

Claude Delannoy

Le guide complet du langage C



● Éditions
EYROLLES

La référence des étudiants et des développeurs professionnels

Cet ouvrage de référence a été conçu pour les étudiants de niveau avancé en programmation et pour les développeurs souhaitant approfondir leur connaissance du C ou trouver une réponse précise aux problèmes techniques rencontrés lors du développement d'applications professionnelles.

Exhaustif et précis, l'ouvrage explore le langage C dans ses moindres recoins. Il clarifie les points délicats et les ambiguïtés du langage, analyse le comportement qu'on peut attendre d'un code ne respectant pas la norme ou confronté à une situation d'exception. Tout au long de l'ouvrage, des notes soulignent les principales différences syntaxiques entre le C et le C++ , de manière à établir des passerelles entre les deux langages.

Une annexe présente les spécificités des deux dernières moutures de la norme ISO du langage, connues sous les noms C99 et C11.

Au sommaire

Les bases du langage C. Historique, programmes et compilation, variables et objets. **Structure d'un programme source.** Jeux de caractères, identificateurs, mots-clés, séparateurs, format libre, commentaires, tokens. **Types de base.** Types entiers, types caractère, types flottants, fichiers *limits.h* et *float.h*. **Opérateurs et expressions.** Opérateurs arithmétiques, relationnels, logiques, de manipulation de bits, d'affectation et d'incréméntation, de cast ; conversions numériques ; opérateur conditionnel, séquentiel, *sizeof* ; priorité et associativité ; expressions constantes. **Instructions exécutables.** Expressions, blocs ; instructions *if*, *switch*, *do... while*, *while*, *for*, *break*, *continue* ; schémas de boucles utiles ; *goto* et les étiquettes. **Tableaux.** Déclaration, utilisation, débordement d'indice, tableau de tableaux, initialisation d'un tableau. **Pointeurs.** Variable de type pointeur et opérateur *, déclaration, propriétés arithmétiques ; opérateurs +, -, &, * et [] ; pointeurs et tableaux, pointeur NULL, pointeurs et affectation, pointeurs génériques, comparaisons, conversions par cast. **Fonctions.** Définition, déclaration et appel d'une fonction, transmission d'arguments, tableaux transmis en arguments, variables globales et variables locales, pointeurs sur des fonctions. **Entrées-sorties standard.** *printf*, *putchar*, *scanf*, *getchar*. **Chaînes de caractères.** Création, utilisation et modification ; écriture et lecture avec *puts*, *printf*, *gets*, *gets_s*, *scanf* ; fonctions de manipulation de chaînes (*strcpy*, *strcat*...) et de suites d'octets (*memcpy*, *memmove*...) ; fonctions de conversion en numérique (*strtol*...). **Structures, unions, énumérations et champs de bits.** Déclaration, représentation en mémoire, utilisation. **Instruction typedef et synonymes.** **Fichiers.** Fichiers binaires et formats ; *fwrite* et *fread* ; *fprintf*, *fscanf*, *fputs* et *fgets* ; *fputc* et *fgetc* ; accès direct avec *fseek*, *ftell*... ; *fopen* et les modes d'ouverture ; flux prédéfinis *stdin*, *stdout* et *stderr*. **Gestion dynamique de la mémoire.** Principes, fonctions *malloc*, *free*, *calloc*, *realloc* ; exemples d'utilisation : tableaux dynamiques et listes chaînées. **Préprocesseur.** Directives et caractère #, définition de symboles et de macros, directives de compilation conditionnelle, d'inclusion de fichier source, etc. **Déclarations.** Syntaxe générale, spécificateurs, définition de fonction, interprétation de déclarations, écriture de déclarateurs. **Fiabilisation des lectures au clavier.** Utilisation de *scanf*, *gets*, *gets_s*, *fgets*. **Catégories de caractères et fonctions associées.** **Gestion des programmes de grande taille.** Avantages et inconvénients des variables globales, partage d'identificateurs entre plusieurs fichiers source. **Fonctions à arguments variables.** Règles d'écriture, macros *va_start*, *va_arg*, *va_end*, fonctions *vprintf*, *vfprintf* et *vsprintf*. **Communication avec l'environnement.** Arguments reçus par la fonction *main*, terminaison d'un programme, fonctions *getenv* et *system*, signaux. **Caractères étendus.** Type *wchar_t*, fonctions *mblen*, *mbtowc* et *wctomb*, chaînes de caractères étendus. **Localisation.** Mécanisme, fonctions *setlocale* et *localeconv*. **Récursivité.** Principes et exemples, empilement des appels. **Branchements non locaux.** Macros *setjmp* et *longjmp*. **Incompatibilités entre C et C++.** **Bibliothèque standard du C.** *assert.h*, *ctype.h*, *errno.h*, *locale.h*, *math.h*, *setjmp.h*, *signal.h*, *stdarg.h*, *stddef.h*, *stdio.h*, *stdlib.h*, *string.h*, *time.h*. **Nouveautés des normes ISO C99 et C11.** Contraintes supplémentaires, division d'entiers, tableaux de dimension variable, nouveaux types, caractères étendus et Unicode, pointeurs restreints, structures anonymes, expressions génériques, fonctions de vérification du débordement mémoire, threads, etc.

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

Studio Eyrolles © Éditions Eyrolles

Code éditeur : 667922
ISBN : 978-2-212-67922-9

**Le guide complet du
langage C**

CHEZ LE MEME EDITEUR

Du même auteur

C. DELANNOY. **Programmer en langage C.**

N°11825, 5^e édition, 2016, 228 pages.

C. DELANNOY. **Exercices en langage C.**

N°11105, 1997, 222 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, 2^e édition, 2016, 360 pages.

C. DELANNOY. – **Programmer en C++ moderne.**

(10^e édition de l’ouvrage édité précédemment sous le titre *Programmer en langage C++*)

N°67895, 2019, 880 pages.

C. DELANNOY. – **Exercices en langage C++ - 178 exercices corrigés.**

N°67663, 4^e édition, 2018, 396 pages.

C. DELANNOY. – **Programmer en Java (couvre Java 9).**

N°67536, 10^e édition, 2017, 920 pages.

C. DELANNOY. – **Exercices en Java (couvre Java 8).**

N°67385, 4^e édition, 2017, 346 pages.

Sur la programmation système en C sous Linux

C. BLAESS. – **Développement système sous Linux.**

Ordonnancement multitâches, gestion mémoire, communications, programmation réseau.

N°67760, 5^e édition, 1 062 pages, 2019.

Dans la même collection

A. TASSO. – **Le livre de Java premier langage Avec 109 exercices corrigés.**

N°67840, 13^e édition, 2019, 600 pages.

G. SWINNEN. – **Apprendre à programmer avec Python 3.**

N°13434, 3^e édition, 2012, 435 pages.

H. BERSINI, I. WELLESZ. – **La programmation orientée objet.**

Cours et exercices en UML 2 avec Java, C#, C++, Python, PHP et LinQ.

N°67399, 7^e édition, 2017, 696 pages.

J. ENGELS. – **PHP 7.**

N°67360, 2017, 585 pages.

P. ROQUES. – **UML 2.5 par la pratique. Études de cas et exercices corrigés.**

N°67565, 8^e édition, 2018, 408 pages.

C. SOUTOU. – **Programmer avec MySQL.**

N°67379, 5^e édition, 2017, 522 pages.

Claude Delannoy

Le guide complet du langage C

Deuxième tirage 2020 avec nouvelle présentation

● Éditions
EYROLLES

Éditions Eyrolles
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Le présent ouvrage est une nouvelle édition du livre publié à l'origine sous le titre
« La référence du C norme ANSI/ISO », puis au format semi-poche sous le titre « Langage C ».
Il bénéficie à l'occasion de ce retraitage d'une nouvelle maquette de couverture.

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 l'autorisation de l'Éditeur ou du Centre fran-
çais d'exploitation du droit de copie, 20, rue des Grands-Augustins, 75006 Paris.

© Groupe Eyrolles, 1999, 2008, 2014, pour le texte de la présente édition.
© Éditions Eyrolles, 2020, pour la nouvelle présentation. ISBN : 978-2-212-67922-9

Table des matières

Avant-propos	1
À qui s'adresse ce livre ?	1
Structure de l'ouvrage	2
À propos des normes ANSI/ISO	4
À propos de la fonction main.	4
Remerciements	5
 CHAPITRE 1	
Généralités	7
1. Historique du langage C.	7
2. Programme source, module objet et programme exécutable	8
3. Compilation en C : existence d'un préprocesseur	9
4. Variable et objet	10
4.1 Définition d'une variable et d'un objet	10
4.2 Utilisation d'un objet	10
5. Lien entre objet, octets et caractères.	12
6. Classe d'allocation des variables	12
 CHAPITRE 2	
Les éléments constitutifs d'un programme source	15
1. Jeu de caractères source et jeu de caractères d'exécution	16
1.1 Généralités	16
1.2 Commentaires à propos du jeu de caractères source	17
1.3 Commentaires à propos du jeu minimal de caractères d'exécution	18

2. Les identificateurs	19
3. Les mots-clés	20
4. Les séparateurs et les espaces blancs	20
5. Le format libre	21
6. Les commentaires	22
7. Notion de token	23
7.1 Les différentes catégories de tokens	23
7.2 Décomposition en tokens	25
CHAPITRE 3	
Les types de base	27
1. Les types entiers	27
1.1 Les six types entiers	28
1.2 Représentation mémoire des entiers et limitations	29
1.3 Critères de choix d'un type entier	32
1.4 Écriture des constantes entières	33
1.5 Le type attribué par le compilateur aux constantes entières	34
1.6 Exemple d'utilisation déraisonnable de constantes hexadécimales	35
1.7 Pour imposer un type aux constantes entières	36
1.8 En cas de dépassement de capacité dans l'écriture des constantes entières	36
2. Les types caractère	37
2.1 Les deux types caractère	37
2.2 Caractéristiques des types caractère	38
2.3 Écriture des constantes caractère	40
2.4 Le type des constantes caractère	43
3. Le fichier limits.h	45
3.1 Son contenu	45
3.2 Précautions d'utilisation	46
4. Les types flottants	46
4.1 Rappels concernant le codage des nombres en flottant	46
4.2 Le modèle proposé par la norme	47
4.3 Les caractéristiques du codage en flottant	48
4.4 Représentation mémoire et limitations	50
4.5 Écriture des constantes flottantes	51

4.6 Le type des constantes flottantes	52
4.7 En cas de dépassement de capacité dans l'écriture des constantes	52
5. Le fichier float.h	53
6. Déclarations des variables d'un type de base	54
6.1 Rôle d'une déclaration	55
6.2 Initialisation lors de la déclaration	56
6.3 Les qualifieurs const et volatile	57
CHAPITRE 4	
Les opérateurs et les expressions	61
1. Généralités	62
1.1 Les particularités des opérateurs et des expressions en C	62
1.2 Priorité et associativité	63
1.3 Pluralité	64
1.4 Conversions implicites	64
1.5 Les différentes catégories d'opérateurs	65
2. Les opérateurs arithmétiques	65
2.1 Les différents opérateurs numériques	66
2.2 Comportement en cas d'exception	69
3. Les conversions numériques implicites	74
3.1 Introduction	74
3.2 Les conversions numériques d'ajustement de type	75
3.3 Les promotions numériques	80
3.4 Combinaisons de conversions	86
3.5 Cas particulier des arguments d'une fonction	88
4. Les opérateurs relationnels	89
4.1 Généralités	89
4.2 Les six opérateurs relationnels du langage C	90
4.3 Leur priorité et leur associativité	91
5. Les opérateurs logiques	93
5.1 Généralités	93
5.2 Les trois opérateurs logiques du langage C	94
5.3 Leur priorité et leur associativité	95
5.4 Les opérandes de && et de ne sont évalués que si nécessaire	96

6. Les opérateurs de manipulation de bits	97
6.1 Présentation des opérateurs de manipulation de bits	97
6.2 Les opérateurs « bit à bit »	98
6.3 Les opérateurs de décalage.	100
6.4 Applications usuelles des opérateurs de manipulation de bits	102
7. Les opérateurs d'affectation et d'incrémentat	104
7.1 Généralités	104
7.2 La lvalue	105
7.3 L'opérateur d'affectation simple	106
7.4 Tableau récapitulatif : l'opérateur d'affectation simple.	108
7.5 Les opérateurs d'affectation élargie	109
8. Les opérateurs de cast	110
8.1 Généralités	110
8.2 Les opérateurs de cast	111
9. Le rôle des conversions numériques	113
9.1 Conversion d'un type flottant vers un autre type flottant	113
9.2 Conversion d'un type flottant vers un type entier.	114
9.3 Conversion d'un type entier vers un type flottant.	114
9.4 Conversion d'un type entier vers un autre type entier	115
9.5 Cas particuliers des conversions d'entier vers caractère.	116
9.6 Tableau récapitulatif des conversions numériques	118
10. L'opérateur conditionnel	119
10.1 Introduction.	119
10.2 Rôle de l'opérateur conditionnel	120
10.3 Contraintes et conversions	120
10.4 La priorité de l'opérateur conditionnel	122
11. L'opérateur séquentiel	122
12. L'opérateur sizeof	124
12.1 L'opérateur sizeof appliqué à un nom de type	124
12.2 L'opérateur sizeof appliqué à une expression	125
13. Tableau récapitulatif : priorités et associativité des opérateurs.	127
14. Les expressions constantes.	128
14.1 Introduction.	128
14.2 Les expressions constantes d'une manière générale.	129

CHAPITRE 5

Les instructions exécutables	133
1. Généralités	133
1.1 Rappels sur les instructions de contrôle	134
1.2 Classification des instructions exécutables du langage C	134
2. L'instruction expression	136
2.1 Syntaxe et rôle	136
2.2 Commentaires	136
3. L'instruction composée ou bloc	137
3.1 Syntaxe d'un bloc	137
3.2 Commentaires	138
3.3 Déclarations dans un bloc	139
3.4 Cas des branchements à l'intérieur d'un bloc	140
4. L'instruction if	140
4.1 Syntaxe et rôle de l'instruction if	140
4.2 Exemples d'utilisation	141
4.3 Cas des if imbriqués	144
4.4 Traduction de choix en cascade	145
5. L'instruction switch	147
5.1 Exemple introductif	147
5.2 Syntaxe usuelle et rôle de switch	148
5.3 Commentaires	149
5.4 Quelques curiosités de l'instruction switch	150
6. Choix entre if et switch	152
7. Les particularités des boucles en C	153
7.1 Rappels concernant la programmation structurée	153
7.2 Les boucles en C	154
8. L'instruction do ... while	155
8.1 Syntaxe	155
8.2 Rôle	156
8.3 Exemples d'utilisation	157
9. L'instruction while	158
9.1 Syntaxe	159
9.2 Rôle	159

9.3 Lien entre while et do ... while	160
9.4 Exemples d'utilisation.	161
10. L'instruction for	162
10.1 Introduction.	162
10.2 Syntaxe	163
10.3 Rôle.	163
10.4 Lien entre for et while.	164
10.5 Commentaires.	165
10.6 Exemples d'utilisation.	166
11. Conseils d'utilisation des différents types de boucles	168
11.1 Boucle définie	168
11.2 Boucle indéfinie	169
12. L'instruction break.	170
12.1 syntaxe et rôle.	170
12.2 Exemple d'utilisation	170
12.3 Commentaires.	171
13. L'instruction continue	172
13.1 Syntaxe et rôle.	172
13.2 Exemples d'utilisation.	172
13.3 Commentaires.	173
14. Quelques schémas de boucles utiles	175
14.1 Boucle à sortie intermédiaire	175
14.2 Boucles à sorties multiples	178
15. L'instruction goto et les étiquettes.	179
15.1 Les étiquettes	179
15.2 Syntaxe et rôle	180
15.3 Exemples et commentaires	181
 CHAPITRE 6	
Les tableaux.	185
1. Exemple introductif d'utilisation d'un tableau	186
2. Déclaration des tableaux	187
2.1 Généralités	187
2.2 Le type des éléments d'un tableau	188

2.3 Déclarateur de tableau	188
2.4 La dimension d'un tableau	190
2.5 Classe de mémorisation associée à la déclaration d'un tableau	192
2.6 Les qualifieurs const et volatile	193
2.7 Nom de type correspondant à un tableau	194
3. Utilisation d'un tableau	195
3.1 Les indices	195
3.2 Un identificateur de tableau n'est pas une lvalue	196
3.3 Utilisation d'un élément d'un tableau	196
3.4 L'opérateur sizeof et les tableaux	197
4. Arrangement d'un tableau et débordement d'indice.	198
4.1 Les éléments d'un tableau sont alloués de manière consécutive	198
4.2 Aucun contrôle n'est effectué sur la valeur de l'indice	199
5. Cas des tableaux de tableaux.	200
5.1 Déclaration des tableaux à deux indices	200
5.2 Utilisation d'un tableau à deux indices	200
5.3 Peut-on parler de lignes et de colonnes d'un tableau à deux indices ?	202
5.4 Arrangement en mémoire d'un tableau à deux indices	202
5.5 Cas des tableaux à plus de deux indices	203
6. Initialisation de tableaux	204
6.1 Initialisation par défaut des tableaux	205
6.2 Initialisation explicite des tableaux	205
 CHAPITRE 7	
Les pointeurs	211
1. Introduction à la notion de pointeur	212
1.1 Attribuer une valeur à une variable de type pointeur	212
1.2 L'opérateur * pour manipuler un objet pointé	213
2. Déclaration des variables de type pointeur	214
2.1 Généralités	215
2.2 Le type des objets désignés par un pointeur	216
2.3 Déclarateur de pointeur	216
2.4 Classe de mémorisation associée à la déclaration d'un pointeur	219
2.5 Les qualifieurs const et volatile	219
2.6 Nom de type correspondant à un pointeur	223

3. Les propriétés des pointeurs	224
3.1 Les propriétés arithmétiques des pointeurs	225
3.2 Lien entre pointeurs et tableaux	226
3.3 Ordre des pointeurs et ordre des adresses	232
3.4 Les restrictions imposées à l'arithmétique des pointeurs	232
4. Tableaux récapitulatifs : les opérateurs +, -, &, * et []	235
5. Le pointeur NULL	237
6. Pointeurs et affectation	239
6.1 Prise en compte des qualifieurs des objets pointés	239
6.2 Les autres possibilités d'affectation	241
6.3 Tableau récapitulatif	242
6.4 Les affectations élargies += et -= et les incréments ++ et --	242
7. Les pointeurs génériques	243
7.1 Généralités	243
7.2 Déclaration du type void *	244
7.3 Interdictions propres au type void *	244
7.4 Possibilités propres au type void *	245
8. Comparaisons de pointeurs	246
8.1 Comparaisons basées sur un ordre	247
8.2 Comparaisons d'égalité ou d'inégalité	248
8.3 Récapitulatif : les comparaisons dans un contexte pointeur	249
9. Conversions de pointeurs par cast.	249
9.1 Conversion d'un pointeur en un pointeur d'un autre type	250
9.2 Conversions entre entiers et pointeurs	252
9.3 Récapitulatif concernant l'opérateur de cast dans un contexte pointeur	253

CHAPITRE 8

Les fonctions	255
1. Les fonctions en C.	256
1.1 Une seule sorte de module en C : la fonction	256
1.2 Fonction et transmission des arguments par valeur	258
1.3 Les variables globales	258
1.4 Les possibilités de compilation séparée	259
2. Exemple introductif de la notion de fonction en langage C	259

3. Définition d'une fonction	261
3.1 Les deux formes de l'en-tête	262
3.2 Les arguments apparaissant dans l'en-tête	262
3.3 La valeur de retour	264
3.4 Classe de mémorisation d'une fonction : extern et static	268
3.5 L'instruction return	269
4. Déclaration et appel d'une fonction	272
4.1 Déclaration sous forme de prototype	273
4.2 Déclaration partielle (déconseillée)	274
4.3 Portée d'une déclaration de fonction	275
4.4 Redéclaration d'une fonction	276
4.5 Une définition de fonction tient lieu de déclaration	277
4.6 En cas d'absence de déclaration	278
4.7 Utilisation de la déclaration dans la traduction d'un appel.	278
4.8 En cas de non-concordance entre arguments muets et arguments effectifs	281
4.9 Les fichiers en-tête standards	282
4.10 Nom de type correspondant à une fonction	283
5. Le mécanisme de transmission d'arguments	284
5.1 Cas où la transmission par valeur est satisfaisante	284
5.2 Cas où la transmission par valeur n'est plus satisfaisante.	285
5.3 Comment simuler une transmission par adresse avec des pointeurs	287
6. Cas des tableaux transmis en arguments	289
6.1 Règles générales	289
6.2 Exemples d'applications	294
6.3 Pour qu'une fonction dispose de la dimension d'un tableau.	296
6.4 Quelques conseils de style à propos des tableaux en argument	298
7. Cas particulier des tableaux de tableaux transmis en arguments	299
7.1 Application des règles générales.	300
7.2 Artifices facilitant la manipulation de tableaux de dimensions variables	304
8. Les variables globales	310
8.1 Exemples introductifs d'utilisation de variables globales.	311
8.2 Les déclarations des variables globales	313
8.3 Portée des variables globales	319
8.4 Variables globales et édition de liens	320
8.5 Les variables globales sont de classe statique	321
8.6 Initialisation des variables globales	322

9. Les variables locales	324
9.1 La portée des variables locales	324
9.2 Classe d'allocation et initialisation des variables locales	326
10. Tableau récapitulatif : portée, accès et classe d'allocation des variables	331
11. Pointeurs sur des fonctions	332
11.1 Déclaration d'une variable pointeur sur une fonction	333
11.2 Affectation de valeurs à une variable pointeur sur une fonction	334
11.3 Appel d'une fonction par le biais d'un pointeur	337
11.4 Exemple de paramétrage d'appel de fonctions	339
11.5 Transmission de fonction en argument	341
11.6 Comparaisons de pointeurs sur des fonctions	342
11.7 Conversions par cast de pointeurs sur des fonctions	342
CHAPITRE 9	
Les entrées-sorties standards	345
1. Caractéristiques générales des entrées-sorties standards.	346
1.1 Mode d'interaction avec l'utilisateur	346
1.2 Formatage des informations échangées	347
1.3 Généralisation aux fichiers de type texte	347
2. Présentation générale de printf	348
2.1 Notions de format d'entrée, de code de format et de code de conversion	349
2.2 L'appel de printf	350
2.3 Les risques d'erreurs dans la rédaction du format	352
3. Les principales possibilités de formatage de printf	355
3.1 Le gabarit d'affichage	355
3.2 Précision des informations flottantes	358
3.3 Justification des informations	360
3.4 Gabarit ou précision variable	360
3.5 Le code de format g	362
3.6 Le drapeau + force la présence d'un signe « plus »	365
3.7 Le drapeau espace force la présence d'un espace	366
3.8 Le drapeau 0 permet d'afficher des zéros de remplissage	366
3.9 Le paramètre de précision permet de limiter l'affichage des chaînes	367
3.10 Cas particulier du type unsigned short int : le modificateur h	368

4. Description des codes de format des fonctions de la famille printf	369
4.1 Structure générale d'un code de format	369
4.2 Le paramètre drapeaux	369
4.3 Le paramètre de gabarit	370
4.4 Le paramètre de précision	372
4.5 Le paramètre modificateur h/l/L	373
4.6 Les codes de conversion	373
4.7 Les codes utilisables avec un type donné	375
5. La fonction putchar	376
5.1 Prototype	377
5.2 L'argument de putchar est de type int	377
5.3 La valeur de retour de putchar	378
6. Présentation générale de scanf	378
6.1 Format de sortie, code de format et code de conversion	378
6.2 L'appel de scanf	379
6.3 Les risques d'erreurs dans la rédaction du format	380
6.4 La fonction scanf utilise un tampon	383
6.5 Notion de caractère invalide et d'arrêt prématuré	384
6.6 La valeur de retour de scanf	385
6.7 Exemples de rencontre de caractères invalides	387
7. Les principales possibilités de scanf	389
7.1 La présentation des informations lues en données	390
7.2 Limitation du gabarit	391
7.3 La fin de ligne joue un rôle ambigu : séparateur ou caractère	392
7.4 Lorsque le format impose certains caractères dans les données	394
7.5 Attention au faux gabarit du code C	394
7.6 Les codes de format de la forme %[...]	395
8. Description des codes de format des fonctions de la famille de scanf	397
8.1 Récapitulatif des règles utilisées par ces fonctions	397
8.2 Structure générale d'un code de format	398
8.3 Les paramètres * et gabarit	399
8.4 Le paramètre modificateur h/l/L	399
8.5 Les codes de conversion	400
8.6 Les codes utilisables avec un type donné	402
8.7 Les différences entre les codes de format en entrée et en sortie	403

9. La fonction <code>getchar</code>	404
9.1 Prototype et valeur de retour	404
9.2 Précautions	404

CHAPITRE 10

Les chaînes de caractères	407
1. Règles générales d'écriture des constantes chaîne	408
1.1 Notation des constantes chaîne	408
1.2 Concaténation des constantes chaîne adjacentes	409
2. Propriétés des constantes chaîne	410
2.1 Conventions de représentation	411
2.2 Emplacement mémoire	412
2.3 Cas des chaînes identiques	413
2.4 Les risques de modification des constantes chaîne	413
2.5 Simulation d'un tableau de constantes chaîne	415
3. Créer, utiliser ou modifier une chaîne	416
3.1 Comment disposer d'un emplacement pour y ranger une chaîne	417
3.2 Comment agir sur le contenu d'une chaîne	418
3.3 Comment utiliser une chaîne existante	421
4. Entrées-sorties standards de chaînes	423
4.1 Généralités	423
4.2 Écriture de chaînes avec <code>puts</code>	424
4.3 Écriture de chaînes avec le code de format <code>%s</code> de <code>printf</code> ou <code>fprintf</code>	425
4.4 Lecture de chaînes avec <code>gets</code>	426
4.5 Lecture de chaînes avec le code de format <code>%s</code> dans <code>scanf</code> ou <code>fscanf</code>	429
4.6 Comparaison entre <code>gets</code> et <code>scanf</code> dans les lectures de chaînes	430
4.7 Limitation de la longueur des chaînes lues sur l'entrée standard	431
5. Généralités concernant les fonctions de manipulation de chaînes	435
5.1 Ces fonctions travaillent toujours sur des adresses	435
5.2 Les adresses sont toujours de type <code>char *</code>	435
5.3 Certains arguments sont déclarés <code>const</code> , d'autres pas	436
5.4 Attention aux valeurs des arguments de limitation de longueur	437
5.5 La fonction <code>strlen</code>	438

6. Les fonctions de copie de chaînes	439
6.1 Généralités	439
6.2 La fonction strcpy	439
6.3 La fonction strncpy	443
7. Les fonctions de concaténation de chaînes	446
7.1 Généralités	446
7.2 La fonction strcat	446
7.3 La fonction strncat	449
8. Les fonctions de comparaison de chaînes	452
8.1 Généralités	452
8.2 La fonction strcmp	453
8.3 La fonction strncmp	454
9. Les fonctions de recherche dans une chaîne	454
9.1 Les fonctions de recherche d'un caractère : strchr et strchr	455
9.2 La fonction de recherche d'une sous-chaîne : strstr	458
9.3 La fonction de recherche d'un caractère parmi plusieurs : strpbrk	458
9.4 Les fonctions de recherche d'un préfixe	459
9.5 La fonction d'éclatement d'une chaîne : strtok	461
10. Les fonctions de conversion d'une chaîne en un nombre	464
10.1 Généralités	464
10.2 La fonction de conversion d'une chaîne en un double : strtod	465
10.3 Les fonctions de conversion d'une chaîne en entier : strtol et strtoul	469
10.4 Cas particulier des fonctions atof, atoi et atol	475
11. Les fonctions de manipulation de suites d'octets	475
11.1 Généralités	476
11.2 Les fonctions de recopie de suites d'octets	476
11.3 La fonction memcmp de comparaison de deux suites d'octets	477
11.4 La fonction memset d'initialisation d'une suite d'octets	478
11.5 La fonction memchr de recherche d'une valeur dans une suite d'octets	479
 CHAPITRE 11	
Les types structure, union et énumération	481
1. Exemples introductifs	482
1.1 Exemple d'utilisation d'une structure	482
1.2 Exemple d'utilisation d'une union	484

2. La déclaration des structures et des unions	486
2.1 Définition conseillée d'un type structure ou union	486
2.2 Déclaration de variables utilisant des types structure ou union.	490
2.3 Déclaration partielle ou déclaration anticipée	492
2.4 Mixage entre définition et déclaration	493
2.5 L'espace de noms des identificateurs de champs	494
2.6 L'espace de noms des identificateurs de types	495
3. Représentation en mémoire d'une structure ou d'une union	495
3.1 Contraintes générales	496
3.2 Cas des structures.	497
3.3 Cas des unions	498
3.4 L'opérateur sizeof appliqué aux structures ou aux unions	499
4. Utilisation d'objets de type structure ou union	499
4.1 Manipulation individuelle des différents champs d'une structure ou d'une union.	500
4.2 Affectation globale entre structures ou unions de même type.	501
4.3 L'opérateur & appliqué aux structures ou aux unions	503
4.4 Comparaison entre pointeurs sur des champs	503
4.5 Comparaison des structures ou des unions par == ou != impossible.	504
4.6 L'opérateur ->	504
4.7 Structure ou union transmise en argument ou en valeur de retour.	505
5. Exemples d'objets utilisant des structures	508
5.1 Structures comportant des tableaux	508
5.2 Structures comportant d'autres structures	509
5.3 Tableaux de structures	510
5.4 Structure comportant des pointeurs sur des structures de son propre type	511
6. Initialisation de structures ou d'unions	511
6.1 Initialisation par défaut des structures ou des unions.	512
6.2 Initialisation explicite des structures	513
6.3 L'initialisation explicite d'une union	517
7. Les champs de bits	517
7.1 Introduction.	517
7.2 Exemples introductifs	518
7.3 Les champs de bits d'une manière générale	519
7.4 Exemple d'utilisation d'une structure de champs de bits dans une union	522

8. Les énumérations	523
8.1 Exemples introductifs	523
8.2 Déclarations associées aux énumérations	526
CHAPITRE 12	
La définition de synonymes avec typedef	529
1. Exemples introductifs	530
1.1 Définition d'un synonyme de int.	530
1.2 Définition d'un synonyme de int *	530
1.3 Définition d'un synonyme de int[3]	531
1.4 Définition d'un synonyme d'un type structure	532
2. L'instruction typedef d'une manière générale	532
2.1 Syntaxe	532
2.2 Définition de plusieurs synonymes	533
2.3 Imbrication des définitions de synonyme	534
3. Utilisation de synonymes	534
3.1 Un synonyme peut s'utiliser comme spécificateur de type	534
3.2 Un synonyme n'est pas un nouveau type	535
3.3 Un synonyme peut s'utiliser à la place d'un nom de type	535
4. Les limitations de l'instruction typedef	536
4.1 Limitations liées à la syntaxe de typedef	536
4.2 Cas des tableaux sans dimension	537
4.3 Cas des synonymes de type fonction	538
CHAPITRE 13	
Les fichiers	541
1. Généralités concernant le traitement des fichiers	542
1.1 Notion d'enregistrement	542
1.2 Archivage de l'information sous forme binaire ou formatée	542
1.3 Accès séquentiel ou accès direct	543
1.4 Fichiers et implémentation	544
2. Le traitement des fichiers en C	544
2.1 L'absence de la notion d'enregistrement en C	545
2.2 Notion de flux	545
2.3 Distinction entre fichier binaire et fichier formaté	547

2.4 Opérations applicables à un fichier et choix du mode d'ouverture	549
2.5 Accès séquentiel et accès direct	551
2.6 Le tampon et sa gestion	551
3. Le traitement des erreurs de gestion de fichier	552
3.1 Introduction	552
3.2 La détection des erreurs en C	553
4. Les entrées-sorties binaires : fwrite et fread	556
4.1 Exemple introductif de création séquentielle d'un fichier binaire	556
4.2 Exemple introductif de liste séquentielle d'un fichier binaire	559
4.3 La fonction fwrite	561
4.4 La fonction fread	564
5. Les opérations formatées avec fprintf, fscanf, fputs et fgets	568
5.1 Exemple introductif de création séquentielle d'un fichier formaté	569
5.2 Exemple introductif de liste séquentielle d'un fichier formaté	571
5.3 La fonction fprintf	573
5.4 La fonction fscanf	575
5.5 La fonction fputs	579
5.6 La fonction fgets	582
6. Les opérations mixtes portant sur des caractères	585
6.1 La fonction fputc et la macro putc	586
6.2 La fonction fgetc et la macro getc	589
7. L'accès direct	593
7.1 Exemple introductif d'accès direct à un fichier binaire existant	593
7.2 La fonction fseek	595
7.3 La fonction ftell	597
7.4 Les possibilités de l'accès direct	598
7.5 Détection des erreurs supplémentaires liées à l'accès direct	600
7.6 Exemple d'accès indexé à un fichier formaté	603
7.7 Les fonctions fsetpos et fgetpos	605
8. La fonction fopen et les différents modes d'ouverture d'un fichier	605
8.1 Généralités	605
8.2 La fonction fopen	606
9. Les flux prédéfinis	609

CHAPITRE 14

La gestion dynamique	611
1. Intérêt de la gestion dynamique	611
2. Exemples introductifs	612
2.1 Allocation et utilisation d'un objet de type double	612
2.2 Cas particulier d'un tableau	614
3. Caractéristiques générales de la gestion dynamique	615
3.1 Absence de typage des objets	616
3.2 Notation des objets	616
3.3 Risques et limitations	617
3.4 Limitations	618
4. La fonction malloc	618
4.1 Prototype	618
4.2 La valeur de retour et la gestion des erreurs	619
5. La fonction free	620
6. La fonction calloc	620
6.1 Prototype	621
6.2 Rôle	621
6.3 Valeur de retour et gestion des erreurs	621
6.4 Précautions	622
7. La fonction realloc	623
7.1 Exemples introductifs	623
7.2 Prototype	624
7.3 Rôle	624
7.4 Valeur de retour	625
7.5 Précautions	626
8. Techniques utilisant la gestion dynamique	626
8.1 Gestion de tableaux dont la taille n'est connue qu'au moment de l'exécution	626
8.2 Gestion de tableaux dont la taille varie pendant l'exécution	627
8.3 Gestion de listes chaînées	628

CHAPITRE 15

Le préprocesseur	631
1. Généralités.	631
1.1 Les directives tiennent compte de la notion de ligne	631
1.2 Les directives et le caractère #	632
1.3 La notion de token pour le préprocesseur.	632
1.4 Classification des différentes directives du préprocesseur	633
2. La directive de définition de symboles et de macros	634
2.1 Exemples introductifs.	634
2.2 La syntaxe de la directive #define	637
2.3 Règles d'expansion d'un symbole ou d'une macro	640
2.4 L'opérateur de conversion en chaîne : #	650
2.5 L'opérateur de concaténation de tokens : ##.	653
2.6 Exemple faisant intervenir les deux opérateurs # et ##	655
2.7 La directive #undef.	655
2.8 Précautions à prendre	656
2.9 Les symboles prédéfinis	660
3. Les directives de compilation conditionnelle	661
3.1 Compilation conditionnelle fondée sur l'existence de symboles	661
3.2 Compilation conditionnelle fondée sur des expressions	664
3.3 Imbrication des directives de compilation conditionnelle	670
3.4 Exemples d'utilisation des directives de compilation conditionnelle	670
4. La directive d'inclusion de fichier source	672
4.1 Généralités	672
4.2 Syntaxe	673
4.3 Précautions à prendre	674
5. Directives diverses	677
5.1 La directive vide	677
5.2 La directive #line	678
5.3 La directive #error	678
5.4 La directive #pragma	679

CHAPITRE 16

Les déclarations	681
1. Généralités	681
1.1 Les principaux éléments : déclarateur et spécificateur de type	682
1.2 Les autres éléments	683
2. Syntaxe générale d'une déclaration	684
2.1 Forme générale d'une déclaration	685
2.2 Spécificateur de type structure	686
2.3 Spécificateur de type union	688
2.4 Spécificateur de type énumération	689
2.5 Déclarateur	689
3. Définition de fonction	691
3.1 Forme moderne de la définition d'une fonction	691
3.2 Forme ancienne de la définition d'une fonction	692
4. Interprétation de déclarations	692
4.1 Les règles	692
4.2 Exemples	693
5. Écriture de déclarateurs	695
5.1 Les règles	695
5.2 Exemples	696

CHAPITRE 17

Fiabilisation des lectures au clavier	699
1. Généralités	699
2. Utilisation de scanf	701
3. Utilisation de gets	702
4. Utilisation de fgets	703
4.1 Pour éviter le risque de débordement en mémoire	703
4.2 Pour ignorer les caractères excédentaires	704
4.3 Pour traiter l'éventuelle fin de fichier et paramétrer la taille des chaînes lues	706

CHAPITRE 18

Les catégories de caractères et les fonctions associées	709
1. Généralités	709
1.1 Dépendance de l'implémentation et de la localisation	709
1.2 Les fonctions de test	710
2. Les catégories de caractères	710
3. Exemples	713
3.1 Pour obtenir la liste de tous les caractères imprimables et leur code	713
3.2 Pour connaître les catégories des caractères d'une implémentation	714
4. Les fonctions de transformation de caractères	715

CHAPITRE 19

Gestion des gros programmes	717
1. Utilisation de variables globales	718
1.1 Avantages des variables globales	719
1.2 Inconvénients des variables globales	719
1.3 Conseils en forme de compromis	720
2. Partage d'identificateurs entre plusieurs fichiers source	721
2.1 Cas des identificateurs de fonctions	721
2.2 Cas des identificateurs de types ou de synonymes	722
2.3 Cas des variables globales	723

CHAPITRE 20

Les arguments variables	725
1. Écriture de fonctions à arguments variables	725
1.1 Exemple introductif	725
1.2 Arguments variables, forme d'en-tête et déclaration	727
1.3 Contraintes imposées par la norme	728
1.4 Syntaxe et rôle des macros <code>va_start</code> , <code>va_arg</code> et <code>va_end</code>	729
2. Transmission d'une liste variable	730
3. Les fonctions <code>vprintf</code>, <code>vfprintf</code> et <code>vsprintf</code>	732

CHAPITRE 21

Communication avec l'environnement	735
1. Cas particulier des programmes autonomes	735
2. Les arguments reçus par la fonction main	736
2.1 L'en-tête de la fonction main	736
2.2 Récupération des arguments reçus par la fonction main	737
3. Terminaison d'un programme	738
3.1 Les fonctions exit et atexit	738
3.2 L'instruction return dans la fonction main	739
4. Communication avec l'environnement	740
4.1 La fonction getenv	740
4.2 La fonction system	741
5. Les signaux	741
5.1 Généralités	741
5.2 Exemple introductif	742
5.3 La fonction signal	744
5.4 La fonction raise	746

CHAPITRE 22

Les caractères étendus	747
1. Le type wchar_t et les caractères multioctets	748
2. Notation des constantes du type wchar_t	749
3. Les fonctions liées aux caractères étendus mblen, mbtowc et wctomb ...	750
3.1 Généralités	750
3.2 La fonction mblen	750
3.3 La fonction mbtowc	751
3.4 La fonction wctomb	751
4. Les chaînes de caractères étendus	752
5. Représentation des constantes chaînes de caractères étendus	753
6. Les fonctions liées aux chaînes de caractères étendus : mbstowcs et wctombs	753
6.1 La fonction mbstowcs	754
6.2 La fonction wctombs	754

CHAPITRE 23

Les adaptations locales	755
1. Le mécanisme de localisation	755
2. La fonction setlocale	756
3. La fonction localeconv	758

CHAPITRE 24

La récursivité	761
1. Notion de récursivité	761
2. Exemple de fonction récursive	762
3. L’empilement des appels	763
4. Autre exemple de récursivité	765

CHAPITRE 25

Les branchements non locaux	769
1. Exemple introductif	769
2. La macro setjmp et la fonction longjmp	770
2.1 Prototypes et rôles	770
2.2 Contraintes d’utilisation	771

CHAPITRE 26

Les incompatibilités entre C et C++	773
1. Les incompatibilités raisonnables	773
1.1 Définition d’une fonction	773
1.2 Les prototypes en C++	774
1.3 Fonctions sans valeur de retour	774
1.4 Compatibilité entre le type void * et les autres pointeurs	774
1.5 Les déclarations multiples	775
1.6 L’instruction goto	775
1.7 Initialisation de tableaux de caractères	776
2. Les incompatibilités incontournables	776
2.1 Fonctions sans arguments	776
2.2 Le qualifieur const	776
2.3 Les constantes de type caractère	778

ANNEXE A

La bibliothèque standard C90	779
1. Généralités	780
1.1 Les différents fichiers en-tête	780
1.2 Redéfinition d'une macro standard par une fonction	781
2. Assert.h : macro de mise au point	781
3. Ctype.h : tests de caractères et conversions majuscules - minuscules	782
3.1 Les fonctions de test d'appartenance d'un caractère à une catégorie	782
3.2 Les fonctions de transformation de caractères	783
4. Errno.h : gestion des erreurs	783
4.1 Constantes prédéfinies	783
4.2 Macros	783
5. Locale.h : caractéristiques locales	784
5.1 Types prédéfinis	784
5.2 Constantes prédéfinies	784
5.3 Fonctions	784
6. Math.h : fonctions mathématiques	784
6.1 Constantes prédéfinies	784
6.2 Traitement des conditions d'erreur	785
6.3 Fonctions trigonométriques	785
6.4 Fonctions hyperboliques	786
6.5 Fonctions exponentielle et logarithme	786
6.6 Fonctions puissance	787
6.7 Autres fonctions	787
7. Setjmp.h : branchements non locaux	788
7.1 Types prédéfinis	788
7.2 Fonctions et macros	788
8. Signal.h : traitement de signaux	788
8.1 Types prédéfinis	788
8.2 Constantes prédéfinies	788
8.3 Fonctions de traitement de signaux	789
9. Stdarg.h : gestion d'arguments variables	789
9.1 Types prédéfinis	789
9.2 Macros	789

10. Stddef.h : définitions communes	790
10.1 Types prédéfinis	790
10.2 Constantes prédéfinies	790
10.3 Macros prédéfinies	790
11. Stdio.h : entrées-sorties	790
11.1 Types prédéfinis	790
11.2 Constantes prédéfinies	790
11.3 Fonctions d'opérations sur les fichiers	791
11.4 Fonctions d'accès aux fichiers	792
11.5 Fonctions d'écriture formatée	793
11.6 Fonctions de lecture formatée	794
11.7 Fonctions d'entrées-sorties de caractères	795
11.8 Fonctions d'entrées-sorties sans formatage	797
11.9 Fonctions agissant sur le pointeur de fichier	798
11.10 Fonctions de gestion des erreurs d'entrée-sortie	799
12. Stdlib.h : utilitaires	800
12.1 Types prédéfinis	800
12.2 Constantes prédéfinies	800
12.3 Fonctions de conversion de chaîne	800
12.4 Fonctions de génération de séquences de nombres pseudo aléatoires	802
12.5 Fonctions de gestion de la mémoire	803
12.6 Fonctions de communication avec l'environnement	804
12.7 Fonctions de tri et de recherche	805
12.8 Fonctions liées à l'arithmétique entière	805
12.9 Fonctions liées aux caractères étendus	806
12.10 Fonctions liées aux chaînes de caractères étendus	806
13. String.h : manipulations de suites de caractères	807
13.1 Types prédéfinis	807
13.2 Constantes prédéfinies	807
13.3 Fonctions de copie	807
13.4 Fonctions de concaténation	808
13.5 Fonctions de comparaison	808
13.6 Fonctions de recherche	809
13.7 Fonctions diverses	810

14. Time.h : gestion de l'heure et de la date	811
14.1 Types prédéfinis	811
14.2 Constantes prédéfinies	812
14.3 Fonctions de manipulation de temps	812
14.4 Fonctions de conversion	813

ANNEXE B

Les normes C99 et C11	815
1. Contraintes supplémentaires (C99)	815
1.1 Type de retour d'une fonction	815
1.2 Déclaration implicite d'une fonction	815
1.3 Instruction return	816
2. Division d'entiers (C99)	816
3. Emplacement des déclarations (C99)	816
4. Commentaires de fin de ligne (C99)	817
5. Tableaux de dimension variable (C99, facultatif en C11)	818
5.1 Dans les déclarations	818
5.2 Dans les en-têtes de fonctions et leurs prototypes	818
6. Nouveaux types (C99)	819
6.1 Nouveau type entier long long (C99)	819
6.2 Types entiers étendus (C99)	819
6.3 Nouveaux types flottants (C99)	820
6.4 Le type booléen (C99)	820
6.5 Les types complexes (C99, facultatif en C11)	821
7. Nouvelles fonctions mathématiques (C99)	822
7.1 Généralisation aux trois types flottants (C99)	823
7.2 Nouvelles fonctions (C99)	823
7.3 Fonctions mathématiques génériques (C99)	824
8. Les fonctions en ligne (C99)	825
9. Les caractères étendus (C99) et Unicode (C11)	826
10. Les pointeurs restreints (C99)	826
11. La directive #pragma (C99)	827

12. Les calculs flottants (C99)	827
12.1 La norme IEEE 754	827
12.2 Choix du mode d'arrondi	828
12.3 Gestion des situations d'exception	828
12.4 Manipulation de l'ensemble de l'environnement de calcul flottant.	829
13. Structures incomplètes (C99)	829
14. Structures anonymes (C11)	829
15. Expressions fonctionnelles génériques (C11)	830
16. Gestion des contraintes d'alignement (C11)	830
17. Fonctions vérifiant le débordement mémoire (C11 facultatif)	830
18. Les threads (C11 facultatif)	831
19. Autres extensions de C99	831
20. Autres extensions de C11	831
Index	833

Avant-propos

À qui s'adresse ce livre ?

L'objectif de ce livre est d'offrir au développeur un outil de référence clair et précis sur le langage C tel qu'il est défini par la norme ANSI/ISO. Il s'adresse à un lecteur possédant déjà de bonnes notions de programmation, qu'elles aient été acquises à travers la pratique du C ou de tout autre langage. La vocation première de l'ouvrage n'est donc pas de servir de manuel d'initiation, mais plutôt de répondre aux besoins des étudiants avancés, des enseignants et des développeurs qui souhaitent approfondir leur maîtrise du langage ou trouver des réponses précises aux problèmes techniques rencontrés dans le développement d'applications professionnelles.

L'ouvrage a été conçu de façon que le lecteur puisse accéder efficacement à l'information recherchée sans avoir à procéder à une lecture linéaire. La tâche lui sera facilitée par un index très détaillé, mais aussi par la présence de nombreuses références croisées et de tableaux de synthèse servant à fois de résumé du contenu d'une section et d'aiguillage vers ses différentes parties. Il y trouvera également de nombreux encadrés décrivant la syntaxe des différentes instructions ou fonctions du langage, ainsi que des tableaux récapitulatifs et des canevas types.

Comme il se doit, nous couvrons l'intégralité du langage jusque dans ses aspects les plus marginaux ou les moins usités. Pour qu'une telle exhaustivité reste exploitable en pratique, nous l'avons largement assortie de commentaires, conseils ou jugements de valeur ; le lecteur pourra ainsi choisir en toute connaissance de cause la solution la plus adaptée à son objectif. Notamment, il sera en mesure de développer des programmes fiables et lisibles en évitant certaines situations à risque dont la connaissance reste malgré tout indispensable pour adapter d'anciens programmes. Il peut s'agir là de quelques rares cas où la norme reste elle-même ambiguë, ou encore d'usages syntaxiques conformes à la norme mais suffisamment « limites » pour être mal interprétés par certains compilateurs. Mais, plus souvent, il s'agira de possibilités désuètes, remontant à l'origine du langage, et dont la norme n'a pas osé se débarrasser, dans le souci de préserver l'existant. La plupart d'entre elles seront précisément absentes du langage C++ ; bon nombre de remarques (titrées *En C++*) visent d'ailleurs à préparer le lecteur à une éventuelle migration vers ce langage.

Une norme n'a d'intérêt que par la manière dont elle est appliquée. C'est pourquoi, au-delà de la norme elle-même, nous apportons un certain nombre d'informations pratiques. Ainsi, nous précisons le comportement qu'on peut attendre des différents compilateurs existants en cas de

non-respect de la syntaxe. Nous faisons de même pour les différentes situations d'exception qui risquent d'apparaître lors de l'exécution du programme. Il va de soi que ces connaissances se révéleront précieuses lors de la phase de mise au point d'un programme.

Malgré le caractère de référence de l'ouvrage, nous lui avons conservé une structure comparable à celle d'un cours. Il pourra ainsi être utilisé soit parallèlement à la phase d'apprentissage, soit ultérieurement, le lecteur retrouvant ses repères habituels. Accessoirement, il pourra servir de support à un cours de langage C avancé.

Toujours dans ce même souci pédagogique, nous avons doté l'ouvrage de nombreux exemples de programmes complets, accompagnés du résultat fourni par leur exécution. La plupart d'entre eux viennent illustrer une notion après qu'elle a été exposée. Mais quelques-uns jouent un rôle d'introduction pour les points que nous avons jugés les plus délicats.

Structure de l'ouvrage

La **première partie de l'ouvrage** est formée de 15 chapitres qui traitent des grandes composantes du langage suivant un déroulement classique.

Le chapitre 1 expose quelques notions de base spécifiques au C qui peuvent faire défaut au programmeur habitué à un autre langage : historique, préprocesseur, compilation séparée, différence entre variable et objet, classe d'allocation.

Le chapitre 2 présente les principaux constituants élémentaires d'un programme source : jeu de caractères, identificateurs, mots clés, séparateurs, espaces blancs, commentaires.

Le chapitre 3 est consacré aux types de base, c'est-à-dire ceux à partir desquels peuvent être construits tous les autres : entiers, flottants, caractères. Il définit également de façon précise l'importante notion d'expression constante.

Le chapitre 4 passe en revue les différents opérateurs, à l'exception de quelques opérateurs dits de référence ([], (), -> et «.») qui trouvent tout naturellement leur place dans d'autres chapitres. Il étudie la manière dont sont conçues les expressions en C, ainsi que les différentes conversions qui peuvent y apparaître : implicites, explicites par cast, forcées par affectation.

Le chapitre 5 étudie l'ensemble des instructions exécutables du langage, après en avoir proposé une classification.

Le chapitre 6 traite de l'utilisation naturelle des tableaux à une ou à plusieurs dimensions. Leur manipulation, particulière au C, par le biais de pointeurs, n'est examinée que dans le chapitre suivant. De même, le cas des tableaux transmis en argument d'une fonction n'est traité qu'au chapitre 8.

Le chapitre 7 porte sur les pointeurs : déclarations, propriétés arithmétiques, lien entre tableau et pointeur, pointeur NULL, affectation, pointeurs génériques, comparaisons, conversions. Toutefois les pointeurs sur des fonctions ne sont abordés qu'au chapitre suivant.

Le chapitre 8 est consacré aux fonctions, tant sur le plan de leur définition que des différentes façons de les déclarer. Il examine notamment le cas des tableaux transmis en argument, en distinguant les tableaux à une dimension des tableaux à plusieurs dimensions. En outre, il fait

le point sur les variables globales et les variables locales, aussi bien en ce qui concerne leur déclaration que leur portée, leur classe d'allocation ou leur initialisation.

Le chapitre 9 étudie les entrées-sorties standard, que nous avons préféré séparer des fichiers, pour des questions de clarté, malgré le lien étroit qui existe entre les deux. Ce chapitre décrit en détail les fonctions `printf`, `scanf`, `puts`, `gets`, `putchar` et `getchar`.

Le chapitre 10 montre comment le C permet de manipuler des chaînes de caractères. Il passe en revue les différentes fonctions standard correspondantes : recopie, concaténation, comparaison, recherche. Il traite également des fonctions de conversion d'une chaîne en un nombre, ainsi que des fonctions de manipulation de suites d'octets.

Le chapitre 11 examine les types définis par l'utilisateur que sont les structures, les unions et les énumérations.

Le chapitre 12 est consacré à l'instruction `typedef` qui permet de définir des types synonymes.

Le chapitre 13 fait le point sur le traitement des fichiers : aspects spécifiques au langage C, traitement des erreurs, distinction entre opérations binaires et formatées... Puis il passe en revue les différentes fonctions standard correspondantes.

Le chapitre 14 montre comment mettre en œuvre ce que l'on nomme la gestion dynamique de la mémoire et en fournit quelques exemples d'application.

Le chapitre 15 étudie les différentes directives du préprocesseur.

La **seconde partie de l'ouvrage**, plus originale dans sa thématique, est composée de 11 chapitres qui traitent de sujets transversaux comme la récursivité ou les déclarations, des modalités d'application de certains éléments de syntaxe, ou encore de possibilités peu usitées du langage.

Le chapitre 16 propose un récapitulatif sur les déclarations aussi bien sur le plan de leur syntaxe, que sur la manière de les écrire ou de les interpréter. Sa présence se justifie surtout par la complexité et l'interdépendance des règles de déclaration en C : il aurait été impossible de traiter ce thème de manière exhaustive dans chacun des chapitres concernés.

Le chapitre 17 fait le point sur la manière de pallier les problèmes de manque de fiabilité que posent les lectures au clavier.

Le chapitre 18 montre comment le langage C distingue différentes catégories de caractères (caractères de contrôle, graphiques, alphanumériques, de ponctuation...) et examine les fonctions standard correspondantes.

Le chapitre 19 fournit quelques informations indispensables dans la gestion de gros programmes nécessitant un découpage en plusieurs fichiers source.

Le chapitre 20 explique comment, à l'image de fonctions standard telles que `printf`, écrire des fonctions à arguments variables en nombre et en type.

Le chapitre 21 examine les différentes façons dont un programme peut recevoir une information de l'environnement ou lui en transmettre, ainsi que des possibilités dites de *traitement de signaux*.

Le chapitre 22 montre comment, par le biais de ce que l'on nomme les *caractères étendus*, le langage C offre un cadre de gestion d'un jeu de caractères plus riche que celui offert par le codage sur un octet.

Le chapitre 23 traite du mécanisme général de *localisation*, qui offre à une implémentation la possibilité d'adapter le comportement de quelques fonctions standard à des particularités nationales ou locales.

Le chapitre 24 illustre les possibilités de récursivité du langage.

Le chapitre 25 traite d'un mécanisme dit de *branchements non locaux*, qui permet de s'affranchir de l'enchaînement classique : appel de fonction, retour.

Le chapitre 26 recense les incompatibilités qui ont subsisté entre le C ANSI et le C++, c'est-à-dire tout ce qui fait que le C++ n'est pas tout à fait un surensemble du C.

Une importante annexe fournit la syntaxe et le rôle de l'ensemble des fonctions de la bibliothèque standard. La plupart du temps, il s'agit d'un résumé d'informations figurant déjà dans le reste de l'ouvrage, à l'exception de quelques fonctions qui, compte tenu de leur usage extrêmement restreint, se trouvent présentées là pour la première fois.

Enfin, cette nouvelle édition tient compte des deux extensions de la norme publiées en 1999 et 2011 et connues sous les acronymes C99 et C11 :

- une importante annexe en présente la plupart des fonctionnalités ; sa situation tardive dans l'ouvrage se justifie par le fait que ces « nouveautés » ne sont pas intégralement appliquées par tous les compilateurs ;
- certains ajouts sont mentionnés au fil du texte, lorsque cela nous a paru utile.

À propos des normes ANSI/ISO

On parle souvent, par habitude, du C ANSI (*American National Standard Institute*) alors que la première norme américaine, publiée en 1989, a été rendue internationale en 1990 par l'ISO (*International Standardization Organisation*), avant d'être reprise par les différents comités de normalisation continentaux ou nationaux sous la référence ISO/IEC 9899:1990. En fait, l'abus de langage se justifie par l'identité des deux documents, même si, en toute rigueur, le texte ISO est structuré différemment du texte ANSI d'origine.

Cette norme a continué d'évoluer. Certains « additifs » publiés séparément ont été intégrés dans une nouvelle norme ISO/IEC 9899:1999 (nommée brièvement C99). Une nouvelle définition est apparue avec ISO/IEC 9899:2011 (C11). La première norme reste désignée par C ANSI ou par C90.

À propos de la fonction main

En théorie, selon la norme (C90, C99 ou C11), la fonction `main` qui, contrairement aux autres fonctions, ne dispose pas de prototype, devrait disposer de l'un des deux en-têtes suivants :

```
int main (void)
int main (int arg, char *argv)
```

En fait, tant que l'on ne cherche pas à utiliser les « arguments de la ligne de commande », les deux formes :

```
int main ()  
main()
```

sont acceptées par toutes les implémentations, la seconde s'accompagnant toutefois fréquemment d'un message d'avertissement.

La première est la plus répandue et c'est celle qu'imposera la norme de C++. Nous l'utiliserons généralement.

Remerciements

Je tiens à remercier tout particulièrement Jean-Yves BROCHOT pour sa relecture extrêmement minutieuse de l'ouvrage, ainsi que pour les discussions nombreuses et enrichissantes qu'il a suscitées.

1

Généralités

A priori, cet ouvrage s'adresse à des personnes ayant déjà une expérience de la programmation, éventuellement dans un langage autre que le C. Bien qu'il puisse être étudié de manière séquentielle, il a été conçu pour permettre l'accès direct à n'importe quelle partie, à condition de disposer d'un certain nombre de notions générales, plutôt spécifiques au C, et qui sont examinées ici.

Nous commencerons par un bref historique du langage, qui permettra souvent d'éclairer certains points particuliers ou redondants. Puis nous verrons comment se présente la traduction d'un programme, à la fois par les possibilités de compilation séparée et par l'existence, originale, d'un préprocesseur. Nous expliquerons ensuite en quoi la classique notion de variable est insuffisante en C et pourquoi il est nécessaire de la compléter par celle, plus générale, d'objet. Enfin, nous verrons qu'il existe plusieurs façons de gérer l'emplacement mémoire alloué à une variable, ce qui se traduira par la notion de classe d'allocation.

1. Historique du langage C

Le langage C a été créé en 1972 par Denis Ritchie avec un objectif relativement limité : écrire un système d'exploitation (Unix). Mais ses qualités opérationnelles ont fait qu'il a très vite été adopté par une large communauté de programmeurs.

Une première définition rigoureuse du langage a été réalisée en 1978 par Kernighan et Ritchie avec la publication de l'ouvrage *The C Programming Language*. De nombreux compilateurs ont alors vu le jour en se fondant sur cette définition, quitte à l'assortir parfois de quelques extensions. Ce succès international du langage a conduit à sa normalisation, d'abord par l'ANSI (*American National Standard Institute*), puis par l'ISO (*International Standardization Organisation*), en 1993 par le CEN (Comité européen de normalisation) et enfin, en 1994, par

l'AFNOR. En fait, et fort heureusement, toutes ces normes sont identiques, et l'usage veut qu'on parle de C90 (autrefois de « C ANSI » ou de « C norme ANSI »).

La norme ANSI élargit, sans la contredire, la première définition de Kernighan et Ritchie. Pour la comprendre et pour l'accepter, il faut savoir qu'elle a cherché à concilier deux intérêts divergents :

- d'une part, améliorer et sécuriser le langage ;
- d'autre part, préserver l'existant, c'est-à-dire faire en sorte que les programmes créés avant la norme soient acceptés par la norme.

Dans ces conditions, certaines formes désuètes ou redondantes ont dû être conservées. L'exemple le plus typique réside dans les déclarations de fonctions : la première définition prévoyait de déclarer une fonction en ne fournissant que le type de son résultat ; la norme ANSI a prévu d'y ajouter le type des arguments, mais sans interdire l'usage de l'ancienne forme. Il en résulte que la maîtrise des différentes situations possibles nécessite des connaissances qui seraient devenues inutiles si la norme avait osé interdire l'ancienne possibilité. On notera, à ce propos, que la norme du C++, langage basé très fortement sur le C, supprime bon nombre de redondances pour lesquelles la norme du C n'a pas osé trancher ; c'est notamment le cas des déclarations de fonctions qui doivent obligatoirement utiliser la seconde forme.

Après cette première normalisation, des extensions ont été apportées, tout d'abord sous forme de simples additifs en 1994 (ISO/IEC 9899/COR1:1994) et en 1995 (ISO/IEC 9899/COR2 :1995), lesquels se sont trouvés intégrés dans la nouvelle norme ISO/IEC 9899:1999, désignée sous l'acronyme C99. Enfin, une dernière norme ISO/IEC 9899:2011 est apparue, plus connue sous l'acronyme C11.

Compte tenu du fait que tous les compilateurs ne respectent pas intégralement les dernières normes C99 et C11 (cette dernière comportant d'ailleurs des fonctionnalités « facultatives »), notre discours se fonde plutôt sur la norme C90. Mais, dans la suite de l'ouvrage, il nous arrivera souvent de préciser :

- ce qui constitue un apport de la norme C90 par rapport à la première définition du C ;
- ce qui dans la première définition est devenu désuet ou déconseillé, bien qu'accepté par la norme ; en particulier, nous ferons souvent référence à ce qui disparaîtra ou qui changera en C++ ;
- les points concernés par les apports des normes C99 et C11.

L'annexe en fin d'ouvrage récapitule les principaux apports de C99 et C11.

2. Programme source, module objet et programme exécutable

Tout programme écrit en langage évolué forme un texte qu'on nomme un « programme source ». En langage C, ce programme source peut être découpé en un ou plusieurs fichiers source. Notez qu'on parle de fichier même si, exceptionnellement, le texte correspondant, saisi en mémoire, n'a pas été véritablement recopié dans un fichier permanent.

Chaque fichier source est traduit en langage machine, indépendamment des autres, par une opération dite de « compilation », réalisée par un logiciel ou une partie de logiciel nommée « compilateur ». Le résultat de cette opération porte le nom de « module objet ». Bien que formé d'instructions machine, un tel module objet n'est pas exécutable tel quel car :

- il peut lui manquer d'autres modules objet ;
- il lui manque, de toute façon, les instructions exécutables des fonctions standards appelées dans le fichier source (par exemple `printf`, `scanf`, `strcat`...).

Le rôle de l'éditeur de liens est précisément de réunir les différents modules objet et les fonctions de la bibliothèque standard afin de constituer un programme exécutable. Ce n'est d'ailleurs que lors de cette édition de liens qu'on pourra s'apercevoir de l'absence d'une fonction utilisée par le programme.

3. Compilation en C : existence d'un préprocesseur

En C, la traduction d'un fichier source se déroule en deux étapes totalement indépendantes :

- un prétraitement ;
- une compilation proprement dite.

La plupart du temps, ces deux étapes sont enchaînées automatiquement, de sorte qu'on a l'impression d'avoir affaire à un seul traitement. Généralement, on parle du préprocesseur pour désigner le programme réalisant le prétraitement. En revanche, les termes de « compilateur » ou de « compilation » restent ambigus puisqu'ils désignent tantôt l'ensemble des deux étapes, tantôt la seconde.

L'étape de prétraitement correspond à une modification du texte d'un fichier source, basée essentiellement sur l'interprétation d'instructions très particulières dites « directives à destination du préprocesseur » ; ces dernières sont reconnaissables par le fait qu'elles commencent par le signe `#`.

Les deux directives les plus importantes sont :

- la directive d'inclusion d'autres fichiers source : `#include` ;
- la directive de définition de macros ou de symboles : `#define`.

La première est surtout utilisée pour incorporer le contenu de fichiers prédéfinis, dits « fichiers en-tête », indispensables à la bonne utilisation des fonctions de la bibliothèque standard, la plus connue étant :

```
#include <stdio.h>
```

La seconde est très utilisée dans les fichiers en-tête prédéfinis. Elle est également souvent exploitée par le programmeur dans des définitions de symboles telles que :

```
#define NB_COUPS_MAX 100  
#define TAILLE 25
```

4. Variable et objet

4.1 Définition d'une variable et d'un objet

Dans beaucoup de langages, les informations sont manipulées par le biais de variables, c'est-à-dire d'emplacements mémoire portant un nom et dont le contenu est susceptible d'évoluer. En C, il existe bien entendu des variables répondant à une telle définition mais on peut également manipuler des informations qui ne sont plus vraiment contenues dans des variables ; le cas le plus typique est celui d'une information manipulée par l'intermédiaire d'un pointeur :

```
int *adi ;      /* adi est une variable destinée à contenir une adresse d'entier */
...
*adi = 5 ;     /* place la valeur 5 dans l'entier pointé par adi */
```

L'entier pointé par `adi` ne porte pas vraiment de nom ; d'ailleurs, au fil de l'exécution, `adi` peut pointer sur des entiers différents.

Pour tenir compte de cette particularité, il est donc nécessaire de définir un nouveau mot. On utilise généralement celui d'objet¹. On dira donc qu'un objet est un emplacement mémoire parfaitement défini qu'on utilise pour représenter une information à laquelle on peut accéder à volonté (autant de fois qu'on le souhaite) au sein du programme.

Bien entendu, une variable constitue un cas particulier d'objet. Mais, dans notre précédent exemple, l'emplacement pointé à un instant donné par `adi` est lui-même un objet. En revanche, une expression telle `n+5` n'est pas un objet dans la mesure où l'emplacement mémoire correspondant n'est pas parfaitement défini et où, de plus, il a un caractère relativement fugitif ; on y accèdera véritablement qu'une seule fois : au moment de l'utilisation de l'expression en question.

La question de savoir si des constantes telles que `34`, `'d'` ou `"bonjour"` sont ou non des objets est relativement ambiguë : une constante utilise un emplacement mémoire mais peut-on dire qu'on y accède à volonté ? En effet, on n'est pas sûr qu'une même constante se réfère toujours au même emplacement, ce point pouvant dépendre de l'implémentation. La norme n'est d'ailleurs pas totalement explicite sur ce sujet qui n'a guère d'importance en pratique. Dans la suite, nous conviendrons qu'une constante n'est pas un objet.

4.2 Utilisation d'un objet

4.2.1 Accès par une expression désignant l'objet

Lorsqu'un objet est une variable, son utilisation ne pose guère de problème, qu'il s'agisse d'en utiliser ou d'en modifier la valeur, même si, au bout du compte, le nom même de la variable possède une signification dépendant du contexte dans lequel il est employé. Par exemple, si `p` est une variable, dans l'expression :

```
p + 5
```

1. Attention, ce terme n'a ici aucun lien avec la programmation orientée objet.

p désigne tout simplement la valeur de la variable p , tandis que dans :

$p = 5$;

p désigne la variable elle-même.

Dans le cas des objets pointés, en revanche, on ne pourra pas recourir à un simple identificateur ; il faudra faire appel à des expressions plus complexes telles que $*adi$ ou $*(adi+3)$. Ici encore, cette expression pourra intervenir soit pour utiliser la valeur de l'objet, soit pour en modifier la valeur. Par exemple, dans l'expression :

$*adi + 5$

$*adi$ désigne la valeur de l'objet pointé par adi , tandis que dans :

$*adi = 12$;

$*adi$ désigne l'objet lui-même.

4.2.2 Type d'un objet

Comme dans la plupart des langages, le type d'un objet n'est pas défini de façon intrinsèque : en examinant une suite d'octets de la mémoire, on est incapable de savoir comment l'information qui s'y trouve a été codée, et donc de donner une valeur à l'objet correspondant. En fait, le type d'un objet n'est défini que par la nature de l'expression qu'on utilise, à un instant donné, pour y accéder ou pour le modifier. Certes, dans un langage où tout objet est contenu dans une variable, cette distinction est peu importante puisque le type est alors défini par le nom de la variable, lequel constitue le seul et unique moyen d'accéder à l'objet². Dans un langage comme le C, en revanche, on peut accéder à un objet par le biais d'un pointeur. Son type se déduira, là encore, de la nature du pointeur mais avec cette différence fondamentale par rapport à la variable qu'il est alors possible, volontairement ou par erreur, d'accéder à un même objet avec des pointeurs de types différents.

Certes, lorsque l'on est amené à utiliser plusieurs expressions pour accéder à un même objet, elles sont généralement de même type, de sorte qu'on a tendance à considérer que le type de l'objet fait partie de l'objet lui-même. Par souci de simplicité, d'ailleurs, il nous arrivera souvent de parler du « type de l'objet ». Il ne s'agira cependant que d'un abus de langage qui se réfère au type ayant servi à créer l'objet, en faisant l'hypothèse que c'est celui qu'on utilisera toujours pour accéder à l'objet.

Remarque

Parmi les différentes expressions permettant d'accéder à un objet, certaines ne permettent pas sa modification. C'est par exemple le cas d'un nom d'une variable ayant reçu l'attribut `const` ou d'un nom de tableau. Comme on le verra à la section 7.2 du chapitre 4, on parle généralement de `lvalue` pour désigner les expressions utilisables pour modifier un objet.

2. Bien que certains langages autorisent, exceptionnellement, d'accéder à un même emplacement par le biais de deux variables différentes, de types éventuellement différents.

5. Lien entre objet, octets et caractères

En langage C, l'octet correspond à la plus petite partie adressable de la mémoire, mais il n'est pas nécessaire, comme pourrait le faire croire la traduction française du terme anglais *byte*, qu'il soit effectivement constitué de 8 bits, même si cela est très souvent le cas.

Tout objet est formé d'un nombre entier d'octets et par conséquent, il possède une adresse, celle de son premier octet. La réciproque n'est théoriquement pas vraie : toute adresse (d'octet) n'est pas nécessairement l'adresse d'un objet ; voici deux contre-exemples :

- l'octet en question n'est pas le premier d'un objet ;
- l'octet en question est le premier octet d'un objet mais l'expression utilisée pour y accéder correspond à un type occupant un nombre d'octets différent.

Malgré tout, il sera souvent possible de considérer une adresse quelconque comme celle d'un objet de type quelconque. Bien entendu, cela ne préjugera nullement des conséquences plus ou moins catastrophiques qui pourront en découler.

Par ailleurs, la notion de caractère en C coïncide totalement avec celle d'octet. Dans ces conditions, on pourra toujours considérer n'importe quelle adresse comme celle d'un objet de type caractère. C'est d'ailleurs ainsi que l'on procédera lorsqu'on voudra traiter individuellement chacun des octets constituant un objet.

6. Classe d'allocation des variables

Comme on le verra, notamment au chapitre 8, l'emplacement mémoire attribué à une variable peut être géré de deux façons différentes, suivant la manière dont elle a été déclarée. On parle de « classe d'allocation statique » ou de « classe d'allocation automatique ».

Les variables de classe statique voient leur emplacement alloué une fois pour toutes avant le début de l'exécution du programme ; il existe jusqu'à la fin du programme. Une variable statique est rémanente, ce qui signifie qu'elle conserve sa dernière valeur jusqu'à une nouvelle éventuelle modification.

Les variables de classe automatique voient leur emplacement alloué au moment de l'entrée dans un bloc ou dans une fonction ; il est supprimé lors de la sortie de ce bloc ou de cette fonction. Une variable automatique n'est donc pas rémanente puisqu'on n'est pas sûr d'y trouver, lors d'une nouvelle entrée dans le bloc, la valeur qu'elle possédait à la sortie précédente de ce bloc.

On verra qu'une variable déclarée à un niveau global est toujours de classe d'allocation statique, tandis qu'une variable déclarée à un niveau local (à un bloc ou à une fonction) est, par défaut, seulement de classe d'allocation automatique. On pourra agir en partie sur la classe d'allocation d'une variable en utilisant, lors de sa déclaration, un mot-clé dit « classe de mémorisation ». Par exemple, une variable locale déclarée avec l'attribut `static` sera de classe d'allocation statique. On veillera cependant à distinguer la classe d'allocation de la classe de mémorisation, malgré le lien étroit qui existe entre les deux ; d'une part la classe de mémorisation est facultative et d'autre part, quand elle est présente, elle ne correspond pas toujours à la classe d'allocation.

Par ailleurs, par le biais de pointeurs, le langage C permet d'allouer dynamiquement des emplacements pour des objets. Il est clair qu'un tel emplacement est géré d'une manière différente de celles évoquées précédemment. On parle souvent de gestion dynamique (ou programmée). La responsabilité de l'allocation et de la libération incombant, cette fois, au programmeur. Comme on peut le constater, les notions de classe statique ou automatique ne concernent donc que les objets contenus dans des variables.

Remarque

En toute rigueur, il existe une troisième classe d'allocation, à savoir la classe registre. Il ne s'agit cependant que d'un cas particulier de la classe d'allocation automatique.

2

Les éléments constitutifs d'un programme source

Un programme peut être considéré, à son niveau le plus élémentaire, comme une suite de caractères appartenant à un ensemble qu'on nomme le « jeu de caractères source ». Bien entendu, pour donner une signification au texte du programme, il est nécessaire d'effectuer des regroupements de ces caractères en vue de constituer ce que l'on pourrait nommer les « éléments constitutifs » du programme ; une telle opération est similaire à celle qui permet de déchiffrer un texte usuel en le découpant en mots et en signes de ponctuation.

Après avoir décrit le jeu de caractères source, en montrant en quoi il se distingue du jeu de caractères d'exécution, nous examinerons certains des éléments constitutifs d'un programme, à savoir : les identificateurs, les mots-clés, les séparateurs et les commentaires ; puis nous parlerons du format libre dans lequel peut être rédigé le texte d'un programme.

Nous terminerons en introduisant la notion de « token ». Peu importante en pratique, elle ne fait que formaliser la façon dont, le préprocesseur d'abord, le compilateur ensuite, effectuent le découpage du programme en ses différents éléments constitutifs.

Notez que certains des éléments constitutifs d'un programme source seront simplement cités ici, dans la mesure où ils se trouvent tout naturellement étudiés en détail dans d'autres chapitres ; cette remarque concerne les constantes numériques, les constantes chaîne ou les opérateurs.

1. Jeu de caractères source et jeu de caractères d'exécution

1.1 Généralités

On appelle « jeu de caractères source » l'ensemble des caractères utilisables pour écrire un programme source. Il est imposé par la norme ANSI ; mais cette dernière permet à des environnements possédant un jeu de caractères plus restreint de représenter certains caractères par le biais de « séquences trigraphes ».

Ce jeu doit être distingué du « jeu de caractères d'exécution » qui correspond aux caractères susceptibles d'être traités par le programme lors de son exécution, que ce soit par le biais d'informations de type `char` ou par le biais de chaînes de caractères. Bien entendu, compte tenu de l'équivalence existant en C entre octet et caractère, on voit qu'on disposera toujours de 256 caractères différents (du moins lorsque l'octet occupe 8 bits, comme c'est généralement le cas). Mais ces caractères, ainsi que leurs codes, pourront varier d'une implémentation à une autre. Cependant, la norme prévoit un jeu minimal qui doit exister dans toute implémentation. Là encore, elle ne précise que les caractères concernés, et en aucun cas leur code.

On pourrait penser que, par sa nature même, le jeu de caractères d'exécution n'a aucune incidence sur l'écriture des instructions d'un programme. En fait, il intervient indirectement lorsqu'on introduit des constantes (caractère ou chaîne) ou des commentaires à l'intérieur d'un programme. Par exemple, dans certaines implémentations françaises, vous pourrez écrire :

```
char * ch = "résultats" ;
char c_cedille = 'ç' ;
```

et ceci, bien que les caractères `é` et `ç` n'appartiennent pas au jeu de caractères source.

Le tableau 2.1 fournit la liste complète du jeu de caractères source et du jeu minimal des caractères d'exécution, tels qu'ils sont imposés par la norme. On trouvera quelques commentaires dans les deux paragraphes suivants.

Tableau 2.1 : les jeux de caractères du langage C

Types de caractères	Jeu de caractères source	Jeu minimal de caractères d'exécution
Lettres majuscules	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Lettres minuscules	a b c d e f g h i j k l m n o p q r s t u v w x y z	a b c d e f g h i j k l m n o p q r s t u v w x y z
Chiffres décimaux	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9
Espace	espace	espace
Autres caractères disposant d'un graphisme	! « % & ' () * + , - . / : ; < = > ? _ # [\] ^ { } ~	! « % & ' () * + , - . / : ; < = > ? _ # [\] ^ { } ~

Types de caractères	Jeu de caractères source	Jeu minimal de caractères d'exécution
Caractères dits de « contrôle » car non imprimables	tabulation horizontale tabulation verticale saut de page « indication » de fin de ligne	tabulation horizontale ¹ (\t) tabulation verticale (\v) saut de page (\f) alerte (\a) retour arrière (\b) retour chariot (\r) nouvelle ligne (\n)

1. Entre parenthèses, on trouvera la séquence d'échappement correspondante, telle qu'elle sera présentée à la section 2.3.2 du chapitre 3.

1.2 Commentaires à propos du jeu de caractères source

On note la présence de 29 caractères disposant d'un graphisme imprimable. Pour les environnements disposant d'un jeu de caractères restreint (tel que celui correspondant à la norme ISO 646-1083¹), la norme ANSI permet que certains des 9 derniers caractères (`# [\] ^ { | } ~`) soient remplacés par ce que l'on nomme des séquences trigraphes (suites de trois caractères commençant par `??`).

Tableau 2.2 : les séquences trigraphes

Séquence trigraphe	Caractère équivalent
??=	#
??([
??/	\
??)]
??'	^
??<	{
??!	
??>	}
??-	~

Assez curieusement, la norme semble rendre obligatoire la présence des trois caractères de contrôle que sont les tabulations et le saut de page alors que, de toute façon, ils joueront pour le compilateur le même rôle que des espaces. Le premier présente l'intérêt de faciliter les « indentations » d'instructions. On ne perdra cependant pas de vue qu'il peut poser des problèmes dès lors qu'on cherche à manipuler un même texte source à l'aide d'éditeurs différents ou à

1. Il s'agit d'un sous-ensemble du code ASCII restreint dont nous parlerons un peu plus loin.

l'aide d'un logiciel de traitement de texte : remplacement de tabulations par un certain nombre (variable !) d'espaces, pose de taquets de tabulation à des endroits différents², etc.

Par ailleurs, on notera que la norme prévoit qu'il doit exister une « manière » d'indiquer les fins de ligne, sans imposer un caractère précis. Cette tolérance vise simplement à reconnaître l'existence de certains environnements (cas des PC notamment) qui utilisent une succession de deux caractères pour représenter la fin de ligne. Mais, en théorie, la norme est très large puisqu'elle autorise n'importe quelle « technique » permettant de reconnaître convenablement les changements de ligne dans un programme source.

Remarque

Les éventuelles substitutions de séquences trigraphes ont lieu (une seule fois) avant passage au préprocesseur, quel que soit l'environnement considéré. Par exemple, l'instruction :

```
printf ("bonjour??/n" );
```

sera toujours équivalente à :

```
printf ("bonjour\n" );
```

En revanche, toute suite de caractères de la forme ??x, où x n'est pas l'un des 9 caractères prévus précédemment, ne sera pas modifiée. Par exemple :

```
printf ("bonjour??+" );
```

affichera bien : bonjour??+

1.3 Commentaires à propos du jeu minimal de caractères d'exécution

Le jeu de caractères d'exécution dépend de l'implémentation. Il s'agit donc ici des caractères qui pourront être manipulés par le programme, c'est-à-dire ceux qui pourront :

- être placés dans une variable de type caractère, par affectation ou par lecture ;
- apparaître dans une constante de type caractère ('a', '+', etc.) ;
- apparaître dans une chaîne constante ("bonjour", "salut\n", etc.).

La norme impose un jeu minimal assez réduit puisqu'il comporte, outre tous les caractères du jeu de caractères source, quelques caractères de contrôle supplémentaires :

- alerte ;
- retour arrière ;
- retour chariot ;
- nouvelle ligne.

Cette fois, le caractère de fin de ligne est bien présent en tant que tel. On notera cependant qu'il s'agit simplement de sa représentation en mémoire. Cela ne préjuge nullement de la manière dont il sera représenté dans un fichier de type texte (par un ou plusieurs caractères, voire

2. Certains environnements peuvent également poser des problèmes opposés en remplaçant automatiquement des suites d'espaces par des tabulations...

suivant une autre convention). Comme nous le verrons au chapitre 13, il existe effectivement un mécanisme permettant à un programme de « voir » les fins de ligne d'un fichier texte comme s'il s'agissait d'un caractère de fin de ligne.

Pour pouvoir faire apparaître dans une constante caractère ou une constante chaîne l'un des caractères de contrôle imposés par la norme, une notation particulière est prévue sous la forme de ce qu'on nomme souvent une « séquence d'échappement », qui utilise le caractère \. La plus connue est \n pour la fin de ligne (ou encore ??/n si l'on utilise les caractères trigraphes) ; les autres seront présentées à la section 2.3.2 du chapitre 3, mais on notera dès maintenant que la norme offre l'avantage de fournir des notations portables pour ces caractères.

Remarque

Si la norme impose le jeu de caractères minimal, elle n'impose aucunement le code utilisé pour les représenter, lequel va dépendre de l'implémentation. En pratique, on ne rencontre qu'un nombre limité de codes, parmi lesquels on peut citer l'EBCDIC et l'ASCII. On notera bien que, en toute rigueur, l'ASCII est un code qui n'exploite que 7 des 8 bits d'un octet et dans lequel les 32 premiers codes (0 à 31) sont réservés à des caractères de contrôle. Beaucoup d'autres codes (comme Latin-1) sont des surensembles de l'ASCII, les 128 premières valeurs ayant la même signification dans toutes les variantes. À noter qu'il existe un abus de langage qui consiste à nommer ASCII ces variantes à 8 bits. Parfois, on parle alors d'US-ASCII pour désigner le code initial à 7 bits.

Par ailleurs, comme nous le verrons au chapitre 22, le langage C permet de gérer des jeux de caractères beaucoup plus riches que ceux offerts par un simple octet, par le biais des « caractères étendus » et des caractères multi-octets.

2. Les identificateurs

Dans un programme, beaucoup d'éléments (variables, fonctions, types...) sont désignés par un nom qu'on appelle « identificateur ». Comme dans la plupart des langages, un tel identificateur est formé d'une suite de caractères choisis parmi les **lettres**, les **chiffres** ou le **caractère souligné** (`_`), le premier d'entre eux étant nécessairement différent d'un chiffre. Voici quelques identificateurs corrects :

```
lg_lig   valeur_5   _total   _89
```

Par ailleurs, les majuscules et les minuscules sont autorisées mais ne sont pas équivalentes (contrairement, par exemple, à ce qui se produit en Turbo Pascal). Ainsi, en C, les identificateurs « ligne » et « Ligne » désignent deux objets différents.

En ce qui concerne la longueur des identificateurs, la norme ANSI prévoit (sauf exception, mentionnée dans la remarque ci-après) que, au moins les 31 premiers caractères (63 en C11), soient « significatifs » ; autrement dit, deux identificateurs qui diffèrent par leurs 31 (63) premières lettres désigneront deux choses différentes.

En langage C, il est donc assez facile de choisir des noms de variables suffisamment évocateurs de leur rôle, voire de leur type ou de leur classe de mémorisation. En voici quelques exemples :

```
Prix_TTC           Taxe_a_la_valeur_ajoutee   TaxeValeurAjoutee  
PtrInt_Adresse
```

Remarque

En théorie, la norme (C90, aussi bien que C99 et même C11) autorise une implémentation à limiter à 6 la longueur des identificateurs dits « externes », c'est-à-dire qui subsistent après compilation : noms de fonctions, variables globales. En pratique, ces restrictions tendent à disparaître.

3. Les mots-clés

Certains mots sont réservés par le langage à un usage bien défini. On les nomme des « mots-clés ». **Un mot-clé ne peut pas être employé comme identificateur.** Le tableau 2.3 liste les mots-clés par ordre alphabétique.

Tableau 2.3 : les mots-clés du langage C

auto	default	float	register	struct	volatile
break	do	for	return	switch	while
case	double	goto	short	typedef	
char	else	if	signed	union	
const	enum	int	sizeof	unsigned	
continue	extern	long	static	void	

Remarque

Contrairement à ce qui se passe dans un langage comme le Pascal, en C il n'y a théoriquement pas de notion d'identificateur prédéfini dont la signification pourrait, le cas échéant, être modifiée par l'utilisateur. Cependant, un tel résultat peut quand même être obtenu en faisant appel à la directive `#define` du préprocesseur, comme on le verra au chapitre 15.

4. Les séparateurs et les espaces blancs

Dans notre langue écrite, les mots sont séparés par un espace, un signe de ponctuation ou une fin de ligne. Il en va presque de même pour le langage C dont les règles vont donc paraître naturelles. Ainsi, dans un programme, deux identificateurs successifs entre lesquels la syntaxe n'impose aucun caractère ou groupe de caractères particulier dit « séparateur », doivent impérativement être séparés par ce que l'on nomme un « espace blanc » (le plus usité étant l'espace) :

On nomme espace blanc une suite de un ou plusieurs caractères choisis parmi : espace, tabulation horizontale, tabulation verticale, changement de ligne ou changement de page.

En revanche, dès que la syntaxe impose un séparateur quelconque, il n'est pas nécessaire de prévoir d'espaces blancs supplémentaires. Cependant, en pratique, des espaces amélioreront généralement la lisibilité du programme.

Les caractères séparateurs comprennent tous les opérateurs (+, -, *, =, +=, ==...) ainsi que les caractères dits de « ponctuation »³ :

```
( ) [ ] { } , ; : ...
```

Par exemple, vous devrez impérativement écrire (avec au moins un espace blanc entre `int` et `x`) :

```
int x,y ;
```

et non :

```
intx,y ;
```

En revanche, vous pourrez écrire indifféremment (le caractère virgule étant un séparateur) :

```
int n,compte,total,p ;
```

ou plus lisiblement :

```
int n, compte, total, p ;
```

5. Le format libre

Comme le langage Pascal, le langage C autorise une « mise en page » parfaitement libre. En particulier, une instruction peut s'étendre sur un nombre quelconque de lignes et une même ligne peut comporter autant d'instructions que voulu.

Les fins de ligne ne jouent pas de rôle particulier, si ce n'est celui de séparateur, au même titre qu'un simple espace, ce qui signifie donc qu'un identificateur ne peut être « coupé en deux par une fin de ligne. Par exemple :

```
int quant
ite ;          /* incorrect */
```

est incorrect car équivalent à :

```
int quant ite ;
```

et non à :

```
int quantite ;
```

ce qui correspond à l'intuition.

En revanche, les fins de ligne ne sont pas autorisées dans les constantes chaîne. Par exemple :

```
const char message = "salut, cher ami,
comment allez-vous?" ;
```

conduira à une erreur de compilation.

3. Certains de ces caractères comme [et] apparaissent dans un opérateur ([]), lequel est lui-même un séparateur. Néanmoins, chacun d'entre eux forme également un séparateur : on peut trouver « quelque chose » entre [et].

Il ne faut pas en conclure qu'il est nécessaire d'écrire une constante chaîne sur une seule ligne car on pourra faire appel aux possibilités dites de « concaténation des chaînes adjacentes », décrites à la section 1.2 du chapitre 10, en écrivant, par exemple :

```
const char message = "salut, cher ami, "
                    "comment allez-vous?" ;
```

Remarques

1. Les directives à destination du préprocesseur (comme `#include` ou `#define`) ne bénéficient pas du format libre dont nous venons de parler. De plus, la fin de ligne `y` joue un rôle important de terminateur.
2. Le fait que le langage C autorise un format libre présente des contreparties. Notamment, le risque existe, si l'on n'y prend pas garde, d'aboutir à des programmes peu lisibles. Voici, à titre d'illustration, un programme accepté par le compilateur :

Exemple de programme mal présenté

```
int main() { int i,n;printf("bonjour\n") ; printf(
"je vais vous calculer 3 carrés\n") ; for (i=1;i<=
3; i++){printf("donnez un nombre entier : ") ; scanf("%d",
&n) ; printf ("son carré est: %d\n", n*n) ; } printf (
"au revoir"); }
```

6. Les commentaires

Comme tout langage évolué, le langage C autorise la présence de commentaires dans les programmes source. Il s'agit de textes explicatifs destinés aux lecteurs du programme et qui n'ont aucune incidence sur sa compilation.

Ces commentaires sont formés de caractères quelconques placés entre les symboles `/*` et `*/`. Ils peuvent apparaître à tout endroit du programme où un espace est autorisé, y compris dans les directives à destination du préprocesseur. En général, cependant, on se limitera à des emplacements propices à une bonne lisibilité du programme.

Voici quelques exemples de commentaires :

```
/* programme de calcul de racines carrées */

/* commentaire fantaisiste &ç${<>} ?%!!!!!! */

/* commentaire s'étendant
sur plusieurs lignes
de programme source */

/* =====
*   commentaire quelque peu esthétique   *
*   et mis en boîte, pouvant servir,     *
*   par exemple, d'en-tête de programme  *
===== */
```

Voici d'autres exemples de commentaires qui, situés au sein d'une instruction de déclaration, permettent de définir le rôle des différentes variables :

```
int nb_points ; /* nombre de valeurs à calculer */
float debut, /* abscisse de début */
    fin, /* abscisse de fin */
    pas ; /* pas choisi */
```

La norme ANSI ne prévoit pas de possibilités dites de « commentaires imbriqués ». Ainsi, une construction telle que :

```
/* début commentaire 1 /* commentaire 2 */ fin commentaire 1 */
```

conduit à considérer la dernière partie (fin commentaire 1) comme des instructions C, et donc à une erreur de syntaxe⁴.

Voici, enfin, un exemple de commentaire déraisonnable mais accepté par le compilateur :

```
int /*type entier*/ n /*une variable*/,/*---*/p/*une autre variable*/ ;

n /*on affecte a n*/= /*la valeur*/p/*de p*/ + /*augmentée de*/2 /* deux */ ;
```

Remarque

La norme C99 autorise une seconde forme de commentaire, dit « de fin de ligne » que l'on retrouve également en C++. Un tel commentaire est introduit par // et tout ce qui suit ces deux caractères jusqu'à la fin de ligne est considéré comme un commentaire. En voici un exemple :

```
printf("bonjour\n") ; // formule de politesse
```

7. Notion de token

Ce chapitre a étudié certains éléments constituant un programme. Les autres, tels que les constantes, les chaînes constantes, les opérateurs, seront abordés dans d'autres chapitres. D'une manière générale, on peut dire qu'un programme source peut se décomposer en une succession d'éléments que l'on nomme « tokens »⁵. Cette notion est bien entendu indispensable au compilateur chargé de cette décomposition. En revanche, elle l'est beaucoup moins pour l'auteur ou le lecteur d'un programme, excepté dans de très rares cas. Cette section présente donc surtout un intérêt anecdotique, hormis pour ceux qui souhaiteraient réaliser un compilateur ou, éventuellement, être en mesure d'interpréter des constructions un peu sophistiquées.

7.1 Les différentes catégories de tokens

Si l'on souhaite être exhaustif, il faut distinguer les tokens reconnus par le préprocesseur de ceux qui seront reconnus par le compilateur proprement dit, même si, comme on s'y attend, la

4. Certains compilateurs acceptent les commentaires imbriqués.

5. Nous n'avons pas cherché à traduire le mot anglais.

plupart d'entre eux sont communs. Le tableau 2.4 liste les différentes catégories de tokens existant dans les deux cas, ainsi que le chapitre de l'ouvrage dans lequel ils sont décrits (certains l'étant dans le présent chapitre).

Tableau 2.4 : les différents tokens du langage C

Catégorie de token	Préprocesseur/ Compilateur	Remarques	Étudié au
Identificateur	oui/oui		Chapitre 2, section 2
Mot-clé	non/oui	Simple identificateur pour le préprocesseur	Chapitre 2, section 3
Constante numérique (caractère, entière, flottante)	oui/oui		Chapitre 3
Chaîne constante (libellé)	oui/oui		Chapitre 10
Opérateur : un ou deux caractères consécutifs, par exemple +, -, <=	oui/oui	Certains symboles (=, ...) peuvent être opérateur ou ponctuation suivant le contexte	Chapitre 4
Caractère de ponctuation : [] () { } * , : = ; ... #	oui/oui	– certains symboles doivent être appariés pour le compilateur – # n'est ponctuation que pour le préprocesseur	
Nom de fichier en-tête	oui/non		Chapitre 15

Comme on peut le constater, les différences de tokens entre le préprocesseur et le compilateur sont minimes et logiques :

- les mots-clés ne sont pas reconnus par le préprocesseur qui n'y voit qu'un identificateur ordinaire ;
- les symboles [] () { } sont encore de simples ponctuations pour le préprocesseur, tandis qu'ils posséderont une signification particulière pour le compilateur, ce qui leur impose alors d'être appariés correctement.

Remarques

1. La norme ne considère pas un commentaire comme un token, dans la mesure où il est assimilé aux caractères dits « espaces blancs », utilisés précisément pour séparer deux tokens. D'ailleurs, ces commentaires ne sont vus que du préprocesseur et non du compilateur. En pratique, ce point est de faible importance.
2. Les opérateurs forment une catégorie de tokens, les caractères de ponctuation en forment une autre. Ce point est de moindre importance. En effet, on pourrait considérer qu'un séparateur n'est plus un token puisqu'il sert à séparer des tokens. En fait, la norme a dû apporter cette précision, tout simplement pour que les tokens identifiés par le préprocesseur soient effectivement retransmis au compilateur et non supprimés comme le sont, par exemple, les espaces blancs ou les commentaires.

7.2 Décomposition en tokens

A priori, la démarche adoptée par le préprocesseur ou le compilateur est assez intuitive. Cependant, la connaissance de l'algorithme utilisé peut s'avérer nécessaire pour interpréter correctement certains cas limites. Ce dernier se résume ainsi :

Le préprocesseur et le compilateur recherchent toujours la plus longue séquence possible de caractères qui soit un token.

Voici quelques exemples :

```
x1 + 2 /* conduit aux tokens : x1, + et 2 */
x++2 /* conduit aux tokens : x, ++ et 2, comme si on avait écrit : x++ 2 */
/* et non, par exemple, aux tokens : x, +, + et 2 */
x+++3 /* conduit aux tokens : x, ++, + et 3, comme si on avait écrit : x ++ +3 */
x++++3 /* conduit aux tokens : x, ++, ++ et 3, comme si on avait écrit :
/* x ++ ++ 3, ou encore x++ ++3, ce qui sera rejeté par le compilateur */
```

D'une manière générale, il est possible d'éviter au lecteur d'avoir à s'interroger sur l'algorithme utilisé en adoptant un style de programmation évitant toute ambiguïté. Le troisième exemple pourrait notamment être écrit ainsi (le deuxième et le quatrième exemple étant, de toute façon, incorrects) :

```
x++ + 3 /* plus lisible que : x+++3 */
```


3

Les types de base

La manipulation d'une information fait généralement intervenir la notion de type, c'est-à-dire la manière dont elle a été codée en mémoire. La connaissance de ce type est nécessaire pour la plupart des opérations qu'on souhaite lui faire subir. Traditionnellement, on distingue les types simples dans lesquels une information est, à un instant donné, caractérisée par une seule valeur, et les types agrégés dans lesquels une information est caractérisée par un ensemble de valeurs.

Ce chapitre étudie tout d'abord les caractéristiques des différents types de base du langage, c'est-à-dire ceux à partir desquels peuvent être construits tous les autres, qu'il s'agisse de types simples comme les pointeurs ou de types agrégés comme les tableaux, les structures ou les unions. Nous les avons classés en trois catégories : entiers, caractères et flottants, même si, comme on le verra, les caractères apparaissent, dans une certaine mesure, comme des cas particuliers d'entiers. Nous présenterons ensuite l'importante notion d'expression constante, c'est-à-dire calculable par le compilateur. Enfin, nous terminerons sur la déclaration et l'initialisation des variables d'un type de base.

1. Les types entiers

Pour les différents types entiers prévus par la norme, nous étudions ici les différentes façons de les nommer et la manière dont les informations correspondantes sont codées en mémoire. Nous fournissons quelques éléments permettant de choisir le type entier le plus approprié à un objectif donné. Enfin, nous indiquons les différentes façons d'écrire des constantes entières dans un programme source.

1.1 Les six types entiers

En théorie, la norme ANSI prévoit qu'il puisse exister six types entiers différents caractérisés par deux paramètres :

- la taille de l'emplacement mémoire utilisé pour les représenter ;
- un attribut précisant si l'on représente des nombres signés, c'est-à-dire des entiers relatifs, ou des nombres non signés, c'est-à-dire des entiers naturels.

Le premier paramètre est assez classique et, comme dans la plupart des langages, il se traduit par l'existence de différents noms de type : à chaque nom correspond une taille qui pourra cependant dépendre de l'implémentation. Le second paramètre, quant à lui, est beaucoup moins classique et on verra qu'il est conseillé de ne recourir aux entiers non signés que dans des circonstances particulières telles que la manipulation de motifs binaires.

Pour chacun des six types, il existe plusieurs façons de les nommer, compte tenu de ce que :

- le paramètre de taille s'exprime par un attribut facultatif : `short` ou `long` ;
- le paramètre de signe s'exprime, lui aussi, par un attribut facultatif : `signed` ou `unsigned` ;
- le mot-clé `int`, correspondant à un entier de taille intermédiaire, peut être omis, dès lors qu'au moins un des qualificatifs précédents est présent.

Le tableau 3.1 récapitule les différentes manières de spécifier chacun des six types (on parle de « spécificateur de type »), ainsi que la taille minimale et le domaine minimal que leur impose la norme, quelle que soit l'implémentation concernée.

Tableau 3.1 : les six types entiers prévus par la norme

Dénomination usuelle	Spécificateurs de type possibles	Taille minimale (en octets)	Domaine minimal
Entier court signé	<code>short</code> <code>short int</code> <code>signed short</code> <code>signed short int</code>	2	-32 767 à +32 767
Entier court non signé	<code>unsigned short</code> <code>unsigned short int</code>	2	0 à 65 535
Entier signé	<code>int</code> <code>signed int</code> <code>signed</code>	2	-32 767 à +32 767
Entier non signé	<code>unsigned int</code> <code>unsigned</code>	2	0 à 65 535
Entier long signé	<code>long</code> <code>long int</code> <code>signed long</code> <code>signed long int</code>	4	-2 147 483 647 à +2 147 483 647
Entier long non signé	<code>unsigned long</code> <code>unsigned long int</code>	4	0 à 4 294 967 295

Remarques

1. La norme impose qu'un type signé et un type non signé de même nom possèdent la même taille ; ce sera par exemple le cas pour `long int` et `unsigned long int`.
2. La norme impose seulement à une implémentation de disposer de ces six types ; en particulier, rien n'interdit que deux types différents possèdent la même taille. Fréquemment, d'ailleurs, soit les entiers courts et les entiers seront de même taille, soit ce seront les entiers et les entiers longs. Les seules choses dont on soit sûr sont que, dans une implémentation donnée, la taille des entiers courts est inférieure ou égale à celle des entiers et que celle des entiers est inférieure ou égale à celle des entiers longs.
3. C99 introduit les types supplémentaires : `long long int` et `unsigned long long int`.

1.2 Représentation mémoire des entiers et limitations

Comme le montre le tableau 3.2, la norme prévoit totalement la manière dont une implémentation doit représenter les entiers non signés ainsi que les valeurs positives des entiers signés. En revanche, elle laisse une toute petite latitude en ce qui concerne la représentation des valeurs négatives.

Tableau 3.2 : contraintes imposées à la représentation des entiers

Nature de l'entier	Contrainte imposée par la norme	Remarque
Non signé	Codage binaire pur	Un entier non signé de taille donnée est donc totalement portable.
Signé	Un entier positif doit avoir la même représentation que l'entier non signé de même valeur.	La plupart des implémentations utilisent, pour les nombres négatifs, la représentation dite du « complément à deux ».

Voyons cela plus en détail en raisonnant, pour fixer les idées, sur des nombres entiers représentés sur 16 bits. Il va de soi qu'il serait facile de généraliser notre propos à une taille quelconque.

1.2.1 Représentation des nombres non signés

La norme leur impose une notation binaire pure, de sorte que le codage de ces nombres est totalement défini : il s'agit simplement de la représentation du nombre en base 2. Voici des exemples de représentation de quelques nombres sur 16 bits : la dernière colonne reprend, sous forme hexadécimale classique, le codage binaire exprimé dans la colonne précédente :

Valeur décimale	Codage en binaire	Codage, exprimé en hexadécimal
1	0000000000000001	0001
2	0000000000000010	0002
3	0000000000000011	0003
16	0000000000010000	0010
127	0000000001111111	007F
255	0000000011111111	00FF
1 025	0000010000000001	0401
32 767	0111111111111111	7FFF
32 768	1000000000000000	8000
32 769	1000000000000001	8001
64 512	1111110000000000	FC00
65 534	1111111111111110	FFFE
65 535	1111111111111111	FFFF

1.2.2 Représentation des nombres entiers signés

Lorsqu'il s'agit d'un nombre positif, la norme impose que sa représentation soit identique à celle du même nombre sous forme non signée, ce qui revient à dire qu'on y trouve le nombre exprimé en base 2. Dans ces conditions, comme l'implémentation doit pouvoir distinguer entre nombres positifs et nombres négatifs, la seule possibilité qui lui reste consiste à se baser sur le premier bit, en considérant que 0 correspond à un nombre positif, tandis que 1 correspond à un nombre négatif¹.

Une latitude subsiste néanmoins dans la manière de coder la valeur absolue du nombre. Actuellement, la représentation dite « en complément à deux » tend à devenir universelle. Elle procède ainsi :

- on exprime la valeur en question en base 2 ;
- tous les bits sont « inversés » : 1 devient 0 et 0 devient 1 ;
- enfin, on ajoute une unité au résultat.

Voici des exemples de représentation de quelques nombres négatifs sur 16 bits, lorsqu'on utilise cette technique : la dernière colonne reprend, sous forme hexadécimale classique, le codage binaire exprimé dans la colonne précédente.

Valeur décimale	Codage en binaire	Codage, exprimé en hexadécimal
-1	1111111111111111	FFFF
-2	1111111111111110	FFFE
-3	1111111111111101	FFFD
-4	1111111111111100	FFFC
-16	1111111111110000	FFFO
-256	1111111000000000	FF00
-1 024	1111110000000000	FC00
-32 768	1000000000000000	8000

1. Même si la norme n'impose pas formellement l'existence d'un bit de signe.

Remarques

1. Dans la représentation en complément à deux, le nombre 0 est codé d'une seule manière, à savoir 0000000000000000.
2. Si l'on ajoute 1 au plus grand nombre positif (ici 0111111111111111, soit 7FFF en hexadécimal ou 32 767 en décimal) et que l'on ne tient pas compte du dépassement de capacité qui se produit, on obtient... le plus petit nombre négatif possible (ici 1000000000000000, soit 8 000 en hexadécimal ou -32 768 en décimal). C'est ce qui explique le phénomène de « modulo » bien connu de l'arithmétique entière dans le cas (fréquent) où les dépassements de capacité ne sont pas détectés.
3. Par sa nature, la représentation en complément à deux conduit à une différence d'une unité entre la valeur absolue du plus petit négatif et celle du plus grand positif. Dans les autres représentations (rares), on a souvent deux façons de coder le nombre 0 : avec bit de signe à 0 ou avec bit de signe à 1. Dans ce cas, on dispose d'autant de combinaisons possibles pour les positifs que pour les négatifs, la valeur absolue de la plus petite valeur négative étant alors égale à celle de la plus grande valeur positive qui se trouve être la même que dans la représentation en complément à deux. Autrement dit, la manière exacte dont on représente les nombres négatifs a une incidence extrêmement faible sur le domaine couvert par un nombre de bits donnés puisqu'elle ne joue que sur une unité.

1.2.3 Limitations

Le nombre de bits utilisés pour représenter un entier et le codage employé dépendent, bien sûr, de l'implémentation considérée. Il en va donc de même des limitations qui en découlent. Le tableau 3.3 indique ce que sont ces limitations relatives aux entiers dans le cas où ils sont codés sur 16 ou 32 bits, en utilisant la représentation en complément à deux.

Tableau 3.3 : limitations relatives aux entiers codés en complément à deux

Attribut	Taille (bits)	Mini		Maxi
Non signé	16	0	à	65 535
Signé	16	-32 768	à	32 767
Non signé	32	0	à	4 294 967 295
Signé	32	-2 147 483 648	à	2 147 483 647

Dans les implémentations respectant la norme ANSI, sans utiliser la représentation en complément à deux, la seule différence pouvant apparaître dans ces limitations concerne uniquement la valeur absolue des valeurs négatives. Ces dernières, comme expliqué dans la troisième remarque, peuvent se trouver diminuées de 1 unité. D'ailleurs, la norme tient compte de cette remarque pour imposer le domaine minimal des différents types entiers, comme nous l'avons présenté dans le tableau 3.1.

D'une manière générale, les limites spécifiques à une implémentation donnée peuvent être connues en utilisant le fichier en-tête `limits.h` qui sera décrit à la section 3 de ce chapitre. Ce fichier contient également d'autres informations concernant les caractéristiques des entiers et des caractères.

1.3 Critères de choix d'un type entier

Compte tenu du grand nombre de types entiers différents dont on dispose, voici quelques indications permettant d'effectuer son choix.

Utiliser les entiers non signés uniquement lorsque cela est indispensable

À taille égale, un type entier non signé permet de représenter des nombres deux fois plus grands (environ) qu'un type signé. Dans ces conditions, certains programmeurs sont tentés de recourir aux types non signés pour profiter de ce gain. En fait, il faut être prudent pour au moins deux raisons :

- dès qu'on est amené à effectuer des calculs, il est généralement difficile d'affirmer qu'on ne sera pas conduit, à un moment ou à un autre, à un résultat négatif non représentable dans un type non signé ;
- même s'il est permis, le mélange de types signés et non signés dans une même expression est fortement déconseillé, compte tenu du peu de signification que possèdent les conversions qui se trouvent alors mises en place (elles sont décrites à la section 3 du chapitre 4).

En définitive, les entiers non signés ne devraient pratiquement jamais être utilisés pour réaliser des calculs. On peut considérer que leur principale vocation est la manipulation de motifs binaires, indépendamment des valeurs numériques correspondantes. On les utilisera souvent comme opérands des opérateurs de manipulation de bits ou comme membres d'unions ou de champs de bits. Dans certains cas, ils pourront également être utilisés pour échanger des informations numériques entre différentes machines, compte tenu du caractère entièrement portable de leur représentation. Toutefois, il sera alors généralement nécessaire de transformer artificiellement des valeurs négatives en valeurs positives (par exemple, en introduisant un décalage donné).

Remarque

Le mélange entre flottants et entiers non signés présente les mêmes risques que le mélange entre entiers non signés et entiers signés. En revanche, le mélange entre entiers signés et flottants ne pose pas de problèmes particuliers. Cela montre que l'arithmétique non signée constitue un cas bien à part.

Efficacité

En général, le type `int` correspond au type standard de la machine, de sorte que l'on est quasiment assuré que c'est dans ce type que les opérations seront les plus rapides. On pourra l'utiliser pour réaliser des programmes portables efficaces, pour peu qu'on accepte les limitations correspondantes.

Signalons que l'on rencontre actuellement des machines à 64 bits, dans lesquelles la taille du type `int` reste limitée à 32 bits, probablement par souci de compatibilité avec des machines antérieures. Le type `int` reste cependant le plus efficace car, généralement, des mécanismes spécifiques à la machine évitent alors la dégradation des performances.

Occupation mémoire

Le type `short` est naturellement celui qui occupera le moins de place, sauf si l'on peut se contenter du type `char`, qui peut jouer le rôle d'un petit entier (voir section 2 de ce chapitre). Toutefois, l'existence de contraintes d'alignement et le fait que ce type peut être plus petit que la taille des entiers manipulés naturellement par la machine, peuvent conduire à un résultat opposé à celui escompté. Par exemple, sur une machine où le type `short` occupe deux octets et où le type `int` occupe 4 octets, on peut très bien aboutir à la situation suivante :

- les informations de 2 octets sont alignées sur des adresses multiples de 4, ce qui peut annuler le gain de place escompté ;
- l'accès à 2 octets peut impliquer l'accès à 4 octets avec sélection des 2 octets utiles d'où une perte de temps.

Dans tous les cas, dans l'évaluation d'une expression, toute valeur de type `short` est convertie systématiquement en `int` (voir éventuellement le chapitre 4), ce qui peut entraîner une perte de temps.

En pratique, le type `short` pourra être utilisé pour des tableaux car la norme impose la contiguïté de leurs éléments : on sera donc assuré d'un gain de place au détriment éventuel d'une perte de temps.

Remarque

Tout ce qui vient d'être dit à propos du type `short` se transposera au petit type entier qu'est le type `char`.

Portabilité des programmes

Ici, le problème est délicat dans la mesure où le terme même de portabilité est quelque peu ambigu. En effet, s'il s'agit d'écrire un programme qui compile correctement dans toute implémentation, on peut effectivement utiliser n'importe quel type. En revanche, s'il s'agit d'écrire un programme qui fonctionne de la même manière dans toute implémentation, il n'en va plus de même étant donné qu'un spécificateur de type donné correspond à un domaine différent d'une implémentation à une autre. Par exemple, `int` pourra correspondre à 2 octets sur certaines machines, à 4 octets sur d'autres... Dans certains cas, on souhaitera disposer d'un type entier ayant une taille bien déterminée ; on pourra y parvenir en utilisant des possibilités de compilation conditionnelle (voir section 3.4.2 du chapitre 15).

1.4 Écriture des constantes entières

Lorsque vous devez introduire une constante entière dans un programme, le langage C vous laisse le choix entre trois formes d'écriture présentées dans le tableau 3.4 :

Tableau 3.4 : les trois formes d'écriture des constantes entières

Forme d'écriture	Définition	Exemples	Remarques
Décimale	Correspond à notre notation usuelle d'un nombre entier, avec ou sans signe.	+533 48 -273	Conseillée dans les cas usuels (arithmétique classique)
Octale	Nombre écrit en base 8, précédé du chiffre 0.	014 équivaut à 12 037 équivaut à 31	Peu portable, conseillée pour imposer un motif binaire (voir section suivante)
Hexadécimale	Nombre écrit en hexadécimal (base 16, les chiffres supérieurs à 9 sont représentés par les lettres A à F majuscules ou minuscules), précédées des deux caractères 0x (ou 0X).	0x1A (16+10) équivaut à 26 0x3F (3*16+15) équivaut à 63	Peu portable, conseillée pour imposer un motif binaire (voir section suivante)

1.5 Le type attribué par le compilateur aux constantes entières

Tant que l'on se limite à l'écriture de constantes sous forme décimale, au sein d'expressions ne faisant pas intervenir d'entiers non signés, on peut tranquillement ignorer la nature exacte du type que leur attribue le compilateur. Mais cette connaissance s'avère indispensable dans les situations suivantes :

- utilisation de constantes écrites en notation décimale dans une expression où apparaissent des quantités non signées ;
- utilisation de constantes écrites en notation hexadécimale ou octale.

Nous allons ici examiner les règles employées par le compilateur pour définir ce type.

1.5.1 Cas usuel de la notation décimale

Une constante entière écrite sous forme décimale est du premier des types suivants dont la taille suffit à la représenter correctement : `int`, `long int`, `unsigned long int`.

Ainsi, dans toute implémentation, la constante 12 sera de type `int` ; la constante 3 000 000 sera représentée en `long` si le type `int` est de capacité insuffisante et dans le type `int` dans le cas contraire. Il faut cependant noter que les valeurs positives non représentables dans le type `long` seront représentées dans le type `unsigned long` si ce dernier a une capacité suffisante. Cela va à l'encontre du conseil prodigué à la section 1.3 de ce chapitre, puisqu'on risque de se retrouver sans le vouloir en présence d'une expression mixte. Toutefois, cette anomalie ne se produit qu'avec des constantes qui, de toute façon, ne sont pas représentables dans l'implémentation.

1.5.2 Cas de la notation octale ou hexadécimale

Une constante entière écrite sous forme octale ou hexadécimale est du premier des types suivants dont la taille suffit à la représenter correctement : `int`, `unsigned int`, `long int`, `unsigned long int`.

Alors qu'une constante décimale n'est jamais non signée (excepté lorsqu'elle n'est pas représentable en `long`), un constante octale ou hexadécimale peut l'être, alors même qu'elle aurait été représentable en `long`. En outre, elle peut se voir représenter avec un attribut de signe différent suivant l'implémentation. Par exemple :

- `0xFF` sera toujours considérée comme un `int` (donc signée) car, dans toutes les implémentations, la capacité du type `int` est supérieure à 255.
- `0xFFFF` sera considérée comme un `unsigned int` dans les implémentations où le type `int` utilise 16 bits et comme un `int` dans celles où le type `int` utilise une taille supérieure.

Cette remarque trouvera sa pleine justification avec les règles utilisées dans l'évaluation d'expressions mixtes (mélangeant des entiers signés et non signés) puisque, comme l'explique le chapitre 4, ces dernières ont alors tendance à privilégier la conservation du motif binaire plutôt que la valeur. Quoi qu'il en soit, on ne perdra pas de vue que l'utilisation de constantes hexadécimales ou octales nécessite souvent la connaissance de la taille exacte du type concerné dans l'implémentation employée et qu'elle est donc, par essence, peu portable.

1.6 Exemple d'utilisation déraisonnable de constantes hexadécimales

La section 6 du chapitre 4, vous présentera des exemples d'utilisation raisonnable de constantes hexadécimales, notamment pour réaliser des « masques binaires ». Ici, nous nous contentons d'un programme qui illustre les risques que présente un usage non justifié de telles constantes. Il a été exécuté dans une implémentation où le type `int` est représenté sur 16 bits, les nombres négatifs utilisant la représentation en complément à deux :

Utilisation déraisonnable de constantes hexadécimales

```
int main()
{ int n ;
  n = 10 + 0xFF ;                               premiere valeur : 265
  printf ("premiere valeur : %d\n", n) ;         seconde valeur  : 9
  n = 10 + 0xFFFF ;
  printf ("seconde valeur  : %d\n", n) ;
}
```

À première vue, tout se passe comme si `0xFF` était interprétée comme valant 255, tandis que `0xFFFF` serait interprétée comme valant -1, ce qui correspond effectivement à la représentation sur 16 bits de la constante -1 dans le type `int`. Or, comme indiqué précédemment à la section 1.5.2, la norme prévoit que `0xFFFF` soit de type `unsigned int`, ce qui correspond à la valeur 65 536. En fait, comme on le verra au chapitre suivant, le calcul de l'expression `10 + 0xFFFF` se fait en convertissant 10 en `unsigned int`. Dans ces conditions, le résultat (65 545) dépasse la capacité de ce type ; mais la norme prévoit exactement le résultat (par une formule de modulo), à savoir 65 545 modulo 65 536, c'est-à-dire 9. La conversion en `int` qu'entraîne son affectation à `n` ne pose ensuite aucun problème.

Ainsi, dans cette implémentation, tout se passe effectivement comme si les constantes hexadécimales étaient signées : avec un type `int` représenté sur 16 bits, `0xFF` apparaît comme positive,

tandis que `xFFFF` apparaît comme négative. En revanche, avec un type `int` représenté sur 32 bits, `0xFFFF` apparaîtrait comme positive, alors que `0xFFFFFFFF` apparaîtrait comme négative.

1.7 Pour imposer un type aux constantes entières

Il est toujours possible, quelle que soit l'écriture employée (décimale, octale, hexadécimale), de forcer le compilateur :

- à utiliser un type `long` en ajoutant la lettre `L` (ou `l`) à la suite de l'écriture de la constante. Par exemple :

```
1L    045L    0x7F8L    25l    0xFFl
```

Bien entendu, la constante correspondante sera de type `signed long` ou `unsigned long` suivant les règles habituelles présentées précédemment ;

- à utiliser un attribut `unsigned` en ajoutant la lettre `U` (ou `u`) à la suite de l'écriture de la constante. Par exemple :

```
1U    035U    0xFFFFu
```

Là encore, la constante correspondante sera de type `unsigned int` ou `unsigned long` suivant les règles présentées précédemment ;

- à combiner les deux possibilités précédentes. Par exemple :

```
1LU    045LU    0xFFFFlu
```

Cette fois, la constante correspondante est obligatoirement du type `unsigned long`.

Remarque

À simple titre indicatif, sachez que le programme précédent (utilisation de constantes hexadécimales), exécuté dans la même implémentation, fournit toujours les mêmes résultats, quelle que soit la façon d'écrire la constante utilisée dans la sixième ligne, à savoir : `0xFFFF`, `0xFFFFu`, `0xFFFFl` ou `0xFFFFlu`.

1.8 En cas de dépassement de capacité dans l'écriture des constantes entières

Comme le compilateur choisit d'office le type approprié, les seuls cas qui posent vraiment problème sont ceux où vous écrivez une constante positive supérieure à la capacité du type `unsigned long int` ou une constante négative inférieure au plus petit nombre représentable dans le type `long int`.

On notera que, dans ce cas, la norme du langage C, comme celle des autres langages, ne prévoit nullement comment doit réagir le compilateur². Certains compilateurs fournissent alors un diagnostic de compilation, ce qui est fort satisfaisant. D'autres se contentent de fabriquer une constante fantaisiste (généralement par perte des bits les plus significatifs !). Ainsi, sur une

2. La norme ne dit jamais comment doit réagir le compilateur ou le programme en cas de situation d'exception, c'est-à-dire de non-respect de la norme.

machine où le type `longint` occupe 32 bits, une constante telle que `4 294 967 297` pourra être acceptée à la compilation et interprétée comme valant 2 !

Rappelons qu'il est déconseillé d'utiliser des constantes décimales dont la valeur est supérieure à la capacité du type `long`, tout en restant inférieure à la capacité du type `unsigned long` car on serait alors amené à créer une valeur de type non signé, sans nécessairement s'en apercevoir ; si cette constante apparaît dans une expression utilisant des types signés, le résultat peut être différent de celui escompté.

2. Les types caractère

Le langage C dispose non pas d'un seul, mais de deux types caractère, l'un signé, l'autre non signé. Cette curiosité est essentiellement liée à la forte connotation numérique de ces deux types. Ici, nous examinerons les différentes façons de nommer ces types, leurs caractéristiques et la façon d'écrire des constantes dans un programme.

2.1 Les deux types caractère

Les types caractère correspondent au mot-clé `char`. La norme ANSI prévoit en fait deux types caractère différents obtenus en introduisant dans le spécificateur de type, de façon facultative, un qualificatif de signe, à savoir `signed` ou `unsigned`. Cet attribut intervient essentiellement lorsqu'on utilise un type caractère pour représenter de petits entiers. C'est la raison pour laquelle la norme définit, comme pour les types entiers, le domaine (numérique) minimal des types caractère.

Contrairement à ce qui se passe pour les types entiers, l'absence de qualificatif de signe pour les caractères ne correspond pas systématiquement au type `signed char` mais plus précisément à l'un des deux types `signed char` ou `unsigned char`, ceci suivant l'implémentation et même, parfois, dans une implémentation donnée, suivant certaines options de compilation.

Tableau 3.5 : les deux types caractère du langage C

Dénomination usuelle	Noms de type possibles	Taille (en octets)	Domaine minimal
Caractère non signé	<code>unsigned char</code> <code>char</code> (suivant l'implémentation)	1	0 à 255
Caractère signé	<code>signed char</code> <code>char</code> (suivant l'implémentation)	1	-127 à 127

En C++

Alors que C dispose de deux types caractère, C++ en disposera de trois : `char` (malgré son ambiguïté, ce sera un type à part entière), `unsigned char` et `signed char`.

2.2 Caractéristiques des types caractère

Le tableau 3.6 récapitule les caractéristiques des types caractère. Ces derniers seront ensuite détaillés dans les sections suivantes de ce chapitre.

Tableau 3.6 : les caractéristiques des types caractère

Code associé à un caractère	<ul style="list-style-type: none"> – indépendant de l'attribut de signe ; – dépend de l'implémentation. 	Voir section 2.2.1 de ce chapitre
Caractères existants	<ul style="list-style-type: none"> – au moins le jeu minimal d'exécution ; – ne pas oublier que certains caractères ne sont pas imprimables. 	Voir section 2.2.2 de ce chapitre
Influence de l'attribut de signe	<ul style="list-style-type: none"> – en pratique, aucune, dans les simples manipulations de variables (type conseillé : <code>char</code> ou <code>unsigned char</code>) ; – importante si l'on utilise ce type pour représenter de petits entiers (type conseillé <code>signed char</code>). 	Voir section 2.2.3 de ce chapitre
Manipulation d'octets	Possible par le biais de ce type, compte tenu de l'équivalence entre octet et caractère (type conseillé <code>unsigned char</code>).	Voir section 2.2.4 de ce chapitre

2.2.1 Code associé à un caractère

Les valeurs de type caractère sont représentées sur un octet, au sens large de ce terme, c'est-à-dire correspondant à la plus petite partie adressable de la mémoire. En pratique, le code associé à un caractère est indépendant de l'attribut de signe. Par exemple, on obtiendra exactement le même motif binaire dans `c1` et `c2` avec :

```
unsigned char c1 ;
signed char c2 ;
c1 = 'a' ;
c2 = 'a' ;
```

Cependant, on verra à la section 2.4, que les constantes caractère sont en fait de type `int`. Les instructions précédentes font donc intervenir des conversions d'entier en caractère dont les règles exactes sont étudiées à la section 9 du chapitre 4. Leur examen attentif montrera que, en théorie, cette conservation du motif binaire ne devrait être assurée que dans certains cas : caractères appartenant au jeu minimal d'exécution, variables caractère non signées (cas de `c1` dans notre exemple) dans les implémentations utilisant la représentation en complément à deux. En pratique, cette unicité se vérifie dans toutes les implémentations que nous avons rencontrées et d'ailleurs, beaucoup de programmes se basent sur elle.

Bien entendu, le code associé à un caractère donné dépend de l'implémentation. Certes, le code dit ASCII tend à se répandre, mais comme indiqué à la section 1.3 du chapitre 2, seul le code ASCII restreint a un caractère universel ; et nos caractères nationaux n'y figurent pas !

2.2.2 Caractères existants

La norme précise le jeu minimal de caractères d'exécution dont on doit disposer ; il est présenté à la section 1.1 du chapitre 2. Mais d'autres caractères peuvent apparaître dans une

implémentation donnée. Ainsi, lorsque l'octet occupe 8 bits (comme c'est presque toujours le cas), on est sûr de disposer d'un jeu de 256 caractères parmi lesquels figurent ceux du jeu minimal. Les caractères supplémentaires peuvent être imprimables ou de contrôle. À ce propos, signalons qu'il existe des fonctions standards permettant de connaître la nature (imprimable, alphabétique, numérique, de contrôle...) d'un caractère de code donné ; elles sont présentées au chapitre 18.

2.2.3 Influence de l'attribut de signe

Dans les manipulations de variables de type caractère

En pratique, tant que l'on se contente de manipuler des caractères en tant que tels, l'attribut de signe n'a pas d'importance. C'est le cas dans des situations telles que :

```
signed char c1 ;
unsigned char c2 ;
.....
c2 = c1 ;
c1 = c2 ;
```

Le motif binaire est conservé par affectation. Cependant, là encore, si l'on examine la norme à la lettre, on constate que ces situations font intervenir des conversions (étudiées à la section 9 du chapitre 4) qui, en théorie, n'assurent cette conservation que dans certains cas : caractères appartenant au jeu minimal d'exécution, conversions de signé en non signé dans les implémentations utilisant la représentation en complément à deux. En pratique, ces conversions conservent le motif binaire dans toutes les implémentations que nous avons rencontrées.

Remarque

Si l'on vise une portabilité absolue, on pourra toujours éviter les conversions en évitant les mélanges d'attribut de signe. Cependant, dans ce cas, il faudra tenir compte du fait qu'une constante caractère est de type `int` (voir section suivante « Le type des constantes caractère »), ce qui pourra influencer sur l'initialisation ou sur l'affectation d'une constante à une variable caractère. Si l'on suit la norme à la lettre, on verra que le seul cas de conservation théorique sera celui où l'on utilise le type `char`. En définitive, il faudra choisir entre :

- un type `char` qui assure la portabilité absolue du motif binaire mais qui présente une ambiguïté de signe (soit quand on s'intéresse à sa valeur numérique, soit lorsqu'on compare deux caractères, voir section 3.3.2 du chapitre 4) ;
- le type `unsigned char` qui, en théorie, n'assure la portabilité que dans les implémentations utilisant la représentation en complément à deux) mais qui ne présente plus l'ambiguïté précédente.

Dans les expressions entières

Comme on le verra à la section 9 du chapitre 4, il existe une conversion implicite de `char` en `int` qui pourra intervenir dans :

- des expressions dans lesquelles figurent des variables de type caractère ;
- des affectations du type caractère vers un type entier.

Ces conversions permettent aux types caractère d'être utilisés pour représenter des petits entiers. Dans ce cas, comme on peut s'y attendre, l'attribut de signe intervient pour définir le résultat de la conversion. On verra par exemple que, avec :

```
signed char c1 ;
unsigned char c2 ;
```

l'expression `c2+1` aura toujours une valeur positive, tandis que `c1+1` pourra, suivant la valeur de `c1`, être négative, positive ou nulle. De même, si `n1` et `n2` sont de type `int`, avec ces affectations :

```
n1 = c1 ;
n2 = c2 ;
```

la valeur de `n2` sera toujours positive ou nulle, tandis que celle de `n1` pourra être négative, positive ou nulle.

Les conseils fournis à la section 1.3 de ce chapitre, à propos des types entiers s'appliquent encore ici : si l'objectif est effectivement de réduire la taille de variables destinées à des calculs numériques classiques, il est conseillé d'utiliser systématiquement le type `signed char`.

2.2.4 Manipulations d'octets

Un des atouts du langage C est de permettre des manipulations dites « proches de la machine ». Parmi celles-ci, on trouve notamment les manipulations du contenu binaire d'un objet, indépendamment de son type. *A priori*, tout accès à un objet requiert un type précis défini par l'expression utilisée, de sorte que les manipulations évoquées semblent irréalisables. En fait, un objet est toujours formé d'une succession d'un nombre entier d'octets et un octet peut toujours être manipulé par le biais du type `char`.

Dans ces conditions, il est bien possible de manipuler les différents octets d'un objet quelconque, pour peu qu'on soit en mesure d'assurer la conservation du motif binaire. Cet aspect a été exposé à la section précédente. On a vu que cette conservation avait toujours lieu en pratique, même si en théorie, elle n'était absolue qu'avec le type `char`. Par ailleurs, les manipulations d'octets sont souvent associées à des manipulations au niveau du bit (masque, décalages...) pour lesquelles le type `unsigned char` sera plus approprié (voir section 6 du chapitre 4). Le type `unsigned char` constituera donc généralement le meilleur choix possible.

2.3 Écriture des constantes caractère

Il existe plusieurs façons d'écrire les constantes caractère dans un programme. Elles ne sont pas totalement équivalentes.

2.3.1 Les caractères « imprimables »

Les constantes caractère correspondant à des caractères imprimables peuvent se noter de façon classique, en écrivant entre apostrophes (ou *quotes*) le caractère voulu, comme dans ces exemples :

```
'a' 'Y' '+' '$' '0' '<' /* caractères du jeu minimal d'exécution */
'é' 'à' 'ç' /* n'existent que dans certaines implémentations */
```

On notera bien que l'utilisation de caractères n'appartenant pas au jeu minimal conduit à des programmes qu'on pourrait qualifier de « semi-portables ». En effet, une telle démarche présente les caractéristiques suivantes :

- elle est plus portable que celle qui consisterait à fournir directement le code du caractère voulu car on dépendrait alors de l'implémentation elle-même ;
- elle n'est cependant portable que sur les implémentations qui possèdent le graphisme en question (quel que soit son codage).

Par exemple, la notation 'é' représente bien le caractère é dans toute implémentation où il existe, quel que soit son codage ; mais cette notation n'est pas utilisable dans les autres implémentations. À ce propos, il faut bien voir que cette notation du caractère imprimable n'est visible qu'à celui qui saisit ou qui lit un programme. Dès qu'on travaille sur des fichiers source, on a affaire à des suites d'octets représentant chacun un caractère. Par exemple, il n'est pas rare de saisir un caractère é dans une implémentation et de le voir apparaître différemment lorsqu'on exploite le même programme source dans une autre implémentation.

2.3.2 Les caractères disposant d'une « séquence d'échappement »

Certains caractères non imprimables possèdent une représentation conventionnelle dite « séquence d'échappement », utilisant le caractère \ (antislash)³. Dans cette catégorie, on trouve également quelques caractères qui, bien que disposant d'un graphisme, jouent un rôle particulier de délimiteurs, ce qui les empêche d'être notés de manière classique entre deux apostrophes.

Tableau 3.7 : les caractères disposant d'une séquence d'échappement

Séquence d'échappement	Code ASCII (hexadécimal)	Abréviation usuelle	Signification
\a	07	BEL	Cloche ou bip (<i>alert ou audible bell</i>)
\b	08	BS	Retour arrière (<i>Backspace</i>)
\f	0C	FF	Saut de page (<i>Form Feed</i>)
\n	0A	LF	Saut de ligne (<i>Line Feed</i>)
\r	0D	CR	Retour chariot (<i>Carriage Return</i>)
\t	09	HT	Tabulation horizontale (<i>Horizontal Tabulation</i>)
\v	0B	VT	Tabulation verticale (<i>Vertical Tabulation</i>)
\\	5C	\	
\'	2C	'	
\»	22	«	
\?	3F	?	

Si le caractère \ apparaît suivi d'un caractère différent de ceux qui sont mentionnés ici, le comportement du programme est indéterminé. Par ailleurs, une implémentation peut introduire d'autres séquences d'échappement ; il lui est cependant conseillé d'éviter les minuscules qui

3. Aussi appelé « barre inverse » ou « contre-slash » (*back-slash* en anglais).

sont réservées pour une future extension de la norme. Par essence, l'emploi de cette notation est totalement portable, quel que soit le code correspondant dans l'implémentation. En revanche, tout recours à un caractère de contrôle n'appartenant pas à cette liste nécessite l'introduction directe de son code, ce qui n'assure la portabilité qu'entre implémentations utilisant le même codage.

Remarque

La notation sous forme d'une séquence d'échappement ne dispense nullement de l'utilisation des apostrophes dans l'écriture d'une constante caractère. Ainsi, il faudra bien écrire `'\n'` et non simplement `\n`. Bien entendu, quand cette même séquence d'échappement apparaîtra dans une chaîne constante, ces apostrophes n'auront plus aucune raison d'être. Par exemple, on écrira bien :

```
"bonjour\nmonsieur"
```

2.3.3 Écriture d'un caractère par son code

Il est possible d'utiliser directement le code du caractère, en l'exprimant, toujours à la suite du caractère `\` :

- soit sous forme **octale** ;
- soit sous forme **hexadécimale** précédée de `x`.

Voici quelques exemples de notations équivalentes (sur une même ligne), dans le code ASCII restreint :

```
'A'    '\x41'    '\101'
'\n'   '\x0d'   '\15'    '\015'
'\a'   '\x07'   '\x7'    '\07'    '\007'
```

La notation octale doit comporter de 1 à 3 chiffres ; la notation hexadécimale n'est pas soumise à des limites. Ainsi, `'\4321'` est incorrect, tandis que `'\x4321'` est correct. Toutefois, dans le dernier cas, le code obtenu en cas de dépassement de la capacité d'un octet, n'est pas précisé par la norme. Il est donc recommandé de ne pas utiliser cette tolérance⁴.

D'une manière générale, ces notations, manifestement non portables, doivent être réservées à des situations particulières telles que :

- besoin d'un caractère de contrôle non prévu dans le jeu d'exécution, par exemple `ACK`, `NACK`... Dans ce cas, on minimisera le travail de portage d'une machine à une autre en prenant bien soin de définir une seule fois, au sein d'un fichier en-tête, chacun de ces caractères par une instruction de la forme :

```
#define ACK 0x5
```

- besoin de décrire le motif binaire contenu dans un octet ; c'est notamment le cas lorsqu'on doit recourir à un masque binaire.

4. D'autant plus que certaines implémentations se permettent de limiter d'office (souvent à 3) le nombre de caractères pris effectivement en compte dans la notation hexadécimale.

Remarques

1. Le caractère `\` suivi d'un caractère autre que ceux du tableau 3.7 ou d'un chiffre de 0 à 7, est simplement ignoré. Ainsi, dans le cas du code ASCII, `\9` correspond au caractère 9 (de code ASCII 57), tandis que `\7` correspond au caractère de code ASCII 7, c'est-à-dire la « cloche ».
2. Avec la notation hexadécimale ou octale, comme avec la notation sous forme d'une séquence d'échappement présentée à la section précédente 2.3.2, il ne faut pas oublier les apostrophes délimitant une constante caractère. Bien entendu, cette remarque ne s'appliquera plus au cas des constantes chaînes.

2.4 Le type des constantes caractère

Assez curieusement, la norme prévoit que :

Toute constante caractère est de type `int` et la valeur correspondante est obtenue comme si l'on convertissait une valeur de type `char` (dont l'attribut de signe dépend de l'implémentation) en `int`.

L'explication réside probablement dans le lien étroit existant en C entre caractères et entiers. Si l'on admet que les types caractère correspondent à des types entiers de petite taille, il n'est alors pas plus choquant de dire qu'une notation comme `'a'` est de type `int` que de dire qu'une « petite constante numérique » comme `+43` était de type `int` (et non `short` !).

Dans ces conditions, on peut s'interroger sur le fait que, suivant les implémentations, le type `char` peut être signé ou non, de sorte que, suivant les règles de conversion étudiées dans le chapitre suivant, le résultat peut être parfois négatif. Nous allons examiner deux situations :

- on utilise les constantes caractère de façon naturelle, c'est-à-dire pour représenter de vrais caractères ;
- on utilise les constantes caractère pour leur valeur numérique.

2.4.1 Utilisation naturelle des constantes caractère

En pratique, les trois instructions suivantes placeront le même motif dans `c1`, `c2` et `c3` (la notation α désignant un caractère quelconque) :

```
char c1 = '\alpha' ;  
unsigned char c2 = '\alpha' ;  
signed char c3 = '\alpha' ;
```

Il en ira de même si la constante caractère est exprimée sous forme octale ou hexadécimale.

Cependant, si l'on examine la norme à la lettre, on constate que ces situations, hormis la première, font intervenir une suite de deux conversions de `char` en `int`, puis en `char`. En théorie (voir section 9 du chapitre 4), elles n'assurent la conservation du motif binaire que dans certains cas : caractères appartenant au jeu minimal d'exécution, conversions de signé en non signé dans les implémentations utilisant le complément à deux. En pratique, ces conversions conservent le motif binaire dans toutes les implémentations que nous avons rencontrées. Si toutefois on cherche une portabilité absolue, on pourra se limiter à l'utilisation du type `char`, à condition que l'ambiguïté correspondante ne s'avère pas gênante (voir section précédente 2.2.3).

2.4.2 Utilisation des constantes caractère pour leur valeur numérique

Il s'agit du cas où une telle constante apparaît dans une expression numérique ou dans une affectation à une variable entière, ce qui peut se produire lorsqu'on s'intéresse à la valeur du code du caractère correspondant. Dans ce cas, l'implémentation intervient, non seulement sur la valeur obtenue, mais éventuellement sur son signe. Voici un exemple dans lequel on suppose que le code du caractère `é` est supérieur à 127, dans une implémentation codant les caractères sur 8 bits et utilisant la représentation en complément à deux :

```
int n = 'é' ; /* la valeur de n sera >0 si char est signé */
           /* et >0 si char n'est pas signé */
```

On peut simplement affirmer que l'effet de cette déclaration sera équivalent à :

```
char c = 'é' ; /* char est signé ou non suivant l'implémentation */
int n ;
.....
n = c ;
```

D'une manière comparable, dans la même implémentation (octets de 8 bits, représentation en complément à deux) :

```
int n = '\xFE' ; /* -2 si le type char est signé par défaut */
                /* 254 si le type char n'est pas signée par défaut */
```

On notera bien que, alors qu'on pouvait choisir l'attribut de signe d'une variable de type `char`, il n'en va plus de même pour une constante. On peut toutefois faire appel à l'opérateur de `cast` comme dans :

```
int n = (signed char) '\xFE' ; /* conv char -> signed char -> int */
                                   /* -2 dans le cas du complément à deux */
int n = (unsigned char) '\xFE' ; /* conv char -> unsigned signed char -> int */
                                   /* 254 dans le cas du complément à deux */
```

Remarques

Ici, il est important de ne pas confondre valeur et motif binaire. Le motif binaire associé à une constante caractère est bien constant après conversion dans un type `char` de type quelconque (du moins, en pratique) ; en revanche, la valeur numérique correspondante de type `int`, ne l'est pas.

La norme autorise des constantes caractère de la forme `'xy'` voire `'xyz'`, contenant plusieurs caractères imprimables. Certes, une telle particularité peut se justifier par le fait que la constante produite est effectivement de type `int` et non de type `char` ; elle reste cependant d'un emploi malaisé et, de toute façon, la valeur ainsi obtenue dépend de l'implémentation.

En C++

En C++, les constantes caractère seront effectivement de type `char`, et non plus de type `int`.

3. Le fichier limits.h

3.1 Son contenu

Ce fichier en-tête contient, sous forme de constantes ou macros (définies par la directive `#define`), de nombreuses informations concernant le codage des entiers et les limitations qu'elles imposent dans une implémentation donnée.

En voici la liste, accompagnée de la valeur minimale qu'on est assuré d'obtenir pour chaque type d'entier quelle que soit l'implémentation. On notera la présence d'une constante `MB_LEN_MAX` relative aux caractères dits « multi-octets », d'usage assez peu répandu, et dont nous parlerons au chapitre 22.

Tableau 3.8 : les valeurs définies dans le fichier limits.h

Symbole	Valeur minimale	Signification
<code>CHAR_BIT</code>	8	Nombre de bits dans un caractère
<code>SCHAR_MIN</code>	-127	Plus petite valeur (négative) du type <code>signed char</code>
<code>SCHAR_MAX</code>	+127	Plus grande valeur du type <code>signed char</code>
<code>UCHAR_MAX</code>	255	Plus grande valeur du type <code>unsigned char</code>
<code>CHAR_MIN</code>		Plus petite valeur du type <code>char</code> (<code>signed char</code> ou <code>unsigned char</code> suivant l'implémentation, ou même suivant les options de compilation)
<code>CHAR_MAX</code>		Plus grande valeur du type <code>char</code> (<code>signed char</code> ou <code>unsigned char</code> suivant l'implémentation, ou même suivant les options de compilation)
<code>MB_LEN_MAX</code>	1	Nombre maximal d'octets dans un caractère multi-octets (quel que soit le choix éventuel de localisation)
<code>SHRT_MIN</code>	- 32 767	Plus petite valeur du type <code>short int</code>
<code>SHRT_MAX</code>	+ 32 767	Plus grande valeur du type <code>short int</code>
<code>USHRT_MAX</code>	65 535	Plus grande valeur du type <code>unsigned short int</code>
<code>INT_MIN</code>	- 32 767	Plus petite valeur du type <code>int</code>
<code>INT_MAX</code>	+ 32 767	Plus grande valeur du type <code>int</code>
<code>UINT_MAX</code>	65 535	Plus grande valeur du type <code>unsigned int</code>
<code>LONG_MIN</code>	- 2 147 483 647	Plus petite valeur du type <code>long int</code>
<code>LONG_MAX</code>	+ 2 147 483 647	Plus grande valeur du type <code>long int</code>
<code>ULONG_MAX</code>	4 294 967 295	Plus grande valeur du type <code>unsigned long int</code>

3.2 Précautions d'utilisation

On notera bien que la norme impose peu de contraintes au type des constantes définies dans `limits.h`. En général, on trouvera des définitions de ce genre :

```
#define INT_MAX +32767
```

Le symbole `INT_MAX` sera donc remplacé par le préprocesseur par la constante `+32 767`, laquelle sera de type `int`. Dans ces conditions, on évitera certains calculs arithmétiques risquant de conduire à des dépassements de capacité dans le type `int`. Le programme suivant montre ce qu'il ne faut pas faire, puis ce qu'il faut faire, pour calculer la valeur de `INT_MAX+5` :

Exemple de mauvais et de bon usage de la constante INT_MAX

```
#include <stdio.h>
#include <limits.h>
int main()
{
    int n ;
    long q ;
    q = INT_MAX + 5 ;                /* calcul incorrect */
    printf ("INT_MAX+5 = %ld\n", q) ;
    q = (long)INT_MAX + 5 ;         /* calcul correct */
    printf ("INT_MAX+5 = %ld\n", q) ;
}
```

```
INT_MAX+5 = -32764
```

```
INT_MAX+5 = 32772
```

4. Les types flottants

Nous commencerons par rappeler brièvement en quoi consiste le codage en flottant d'un nombre réel et quels sont les éléments caractéristiques d'un codage donné : précision, limitations, *epsilon machine*... Puis nous examinerons les trois types de flottants prévus par la norme, leur nom et leurs caractéristiques respectives. Nous terminerons par les différentes façons d'écrire des constantes flottantes dans un programme source.

4.1 Rappels concernant le codage des nombres en flottant

Les types flottants (appelés parfois, un peu à tort, réels) servent à représenter de manière approchée une partie des nombres réels. Ils s'inspirent de la notation scientifique des calculettes, dans laquelle on exprime un nombre sous forme d'une mantisse et d'un exposant correspondant à une puissance de 10, comme dans 0.453 E 15 (mantisse 0,453, exposant 15) ou dans 45.3 E 13 (mantisse 45,3, exposant 13). Le codage en flottant se distinguera cependant de cette notation scientifique sur deux points :

- le codage de la mantisse : il est généralement fait en binaire et non plus en base 10 ;

- la base utilisée pour l'exposant : on n'a guère de raison d'employer une base décimale. En général, des bases de 2 ou de 16 seront utilisées car elles facilitent grandement les calculs de la machine (attention, l'utilisation d'une base 16 n'est pas incompatible avec le codage en binaire des valeurs de la mantisse et de l'exposant) .

D'une manière générale, on peut dire que la représentation d'un nombre réel en flottant revient à l'approcher par une quantité de la forme

$$s \cdot m \cdot b^e$$

dans laquelle :

- s représente un signe, ce qui revient à dire que s vaut soit -1, soit +1 ;
- m représente la mantisse ;
- e représente l'exposant, tel que : $e_{\min} \leq e \leq e_{\max}$;
- b représente la base.

La base b (en pratique 2 ou 16) est fixée pour une implémentation donnée⁵. Il n'en reste pas moins qu'un même nombre réel peut, pour une valeur b donnée, être approché de plusieurs façons différentes par la formule précédente. Par exemple, si un nombre est représentable par le couple de valeurs (m, e) , il reste représentable par le couple de valeurs $(b/m, e+1)$ ou $(mb, e-1)$...

Pour assurer l'unicité de la représentation, on fait donc appel à une contrainte dite de « normalisation ». En général, il s'agit de :

$$1/b \leq m < 1$$

Dans le cas de la notation scientifique des calculettes, l'application de cette contrainte conduirait à une mantisse commençant toujours par 0 et dont le premier chiffre après le point décimal est non nul. Par exemple, 0.2345 et 0.124 seraient des mantisses normalisées, tandis que 3.45 ou 0.034 ne le seraient pas⁶.

4.2 Le modèle proposé par la norme

Contrairement à ce qui se passe, en partie du moins, pour les nombres entiers, la norme ANSI n'impose pas de contraintes précises quant à la manière dont une implémentation représente les types flottants. Elle se contente de proposer un modèle théorique possible, dont le principal avantage est de donner une définition formelle d'un certain nombre de paramètres caractéristiques tels que la précision ou l'epsilon machine. Ce modèle correspond à la formule précédente (voir section 4.1), dans laquelle on explicite la mantisse de la façon suivante, le coefficient f_j étant non nul si le nombre est non nul :

$$m = \sum_{k=1}^p f_k b^{-k}$$

5. En toute rigueur, rien n'interdirait à une implémentation d'utiliser une base (par exemple, 2) pour un type (par exemple, float) et une autre base (par exemple, 16) pour un autre type (par exemple, long double). Cela conduirait toutefois à complexifier inutilement l'unité centrale, de sorte qu'en pratique, cette situation ne se rencontre pas !
6. Attention à ne pas confondre la contrainte de normalisation utilisée pour coder un nombre flottant en mémoire avec celle qu'utilise printf pour afficher un tel nombre.

Certes, cette formule définit la mantisse de façon unique, dès lors que p , b et les limites e_{\min} et e_{\max} sont fixées. Comme on s'en doute, elles dépendront de l'implémentation et du type de flottant utilisé (`float`, `double` ou `long double`). Mais il ne s'agit que d'un modèle théorique de comportement et, même si la plupart des implémentations s'en inspirent, quelques petits détails de codage peuvent apparaître :

- élimination de certains bits superflus de la mantisse ; par exemple, lorsque la base est égale à 2, le premier bit est toujours à 1 et certaines implémentations ne le conservent pas, ce qui double la capacité ;
- réservation, comme le propose la norme IEEE 754, de certains motifs binaires pour représenter des nombres infinis ou des quantités non représentables.

4.3 Les caractéristiques du codage en flottant

Le codage en entier n'a guère d'autres conséquences que de limiter le domaine des valeurs utilisables. Dans le cas du codage en flottant, les conséquences sont moins triviales et nous allons les examiner ici, en utilisant parfois le modèle théorique présenté précédemment.

4.3.1 Représentation approchée

Le codage en flottant permet de représenter un nombre réel de façon approchée, comme on le fait dans la vie courante en approchant le nombre réel π par 3.14 ou 3.14159... La notion de représentation approchée paraît alors relativement naturelle. En revanche, lorsqu'on a affaire à un nombre décimal tel que 0.1, qui s'exprime de manière exacte dans notre système décimal, on peut être surpris de ce qu'il ne s'exprime plus toujours de façon exacte une fois codé en flottant⁷. Voici un petit programme illustrant ce phénomène, dans une implémentation où la base b de l'exposant est égale à 2 :

Conséquences de la représentation approchée des nombres flottants

```
include <stdio.h>
int main()
{ float x = 0.1 ;
  printf ("x avec 1  decimale  : %.1e\n", x) ;
  printf ("x avec 10  decimales : %.10e\n", x) ;
}

x avec 1  decimale  : 1.0e-01
x avec 10  decimales : 1.0000000149e-01
```

Cependant, la norme impose aux entiers dont la valeur absolue est inférieure à une certaine limite d'être représentés de façon exacte en flottant, de façon à ce qu'un cycle de conversion

7. Pour s'en convaincre, il suffit d'exprimer la valeur 0,1 dans le modèle de comportement proposé par la norme : on s'aperçoit que la conversion en binaire (exprimée en puissances négatives de b) conduit dans les cas usuels ($b=2$ ou $b=16$) à une mantisse m ayant un nombre infini de décimales, et donc obligatoirement à une approximation, quel que soit le nombre de bits réservés à m .

entier → flottant → entier permette de retrouver la valeur d'origine. Ces limites dépendent à la fois du type de flottant concerné et de l'implémentation ; elles sont précisées dans le tableau 3.8, qui montre qu'elles sont toujours au moins égales à 1E6.

4.3.2 Notion de précision

On peut définir la précision d'une représentation flottante :

- soit en considérant le nombre de chiffres en base b , c'est-à-dire finalement la valeur de p dans le modèle défini par la norme et présenté à la section 4.2 ; cette valeur est définie de façon exacte ;
- soit en cherchant à exprimer cette précision en termes de chiffres décimaux significatifs ; en théorie, on peut montrer que p chiffres exacts en base b conduisent toujours à au moins q chiffres décimaux exacts, avec $q = (p-1) \cdot \log_{10} b$; autrement dit, tout nombre entier d'au plus q chiffres s'exprime sans erreur en flottant.

Ces différentes valeurs sont fournies dans le fichier `float.h` décrit à la section 5.

4.3.3 Limitations des valeurs représentables

Dans le cas du type entier, les valeurs représentables appartiennent simplement à un intervalle de l'ensemble des entiers relatifs. Dans le cas des flottants, on a affaire à une limitation des valeurs de l'exposant e_{\min} et e_{\max} , lesquelles conduisent en fait à une limitation de l'amplitude de la valeur absolue du nombre. Les valeurs réelles représentables appartiennent donc à deux intervalles disjoints, de la forme :

$$[-x_{\max}, -x_{\min}] \quad [x_{\min}, x_{\max}] \quad \text{avec } x_{\min} = b^{e_{\min}} \quad \text{et} \quad x_{\max} = b^{e_{\max}}$$

En outre, la valeur 0 (qui n'appartient à aucun de ces deux intervalles) est toujours représentable de façon exacte ; l'unicité de sa représentation nécessite l'introduction d'une contrainte conventionnelle, par exemple, mantisse nulle, exposant 1.

4.3.4 Non-respect de certaines règles de l'algèbre

La représentation approchée des types flottants induit des différences de comportement par rapport à l'algèbre traditionnelle.

Certes, la commutativité des opérations est toujours respectée ; ainsi les opérations $a+b$ ou $b+a$ donneront-elles toujours le même résultat (même si celui-ci n'est qu'une approximation de la somme).

En revanche, si a et b désignent des valeurs réelles et si x' désigne l'approximation en flottant de l'expression x , on n'est pas assuré que les conditions suivantes soient vérifiées :

$$(a + b)' = a' + b'$$

$$(a' + b')' = a' + b'$$

Par exemple :

```
float x = 0.1, y = 0.1
.....
if (x + y == 0.2)          /* peut être vrai ou faux */
```

De façon comparable, on n'est pas assuré que ces conditions soient vérifiées⁸ :

$$(3 * a)' = 3 * a'$$

$$(3 * a')' = 3 * a'$$

Par exemple :

```
float x = 0.1 ;
if (3*x == 0.3)      /* peut être vrai ou faux */
```

Par ailleurs, l'associativité de certaines opérations n'est plus nécessairement respectée. On n'est plus assuré que :

$$a' + (b'+c')' = (a' + b')' + c'$$

Tout ceci se compliquera encore un peu plus avec l'incertitude qui règne en C sur l'ordre d'évaluation des opérateurs commutatifs, comme nous le verrons à la section 2.1.4 du chapitre 4.

4.3.5 Notion d'epsilon machine

Compte tenu de la représentation approchée du type flottant, on peut aisément trouver des nombres *eps* tels que la représentation de la somme de *eps*+1 soit identique à celle de 1, autrement dit que la condition suivante soit vraie :

```
1 + eps == 1
```

La plus grande de ces valeurs se nomme souvent « l'epsilon machine ». On peut montrer qu'elle est définie par :

$$\text{eps} = b^{1-p}$$

où *b* et *p* sont définis par le modèle de comportement ANSI présenté à la section 4.2.

On en trouvera la valeur pour chacun des types flottants dans le fichier `float.h` décrit à la section 5.

4.4 Représentation mémoire et limitations

La norme ANSI prévoit les trois types de flottants suivants :

Tableau 3.9 : les trois types flottants prévus par la norme

Spécificateur de type	Domaine minimal	Précision minimale	Propriétés imposées par la norme
<code>float</code>	10^{-37} à 10^{+37}	6	Les entiers d'au plus 6 chiffres sont toujours représentés de façon exacte.
<code>double</code>	10^{-37} à 10^{+37}	10	Les entiers d'au plus 10 chiffres sont toujours représentés de façon exacte.

8. Ici, il n'est pas nécessaire de considérer $3'$ car, dans toute implémentation, $3' = 3$ puisque tout nombre entier d'au plus *q* chiffres s'exprime exactement en flottant (voir section 4.3.2) et que la valeur de *q* est toujours supérieure ou égale à 6 (voir section 5).

Spécificateur de type	Domaine minimal	Précision minimale	Propriétés imposées par la norme
<code>long double</code>	10^{-37} à 10^{+37}	10	Les entiers d'au plus 10 chiffres sont toujours représentés de façon exacte.

Bien entendu, les caractéristiques exactes de chacun de ces types dépendent à la fois de l'implémentation et du type concerné. Un certain nombre d'éléments sont cependant généralement communs, dans une implémentation donnée, aux trois types de flottants :

- la technique d'approximation (arrondi par défaut, par excès, au plus proche...) ;
- la valeur de la base de l'exposant b ;
- la manière dont la mantisse est normalisée.

D'autres éléments, en revanche, dépendent effectivement du type de flottant (et aussi de l'implémentation) à savoir :

- le nombre de bits utilisés pour coder la mantisse m ;
- le nombre de bits utilisés pour coder l'exposant e .

D'une manière générale, le fichier `float.h` contient bon nombre d'informations concernant les caractéristiques des flottants.

Remarques

1. La première définition du langage C (Kernighan et Ritchie) ne comportait pas le type `long double`. En outre, `long float` y apparaissait comme un synonyme de `double` ; cette possibilité a disparu de la norme.
2. Certaines implémentations acceptent des valeurs flottantes non normalisées, c'est-à-dire des valeurs dans lesquelles la mantisse comporte un ou plusieurs de ses premiers chiffres (en base b) nuls. Dans ce cas, il devient possible de manipuler des valeurs inférieures en valeur absolue au minimum imparti au type, moyennant, alors une perte de précision...

4.5 Écriture des constantes flottantes

Comme dans la plupart des langages, les constantes réelles peuvent s'écrire indifféremment suivant l'une des deux notations :

- décimale ;
- exponentielle.

La notation décimale doit obligatoirement comporter un point (correspondant à notre virgule). La partie entière ou la partie décimale peuvent être omises (mais bien sûr pas toutes les deux en même temps !). En voici quelques exemples corrects :

12.43 -0.38 -.38 4. .27

En revanche, la constante 47 serait considérée comme entière et non comme flottante. Dans la pratique, ce fait aura peu d'importance⁹, compte tenu des conversions automatiques qui seront mises en place par le compilateur (et dont nous parlerons au chapitre suivant).

La notation exponentielle utilise la lettre e (ou E) pour introduire un exposant entier (puissance de 10), avec ou sans signe. La mantisse peut être n'importe quel nombre décimal ou entier (le point peut être absent dès qu'on utilise un exposant). Voici quelques exemples corrects (les exemples d'une même ligne étant équivalents) :

4.25E4	4.25e+4	42.5E3
54.27E-32	542.7E-33	5427e-34
48e13	48.e13	48.0E13

4.6 Le type des constantes flottantes

Par défaut, toutes les constantes sont créées par le compilateur dans le type `double`. Il est cependant possible d'imposer à une constante flottante :

- d'être du type `float`, en faisant suivre son écriture de la lettre F (ou f), comme dans `1.25E+03f` ; cela permet de gagner un peu de place mémoire, en contrepartie d'une éventuelle perte de précision ;
- d'être du type `long double`, en faisant suivre son écriture de la lettre L (ou l), comme dans `1.0L` ; cela permet de gagner en précision, en contrepartie d'une perte de place mémoire ; c'est aussi le seul moyen de représenter les valeurs très grandes ou très petites (bien qu'un tel besoin soit rare en pratique).

4.7 En cas de dépassement de capacité dans l'écriture des constantes

Contrairement à ce qui se produit pour les entiers, la manière dont est écrite une constante flottante impose son type, de manière unique : `float`, `double` ou `long double`. Dans chacun de ces trois cas, vous avez affaire à des limitations propres, à la fois :

- vers « le bas » : une constante de valeur absolue trop petite ne peut être représentée (avec une erreur relative d'approximation raisonnable) ; on parle alors de sous-dépassement de capacité (en anglais *underflow*) ;
- vers « le haut » : une constante de valeur absolue trop grande ne peut être représentée (avec une erreur relative d'approximation raisonnable) ; on parle alors de dépassement de capacité (en anglais *overflow*).

Là encore, suivant les compilateurs, on pourra obtenir : un diagnostic de compilation, une valeur fantaisiste ou une utilisation des conventions IEEE 754 (présentées à la section 2.2.2) en cas de dépassement de capacité, une valeur fantaisiste ou une valeur nulle en cas de sous-dépassement de capacité.

9. Si ce n'est au niveau du temps d'exécution.

5. Le fichier float.h

Ce fichier contient, sous forme de constantes ou de macros (définies par la directive `#define`), de nombreuses informations concernant :

- les caractéristiques du codage des flottants tel qu'il est défini par le modèle théorique de comportement proposé par la norme et présenté à la section 4.2 : base b , précision p en base b ou précision en base 10, epsilon machine...
- les limitations correspondantes : e_{\min} , e_{\max} , x_{\min} , x_{\max} ...

En voici la liste. On notera que, à l'exception des symboles `FLT_ROUNDS` et `FLT_RADIX` qui concernent les trois types flottants (`float`, `double` et `long double`), les autres symboles sont définis pour les trois types avec le même suffixe et un préfixe indiquant le type concerné :

- `FLT` : le symbole correspondant (par exemple, `FLT_MIN_EXP`) concerne le type `float` ;
- `DBL` : le symbole correspondant (par exemple, `DBL_MIN_EXP`) concerne le type `double` ;
- `LDBL` : le symbole correspondant (par exemple, `LDBL_MIN_EXP`) concerne le type `long double`.

Tableau 3.10 : le contenu du fichier float.h

Symbole	Valeur minimale	Signification
<code>FLT_RADIX</code>	2	Base b telle que définie à la section 4.2
<code>FLT_ROUNDS</code>		Méthode utilisée pour déterminer la représentation d'un nombre réel donné : -1 : indéterminée 0 : arrondi vers zéro 1 : arrondi au plus proche 2 : arrondi vers plus l'infini 3 : arrondi vers moins l'infini autre : méthode définie par l'implémentation
<code>FLT_MANT_DIG</code> <code>DBL_MANT_DIG</code> <code>LDBL_MANT_DIG</code>		Précision (p dans la formule de la section 4.2)
<code>FLT_DIG</code> <code>DBL_DIG</code> <code>LDBL_DIG</code>	6 10 10	Valeur q , telle que tout nombre décimal de q chiffres puisse être exprimé sans erreur en notation flottante ; on peut montrer que : $q = (p-1) \cdot \log_{10} b$ (+1 si b est puissance de 10)
<code>FLT_MIN_EXP</code> <code>DBL_MIN_EXP</code> <code>LDBL_MIN_EXP</code>		Plus petit nombre négatif n tel que FLT_RADIX^{n-1} soit un nombre flottant normalisé ; il s'agit de e_{\min} tel qu'il est défini à la section 4.2
<code>FLT_MIN_10_EXP</code> <code>DBL_MIN_10_EXP</code> <code>LDBL_MIN_10_EXP</code>	-37 -37 -37	Plus petit nombre négatif n tel que 10^n soit dans l'intervalle des nombres flottants normalisés ; on peut montrer que : $n = \log_{10} b^{e_{\min}-1}$
<code>FLT_MAX_EXP</code> <code>DBL_MAX_EXP</code> <code>LDBL_MAX_EXP</code>		Plus grand nombre n tel que FLT_RADIX^{n-1} soit un nombre flottant fini représentable ; il s'agit de e_{\max} tel qu'il est défini à la section 4.2

Symbole	Valeur minimale	Signification
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	+37 +37 +37	Plus grand entier n tel que 10^n soit dans l'intervalle des nombres flottants finis représentables ; on peut montrer que : $n = \log_{10}((1-b^{-p})b^{e_{\max}})$
FLT_MAX DBL_MAX LDBL_MAX	1e37 1e37 1e37	Plus grande valeur finie représentable ; on peut montrer qu'il s'agit de : $(1-b^{-p})b^{e_{\max}}$
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1e-5 1e-9 1e-9	Écart entre 1 et la plus petite valeur supérieure à 1 qui soit représentable ; il s'agit de ce que l'on nomme généralement l'epsilon machine dont on montre qu'il est égal à : b^{1-p}
FLT_MIN DBL_MIN LDBL_MIN	1e-37 1e-37 1e-37	Plus petit nombre flottant positif normalisé. On montre qu'il est égal à : $b^{e_{\min}-1}$

6. Déclarations des variables d'un type de base

Le tableau 3.11 récapitule les différents éléments pouvant intervenir dans la déclaration des variables d'un type de base. Ils seront détaillés dans les sections indiquées.

Tableau 3.11 : déclaration de variables d'un type de base

Rôle d'une déclaration	<ul style="list-style-type: none"> – associe un spécificateur de type (éventuellement complété de qualifieurs et d'une classe de mémorisation), à un déclarateur ; – dans le cas des types de base, le déclarateur se limite au nom de la variable. 	voir section 6.1
Initialisation variables classe automatique (ou registre)	<ul style="list-style-type: none"> – aucune initialisation par défaut ; – initialisation explicite par expressions quelconques. 	voir section 6.2
Initialisation variables classe statique	<ul style="list-style-type: none"> – initialisation par défaut à zéro ; – initialisation explicite par des expressions constantes. 	voir section 6.2
Qualifieurs (<code>const</code> , <code>volatile</code>)	<ul style="list-style-type: none"> – <code>const</code> : la variable ne peut pas voir sa valeur modifiée ; – <code>volatile</code> : la valeur de la variable peut changer, indépendamment des instructions du programme ; – une variable constante doit être initialisée (il existe deux rares exceptions – voir remarque 3 de la section 6.3.2). 	voir section 6.3
Classe de mémorisation	<ul style="list-style-type: none"> – <code>extern</code> : pour les redéclarations de variables globales ; – <code>auto</code> : pour les variables locales (superflu) ; – <code>static</code> : variable rémanente ; – <code>register</code> : demande de maintien dans un registre. 	voir chapitre 8

6.1 Rôle d'une déclaration

6.1.1 Quelques exemples simples

Comme on s'y attend, la déclaration d'une variable d'un type de base permet de préciser son nom et son type, par exemple :

```
unsigned int n ;    /* n est de type unsigned int */
```

On peut déclarer plusieurs variables dans une seule instruction. Par exemple :

```
unsigned int n, p ;
```

est équivalente à :

```
unsigned int n ;  
unsigned int p ;
```

Un même type peut être défini par des spécificateurs de type différents. Par exemple, ces quatre instructions sont équivalentes :

```
short int p  
signed short p  
signed short int p ;  
short p ;
```

Les tableaux 3.1, 3.5 et 3.9 fournissent les différents spécificateurs de type qu'il est possible d'utiliser pour un type donné.

6.1.2 Les déclarations de variables en général

Tant qu'on se limite à des variables d'un type de base, les déclarations restent relativement simples puisque, comme dans les précédents exemples, elles associent un simple identificateur à un spécificateur de type. On peut cependant y trouver quelques informations supplémentaires, parmi les suivantes :

- une valeur initiale de la variable, comme dans :

```
int n, p=5, q ;    /* n, p et q sont des int; p est initialisée à 5 */
```

- un ou plusieurs qualifieurs (*const* ou *volatile*) associés au spécificateur de type et qui concernent donc l'ensemble des variables de la déclaration, par exemple :

```
const float x, y ;    /* x et y sont des float constants */
```

- une classe de mémorisation associée, elle aussi, à l'ensemble des variables de la déclaration, par exemple :

```
static int n, q ;    /* n et q sont déclarés avec la classe de mémorisation static */
```

Les deux premiers points (valeur initiale et qualifieurs) sont examinés ici, dans le seul cas cependant des variables d'un type de base. Pour les autres types, on trouvera des compléments d'information dans les chapitres correspondants (tableaux, pointeurs, structures, unions,