

**Claude Delannoy**

**3<sup>e</sup> édition**

# **Exercices en langage C++**

**EYROLLES**

# **Exercices** **C++** en langage

*Du même auteur*

---

C. DELANNOY. – **Apprendre le C++**.  
N°12135, 1<sup>re</sup> édition 2007, 780 pages.

C. DELANNOY. – **C++ pour les programmeurs C**.  
N°12231, environ 580 pages, à paraître.

C. DELANNOY. – **Programmer en Java** (*Java 5 et 6*).  
N°12232, 5<sup>e</sup> édition, environ 780 pages + CD-Rom, à paraître.

C. DELANNOY. – **Exercices en Java** (*Java 5*).  
N°11989, 2<sup>e</sup> édition, 2006, 330 pages.

C. DELANNOY. – **Langage C**.  
N°11123, 1998, 944 pages (réédition au format semi-poche).

C. DELANNOY. – **Programmer en langage C. Avec exercices corrigés**.  
N°11072, 1996, 280 pages.

C. DELANNOY. – **Exercices en langage C**.  
N°11105, 1997, 260 pages.

*Autres ouvrages dans la même collection*

---

P. ROQUES. – **UML 2 par la pratique. Cours et exercices**.  
N°12014, 5<sup>e</sup> édition 2006, 360 pages.

X. BLANC, I. MOUNIER. – **UML 2 pour les développeurs. Cours et exercices corrigés**.  
N°12029, 2006, 218 pages.

H. BERSINI, I. WELLESZ. – **L'orienté objet**.  
*Cours et exercices en UML 2 avec PHP, Java, Python, C# et C++*.  
N°12084, 3<sup>e</sup> édition, 2007, 520 pages.

J. ENGELS. – **XHTML et CSS : cours et exercices**.  
N°11637, 2005, 350 pages.

J. ENGELS. – **PHP 5 : cours et exercices**.  
N°11407, 2005, 518 pages.

*Autres ouvrages*

---

I. HORTON. – **Visual C++ 6**.  
*Avec un CD-Rom contenant le produit Microsoft Visual C++ 6 Introductory Edition*.  
N°9043, 1999, 1 250 pages.

G. LEBLANC. – **C# et .NET 2.0**.  
N°11778, 2006, 700 pages.

E. DASPET et C. PIERRE de GEYER. – **PHP 5 avancé**.  
N°12167, 4<sup>e</sup> édition, environ 800 pages, à paraître en octobre 2007.

A. GONCALVES. – **Cahier du programmeur Java EE5**.  
N°12038, 2007, 330 pages.

C. PORTENEUVE. – **Bien développer pour le Web 2.0**.  
N°12028, 2006, 560 pages.

Claude Delannoy

# Exercices en langage C++

3<sup>e</sup> édition

EYROLLES

A horizontal line with a small circle in the center, positioned below the publisher's name.

ÉDITIONS EYROLLES  
61, bd Saint-Germain  
75240 Paris Cedex 05  
[www.editions-eyrolles.com](http://www.editions-eyrolles.com)



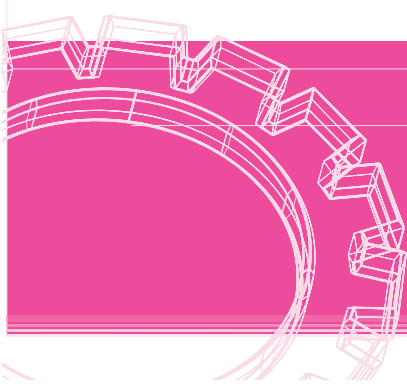
Le code de la propriété intellectuelle du 1<sup>er</sup> juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée notamment dans les établissements d'enseignement, provoquant une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.

© Groupe Eyrolles, 1997, 1999, 2007, ISBN : 978-2-212-12201-5

Dépôt légal : août 2007  
N° d'éditeur : 7676  
Imprimé en France

# 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, Visual Basic, Pascal, ...),
- soit elle connaît déjà la 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 :

- 4 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 ;
- un chapitre destiné à assurer la transition de C à C++ destinés à ceux qui connaissent déjà le langage C ;
- seize 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. Le chapitre 20 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'ouvrage *Apprendre le C++*, du même auteur.

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 d'un 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 copie ;

- 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).

---

Quelques exercices proposés dans les précédentes éditions de l'ouvrage trouvent maintenant une solution évidente en faisant appel aux composants standard introduits par la norme. Nous les avons cependant conservés, dans la mesure où la recherche d'une solution ne faisant pas appel aux composants standard conserve un intérêt didactique manifeste. De surcroît, nous avons introduit un nouveau chapitre (21), qui montre comment résoudre les exercices lorsqu'on accepte, cette fois, de recourir à ces composants standard.

---



# Table des matières

---

<b>1 Généralités, types de base, opérateurs et expressions.....</b>	<b>1</b>
Exercice 1 .....	5
Exercice 2 .....	6
Exercice 3 .....	7
Exercice 4 .....	8
Exercice 5 .....	9
Exercice 6 .....	10
Exercice 7 .....	11
<b>2 Les instructions de contrôle .....</b>	<b>13</b>
Exercice 8 .....	15
Exercice 9 .....	16
Exercice 10 .....	17
Exercice 11 .....	18
Exercice 12 .....	19
Exercice 13 .....	20
Exercice 14 .....	21
Exercice 15 .....	22
Exercice 16 .....	23
Exercice 17 .....	24
Exercice 18 .....	25
Exercice 19 .....	27
Exercice 20 .....	28
Exercice 21 .....	29
Exercice 22 .....	30
Exercice 23 .....	32
<b>3 Les fonctions .....</b>	<b>33</b>
Exercice 24 .....	37
Exercice 25 .....	38
Exercice 26 .....	39
Exercice 27 .....	40

Exercice 28 .....	41
Exercice 29 .....	42
Exercice 30 .....	43
Exercice 31 .....	44
Exercice 32 .....	45
Exercice 33 .....	46
<b>4 Les tableaux, les pointeurs et les chaînes de style C .....</b>	<b>49</b>
Exercice 34 .....	54
Exercice 35 .....	55
Exercice 36 .....	56
Exercice 37 .....	57
Exercice 38 .....	58
Exercice 39 .....	59
Exercice 40 .....	61
Exercice 41 .....	63
Exercice 42 .....	64
Exercice 43 .....	65
Exercice 44 .....	66
Exercice 45 .....	66
Exercice 46 .....	67
<b>5 Les structures .....</b>	<b>69</b>
Exercice 47 .....	71
Exercice 48 .....	73
Exercice 49 .....	75
Exercice 50 .....	76
Exercice 51 .....	78
Exercice 52 .....	79
<b>6 De C à C++ .....</b>	<b>81</b>
Exercice 53 .....	86
Exercice 54 .....	87
Exercice 55 .....	88
Exercice 56 .....	88
Exercice 57 .....	90
Exercice 58 .....	91

Exercice 59 .....	92
Exercice 60 .....	93
Exercice 61 .....	94
<b>7 Notions de classe, constructeur et destructeur .....</b>	<b>97</b>
Exercice 62 .....	100
Exercice 63 .....	102
Exercice 64 .....	104
Exercice 65 .....	106
Exercice 66 .....	108
Exercice 67 .....	109
Exercice 68 .....	112
<b>8 Propriétés des fonctions membre .....</b>	<b>115</b>
Exercice 69 .....	117
Exercice 70 .....	119
Exercice 71 .....	121
Exercice 72 .....	123
Exercice 73 .....	125
<b>9 Construction, destruction et initialisation des objets .....</b>	<b>127</b>
Exercice 74 .....	131
Exercice 75 .....	132
Exercice 76 .....	134
Exercice 77 .....	135
Exercice 78 .....	138
Exercice 79 .....	141
Exercice 80 .....	144
<b>10 Les fonctions amies .....</b>	<b>147</b>
Exercice 81 .....	149
Exercice 82 .....	151
Exercice 83 .....	152
<b>11 Surdéfinition d'opérateurs .....</b>	<b>155</b>
Exercice 84 .....	158
Exercice 85 .....	160
Exercice 86 .....	161

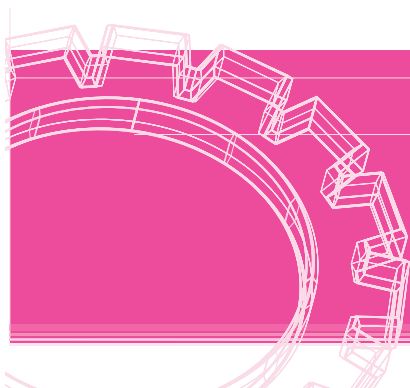
Exercice 87 .....	163
Exercice 88 .....	165
Exercice 89 .....	167
Exercice 90 .....	169
Exercice 91 .....	171
Exercice 92 .....	174
Exercice 93 .....	177
Exercice 94 .....	181
<b>12 Les conversions de type définies par l'utilisateur .....</b>	<b>183</b>
Exercice 95 .....	184
Exercice 96 .....	185
Exercice 97 .....	187
Exercice 98 .....	188
Exercice 99 .....	189
Exercice 100 .....	190
Exercice 101 .....	192
<b>13 La technique de l'héritage .....</b>	<b>195</b>
Exercice 102 .....	198
Exercice 103 .....	200
Exercice 104 .....	202
Exercice 105 .....	203
Exercice 106 .....	205
Exercice 107 .....	207
Exercice 108 .....	208
<b>14 L'héritage multiple .....</b>	<b>213</b>
Exercice 109 .....	215
Exercice 110 .....	216
Exercice 111 .....	217
Exercice 112 .....	217
Exercice 113 .....	219
Exercice 114 .....	220
<b>15 Les fonctions virtuelles .....</b>	<b>225</b>
Exercice 115 .....	227

Exercice 116 .....	228
Exercice 117 .....	230
<b>16 Les flots d'entrée et de sortie .....</b>	<b>237</b>
Exercice 118 .....	242
Exercice 119 .....	244
Exercice 120 .....	247
Exercice 121 .....	248
Exercice 122 .....	249
<b>17 Les patrons de fonctions.....</b>	<b>251</b>
Exercice 123 .....	254
Exercice 124 .....	255
Exercice 125 .....	256
Exercice 126 .....	257
<b>18 Les patrons de classes .....</b>	<b>259</b>
Exercice 127 .....	262
Exercice 128 .....	264
Exercice 129 .....	265
Exercice 130 .....	266
Exercice 131 .....	268
Exercice 132 .....	269
Exercice 133 .....	272
Exercice 134 .....	273
<b>19 Gestion des exceptions .....</b>	<b>275</b>
Exercice 135 .....	277
Exercice 136 .....	278
Exercice 137 .....	280
Exercice 138 .....	282
Exercice 139 .....	284
Exercice 140 .....	285
Exercice 141 .....	287
<b>20 Exercices de synthèse .....</b>	<b>289</b>
Exercice 142 .....	289
Exercice 143 .....	294

Exercice 144 .....	297
Exercice 145 .....	302
Exercice 146 .....	308
Exercice 147 .....	310
<b>21 Les composants standard .....</b>	<b>313</b>
Exercice 148 (67 revisité) .....	314
Exercice 149 (68 revisité) .....	315
Exercice 150 (77 revisité) .....	317
Exercice 151 (78 revisité) .....	318
Exercice 152 (79 revisité) .....	318
Exercice 153 (90 revisité) .....	319
Exercice 154 (91 revisité) .....	322
Exercice 155 (93 revisité) .....	325
Exercice 156 (142 revisité) .....	327
Exercice 157 (143 revisité) .....	331
Exercice 158 (94 revisité) .....	334

# Chapitre 1

## Généralités, types de base, opérateurs et expressions



### Rappels

---

#### Généralités

Le canevas minimal à utiliser pour réaliser un programme 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 « `\` » 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 des 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
unsigned 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 */
```

### Solution

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*c`, 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

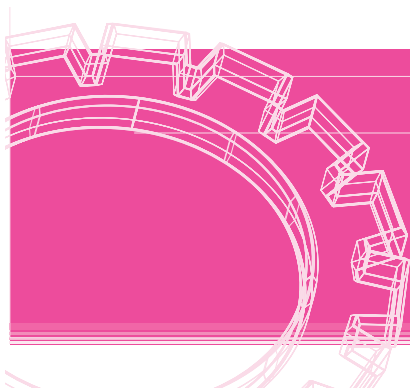
Il ne faut pas oublier que les opérateurs && et || n'évaluent leur second opérande que lorsque cela est nécessaire. Ainsi, ici, il n'est pas évalué dans les cas 1 et 3. Voici les résultats fournis par ce programme :

```
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 ;
                      }
}
```

## 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" ;
}
```

### Solution

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

### Solution

- 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) ;`

### Solution

- 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` :

- une instruction `while`,
- une instruction `do ... while`.

### Solution

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
```

## Exercice 17

### Énoncé

Écrire un programme qui calcule les racines carrées de nombres fournis en donnée. Il s'arrêtera lorsqu'on lui fournira la valeur 0. Il refusera les valeurs négatives. Son exécution se présentera ainsi :

```
donnez un nombre positif : 2
sa racine carrée est : 1.414214e+00
donnez un nombre positif : -1
svp positif
donnez un nombre positif : 5
sa racine carrée est : 2.236068e+00
donnez un nombre positif : 0
```

Rappelons que la fonction `sqrt` fournit la racine carrée (double) de la valeur (double) qu'on lui donne en argument.

### Solution

Il existe beaucoup de rédactions possibles ; en voici 3 :

1.

```
#include <iostream>
#include <cmath> // pour la déclaration de sqrt
using namespace std ;
main()
{ double x ;
  do
  { cout << "donnez un nombre positif : " ;
    cin >> x ;
    if (x < 0) cout << "svp positif \n" ;
    if (x <= 0) continue ;
    cout << "sa racine carrée est : " << sqrt (x) << "\n" ;
  }
  while (x) ;
}
```

2.

```
#include <iostream>
#include <cmath>
using namespace std ;
main()
{ double x ;
  do
  { cout << "donnez un nombre positif : " ;
    cin >> x ;
```

```

        if (x<0) { cout << "svp positif \n" ;
                  continue ;
        }
        if (x>0) cout << "sa racine carrée est : " << sqrt (x) << "\n" ;
    }
    while (x) ;
}

```

## 3.

```

#include <iostream>
#include <cmath>
using namespace std ;
main()
{ double x ;
  do
  { cout <<"donnez un nombre positif : " ;
    cin >> x ;

    if (x < 0) { cout << "svp positif \n" ;
                  continue ;
    }

    if (x>0) cout << "sa racine carrée est : " << sqrt (x) << "\n" ;
    if (x==0) break ;
  }
  while (1) ;
}

```

## Exercice 18

### Énoncé

Calculer la somme des  $n$  premiers termes de la « série harmonique », c'est-à-dire la somme :

$$1 + 1/2 + 1/3 + 1/4 + \dots + 1/n$$

La valeur de  $n$  sera lue en donnée.

### Solution

```

#include <iostream>
using namespace std ;
main()
{
    int nt ;           /* nombre de termes de la série harmonique */

```

```

float som ;          /* pour la somme de la série */
int i ;

do
{ cout << "combien de termes : " ;
  cin >> nt ;
}
while (nt<1) ;
for (i=1, som=0 ; i<=nt ; i++) som += (float)1/i ;
cout << "Somme des " << nt << " premiers termes = "<< som ;
}

```

## Remarques

### 1. Rappelons que dans :

```
som += (float)1/i
```

l'expression de droite est évaluée en convertissant d'abord 1 et i en float.

Il faut éviter d'écrire :

```
som += 1/i
```

auquel cas, les valeurs de  $1/i$  seraient toujours nulles (sauf pour  $i=1$ ) puisque l'opérateur /, lorsqu'il porte sur des entiers, correspond à la division entière.

De même, en écrivant :

```
som += (float) (1/i)
```

le résultat ne serait pas plus satisfaisant puisque la conversion en flottant n'aurait lieu qu'après la division (en entier).

En revanche, on pourrait écrire :

```
som += 1.0/i ;
```

2. Si l'on cherchait à exécuter ce programme pour des valeurs élevées de n (en prévoyant alors une variable de type float ou double), on constaterait que la valeur de la somme semble « converger » vers une limite (bien qu'en théorie la série harmonique « diverge »). Cela provient tout simplement de ce que, dès que la valeur de  $1/i$  est « petite » devant som, le résultat de l'addition de  $1/i$  et de som est **exactement** som. On pourrait toutefois améliorer le résultat en effectuant la somme « à l'envers » (en effet, dans ce cas, le rapport entre la valeur à ajouter et la somme courante serait plus faible que précédemment).

## Exercice 19

### Énoncé

Afficher un triangle isocèle formé d'étoiles. La hauteur du triangle (c'est-à-dire le nombre de lignes) sera fourni en donnée, comme dans l'exemple ci-dessous. On s'arrangera pour que la dernière ligne du triangle s'affiche sur le bord gauche de l'écran.

combien de lignes ? 10

```

      *
    ***
  *****
 *****
*****
*****
*****
*****
*****
*****
*****

```

### Solution

```

#include <iostream>
using namespace std ;

main()
{
    const char car = '*' ; /* caractère de remplissage */
    int nlines ;           /* nombre total de lignes */
    int nl ;               /* compteur de ligne */
    int nesp ;             /* nombre d'espaces précédant une étoile */
    int j ;

    cout << "combien de lignes ? " ;
    cin >> nlines ;
    for (nl=0 ; nl<nlines ; nl++)
    {
        nesp = nlines - nl - 1 ;
        for (j=0 ; j<nesp ; j++) cout << ' ' ;
        for (j=0 ; j<2*nl+1 ; j++) cout << car ;
        cout << '\n' ;
    }
}

```

## Exercice 20

### Énoncé

Afficher toutes les manières possibles d'obtenir un euro avec des pièces de 2 cents, 5 cents et 10 cents. Dire combien de possibilités ont été ainsi trouvées. Les résultats seront affichés comme suit :

```

1 euro = 50 X 2c
1 euro = 45 X 2c   2 X 5c
1 euro = 40 X 2c   4 X 5c
1 euro = 35 X 2c   6 X 5c
1 euro = 30 X 2c   8 X 5c
1 euro = 25 X 2c  10 X 5c
1 euro = 20 X 2c  12 X 5c
1 euro = 15 X 2c  14 X 5c
1 euro = 10 X 2c  16 X 5c
1 euro =  5 X 2c  18 X 5c
1 euro = 20 X 5c
1 euro = 45 X 2c   1 X 10c
1 euro = 40 X 2c   2 X 5c   1 X 10c
1 euro = 35 X 2c   4 X 5c   1 X 10c
1 euro = 10 X 2c   2 X 5c   7 X 10c
1 euro =  5 X 2c   4 X 5c   7 X 10c
1 euro =  6 X 5c   7 X 10c
1 euro = 10 X 2c   8 X 10c
1 euro =  5 X 2c   2 X 5c   8 X 10c
1 euro =  4 X 5c   8 X 10c
1 euro =  5 X 2c   9 X 10c
1 euro =  2 X 5c   9 X 10c
1 euro = 10 X 10c

```

En tout, il y a 66 façons de faire 1 euro

Rappelons que l'insertion dans le flot `cout` d'une expression de la forme `setw(n)`, où `n` est une expression entière, demande de réaliser l'affichage suivant (et uniquement ce dernier) sur `n` caractères au minimum. L'emploi de `setw` nécessite l'inclusion du fichier `iomanip`.

### Solution

```

#include <iostream>
#include <iomanip>    // pour setw
using namespace std ;
main()
{
    int nbf ;           // compteur du nombre de façons de faire 1 euro
    int n10 ;           // nombre de pièces de 10 centimes
    int n5 ;            // nombre de pièces de 5 centimes
    int n2 ;            // nombre de pièces de 2 centimes

```

```

    nbf = 0 ;
    for (n10=0 ; n10<=10 ; n10++)
        for (n5=0 ; n5<=20 ; n5++)
            for (n2=0 ; n2<=50 ; n2++)
                if ( 2*n2 + 5*n5 + 10*n10 == 100)
                    { nbf ++ ;
                      cout << "1 euro = " ;
                      if (n2) cout << setw(2) << n2 << " X 2c " ;
                      if (n5) cout << setw(2) << n5 << " X 5c " ;
                      if (n10) cout << setw(2) << n10 << " X 10c " ;
                      cout << "\n" ;
                    }

    cout << "\nEn tout, il y a " << nbf << " façons de faire 1 euro\n" ;
}

```

## Exercice 21

### Énoncé

Écrire un programme qui détermine la  $n^{\text{ième}}$  valeur  $u_n$  ( $n$  étant fourni en donnée) de la « suite de Fibonacci » définie comme suit :

```

u1 = 1
u2 = 1
 $u_n = u_{n-1} + u_{n-2}$     pour  $n > 2$ 

```

### Solution

```

#include <iostream>
using namespace std ;

main()
{
    int u1, u2, u3 ;           /* pour "parcourir" la suite */
    int n ;                   /* rang du terme demandé */
    int i ;                   /* compteur */

    do
    { cout << "rang du terme demandé (au moins 3) ? " ;
      cin >> n ;
    }
    while (n<3) ;

    u2 = u1 = 1 ;              /* les deux premiers termes */
    i = 2 ;

```

```

while (i++ <= n)          /* attention, l'algorithme ne fonctionne */
{ u3 = u1 + u2 ;          /* que pour n > 2 */
  u1 = u2 ;
  u2 = u3 ;
}
//      autre formulation possible :
// for (i=3 ; i<=n ; i++, u1=u2, u2=u3) u3 = u1 + u2 ;

cout << "Valeur du terme de rang " << n << " : " << u3 ;
}

```

Notez que, comme à l'accoutumée en C++, beaucoup de formulations sont possibles. Nous en avons d'ailleurs placé une seconde en commentaire de notre programme.

## Exercice 22

### Énoncé

Écrire un programme qui trouve la plus grande et la plus petite valeur d'une succession de notes (nombres entiers entre 0 et 20) fournies en données, ainsi que le nombre de fois où ce maximum et ce minimum ont été attribués. On supposera que les notes, en nombre non connu à l'avance, seront terminées par une valeur négative. On s'astreindra à ne pas utiliser de « tableau ». L'exécution du programme pourra se présenter ainsi :

```

donnez une note (-1 pour finir) : 12
donnez une note (-1 pour finir) : 8
donnez une note (-1 pour finir) : 13
donnez une note (-1 pour finir) : 7
donnez une note (-1 pour finir) : 11
donnez une note (-1 pour finir) : 12
donnez une note (-1 pour finir) : 7
donnez une note (-1 pour finir) : 9
donnez une note (-1 pour finir) : -1

```

```

note maximale : 13 attribuée 1 fois
note minimale : 7 attribuée 2 fois

```

### Solution

```

#include <iostream>
using namespace std ;

main()
{
    int note ;          // note "courante"
    int max ;           // note maxi
    int min ;           // note mini
    int nmax ;          // nombre de fois où la note maxi a été trouvée
    int nmin ;          // nombre de fois où la note mini a été trouvée

```

```

max = -1 ;    // initialisation max (possible car toutes notes >=0
min = 21 ;    // initialisation min (possible car toutes notes < 21
while (cout << "donnez une note (-1 pour finir) : ",
      cin >> note, note >=0)
{ if (note == max) nmax++ ;
  if (note > max) { max = note ;
                  nmax = 1 ;
                }
  if (note == min) nmin++ ;
  if (note < min) { min = note ;
                  nmin = 1 ;
                }
}

if (max >= 0)
{ cout << "\nnote maximale : " << max << " attribuée "
  << nmax << " fois\n" ;
  cout << "note minimale : " << min << " attribuée "
  << nmin << " fois\n" ;
}
else cout << "vous n'avez fourni aucune note" ;
}

```

## Exercice 23

### Énoncé

Écrire un programme qui affiche la « table de multiplication » des nombres de 1 à 10, sous la forme suivante :

	I	1	2	3	4	5	6	7	8	9	10
1	I	1	2	3	4	5	6	7	8	9	10
2	I	2	4	6	8	10	12	14	16	18	20
3	I	3	6	9	12	15	18	21	24	27	30
4	I	4	8	12	16	20	24	28	32	36	40
5	I	5	10	15	20	25	30	35	40	45	50
6	I	6	12	18	24	30	36	42	48	54	60
7	I	7	14	21	28	35	42	49	56	63	70
8	I	8	16	24	32	40	48	56	64	72	80
9	I	9	18	27	36	45	54	63	72	81	90
10	I	10	20	30	40	50	60	70	80	90	100

Rappelons que l'insertion dans le flot `cout` d'une expression de la forme `setw(n)`, où `n` est une expression entière, demande de réaliser l'affichage suivant sur `n` caractères au minimum. L'emploi de `setw` nécessite l'inclusion du fichier `iomanip`.

### Solution

```
#include <iostream>
#include <iomanip> // pour setw
using namespace std ;

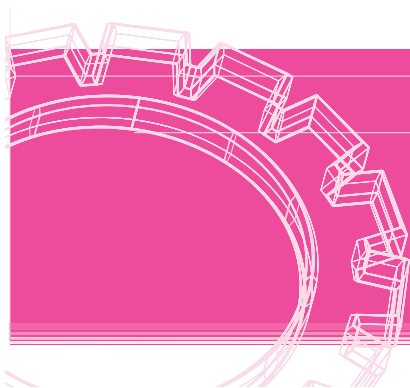
main()
{
    const int NMAX = 10 ;    // nombre de valeurs
    int i, j ;

    /* affichage ligne en-tête */
    cout << "          I" ;
    for (j=1 ; j<=NMAX ; j++) cout << setw(4) << j ;
    cout << "\n" ;
    printf ("-----") ;
    for (j=1 ; j<=NMAX ; j++) cout << "----" ;
    cout << "\n" ;

    /* affichage des différentes lignes */
    for (i=1 ; i<=NMAX ; i++)
    {
        cout << setw(4) << i << " I" ;
        for (j=1 ; j<=NMAX ; j++)
            cout << setw(4) << i*j ;
        cout << "\n" ;
    }
}
```

# Chapitre 3

## Les fonctions



### Rappels

---

#### Généralités

Une fonction est un bloc d'instructions éventuellement paramétré par un ou plusieurs arguments et pouvant fournir un résultat nommé souvent « valeur de retour ». On distingue la définition d'une fonction de son utilisation, cette dernière nécessitant une déclaration.

La définition d'une fonction se présente comme dans cet exemple :

```
float fexple (float x, int b, int c)    // en-tête de la fonction
{ // corps de la fonction
}
```

L'en-tête précise le nom de la fonction (*fexple*) ainsi que le type et le nom (muet) de ses différents arguments (*x*, *b* et *c*). Le corps est un bloc d'instructions qui définit le rôle de la fonction.

Au sein d'une autre fonction (*y* compris *main*), on utilise cette fonction de cette façon :

```
float fexple (float, int, int) ; // déclaration de fexple ("prototype")
.....
fexple (z, n, p) ; //appel fexple avec les arguments effectifs z, n et p
```