       n += 1 ;
     }
   while (1) ;
}
```

Solution

```
0 est pair
3 est multiple de 3
9 est multiple de 3
```

```
15 est multiple de 3
20 est multiple de 5
```

Exercice 14

Énoncé

Quels résultats fournit le programme suivant :

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

    n=0 ;
    while (n<=5) n++ ;
    cout << "A : n = " << n << "\n" ;

    n=p=0 ;
    while (n<=8) n += p++ ;
    cout << "B : n = " << n << "\n" ;

    n=p=0 ;
    while (n<=8) n += ++p ;
    cout << "C : n = " << n << "\n" ;

    n=p=0 ;
    while (p<=5) n+= p++ ;
    cout << "D : n = " << n << "\n"  ;

    n=p=0 ;
    while (p<=5) n+= ++p ;
    cout << "E : n = " << n << "\n" ;
}
```

Solution

```
A : n = 6
B : n = 10
C : n = 10
D : n = 15
E : n = 21
```

Exercice 15

Énoncé

Quels résultats fournit le programme suivant :

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

    n=p=0 ;
    while (n<5) n+=2 ; p++ ;
    cout << "A : n = " << n << " p = " << p <<  "\n" ;

    n=p=0 ;
    while (n<5) { n+=2 ; p++ ; }
    cout << "B : n = " << n << " p = " << p << "\n" ;
}
```

Solution

```
A : n = 6, p = 1
B : n = 6, p = 3
```

Exercice 16

Énoncé

Quels résultats fournit le programme suivant :

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

    for (i=0, n=0 ; i<5 ; i++) n++ ;
    cout << "A : i = " << i << " n = " << n << "\n" ;

    for (i=0, n=0 ; i<5 ; i++, n++) {}
    cout << "B : i = " << i << " n = " << n << "\n" ;

    for (i=0, n=50 ; n>10 ; i++, n-= i ) {}
    cout << "C : i = " << i << " n = " << n << "\n" ;

    for (i=0, n=0 ; i<3 ; i++, n+=i,
       cout << "D : i = " << i << " n = " << n << "\n" ) ;
    cout << "E : i = " << i << " n = " << n << "\n" ;
}
```

Solution

```
A : i = 5, n = 5
B : i = 5, n = 5
C : i = 9, n = 5
D : i = 1, n = 1
D : i = 2, n = 3
D : i = 3, n = 6
E : i = 3, n = 6
```