



Ressourcesinformatiques

du C au C++

De la programmation procédurale à l'objet

2^{ème} édition

Frédéric DROUILLON

Téléchargement
www.editions-eni.fr



Les éléments à télécharger sont disponibles à l'adresse suivante :
<http://www.editions-eni.fr>
Saisissez la référence ENI du livre **RI2CC** dans la zone de recherche
et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Introduction

1. Objectif : apprendre à programmer	25
2. Contenu	25
3. Librairies graphiques	26
4. Public visé.	26
5. Comment apprendre à programmer ?	27
5.1 Comprendre n'est pas savoir faire	27
5.2 Trois niveaux de difficulté	28
5.2.1 Maîtriser les outils	28
5.2.2 Résoudre un problème.	29
5.2.3 Concevoir un programme	30
5.3 Un apprentissage non linéaire	30
6. Organisation du livre.	31
6.1 Chapitres	31
6.2 Solutions des exercices	32
7. Environnements de développement	32
8. Remerciements	33
9. Conclusion : la programmation comme écriture.	34

Chapitre 1

Variables simples

1. Introduction dans la programmation C.	35
1.1 Qu'est-ce que programmer ?	35
1.2 Le positionnement par rapport à l'ordinateur.	36
1.3 Les logiciels de la programmation	37
1.3.1 Le compilateur	37
1.3.2 L'IDE : environnement de développement.	37
1.3.3 Installation.	37

1.4	Base d'un programme	38
1.4.1	Des données et des instructions	38
1.4.2	Des bibliothèques de fonctions	39
1.5	Créer un projet	39
1.5.1	Créer un projet sous Code::Blocks	39
1.5.2	Créer un projet sous Visual Studio 2013	43
1.5.3	Créer un modèle de projet (template) sous Visual Studio.	47
1.6	Premier programme.	50
1.6.1	La fonction main() : entrée du programme	50
1.6.2	Afficher du texte avec la fonction printf()	51
1.6.3	Compiler et lancer le programme	52
1.7	Mise en pratique : découverte du compilateur	53
2.	Variables simples	54
2.1	Qu'est-ce qu'une variable en informatique ?	54
2.2	Définir des variables dans un programme	55
2.2.1	Les types de variables élémentaires en C	55
2.2.2	Déclarer ses variables dans un programme.	55
2.2.3	Contraintes pour le choix des noms	56
2.3	Mise en pratique : définir des variables dans un programme.	57
2.4	Manipulations de base sur les variables.	57
2.4.1	Affecter une valeur à une variable.	57
2.4.2	Des valeurs de type caractère (codage ASCII)	59
2.4.3	printf() pour afficher des valeurs	60
2.4.4	Obtenir des caractères accentués dans la console (sous Windows)	62
2.4.5	Obtenir et afficher la taille en mémoire d'une variable	63
2.4.6	Obtenir et afficher l'adresse mémoire d'une variable	64
2.4.7	scanf() pour récupérer une valeur entrée par l'utilisateur	64
2.4.8	Les pièges de scanf()	66
2.5	Mise en pratique : manipulations des variables	67
2.6	Les constantes	69
2.6.1	Définition, mot-clé const	69
2.6.2	Macro constantes #define.	69
2.6.3	Suite de constantes enum	70

2.7	Comprendre les variables	70
2.7.1	Codage et mesure de l'information	70
2.7.2	Plages de valeurs en décimal	71
2.7.3	Troncature	72
2.7.4	Codage des nombres négatifs	72
2.8	Mise en pratique : codage des informations numériques	73
2.9	Expérimentation : variables simples, déclaration, affectation, affichage, saisie	74
3.	Les opérations	75
3.1	La notion d'expression	75
3.2	Opérations arithmétiques	76
3.2.1	Les opérateurs +, -, *, /, %	76
3.2.2	Les affectations combinées	77
3.2.3	Opérateurs d'incrément et de décrémentation	78
3.2.4	Opérations entre types différents, opérateur de conversion (cast)	79
3.2.5	Priorités entre opérateurs	80
3.3	Mise en pratique : opérations arithmétiques, cast	83
3.4	Obtenir des valeurs aléatoires	86
3.4.1	Principe du pseudo aléatoire	86
3.4.2	La fonction rand()	87
3.4.3	La fonction srand()	88
3.4.4	Valeurs aléatoires dans des fourchettes	89
3.5	Mise en pratique : opérations et nombres aléatoires	91
3.6	Opérations bit à bit	91
3.6.1	ET : opérateur &	92
3.6.2	OU exclusif : opérateur ^	93
3.6.3	OU inclusif : opérateur 	93
3.6.4	Complément : opérateur ~	93
3.6.5	Décalages gauche et droite : opérateurs >> et <<	94
3.6.6	Priorités des opérateurs bit à bit	94
3.7	Mise en pratique : opérations bit à bit	94
3.8	Expérimentation : opérations arithmétiques, valeurs aléatoires	95

Chapitre 2

Les contrôles des blocs d'instructions

1. Blocs d'instructions et conditions	99
1.1 Qu'est-ce qu'un bloc d'instructions ?	99
1.1.1 Définition	99
1.1.2 Exemple	99
1.1.3 Utilité d'un bloc d'instructions	100
1.2 Définir une condition	101
1.2.1 Pourquoi une condition ?	101
1.2.2 Comment définir une condition ?	101
1.2.3 Les opérateurs de comparaison	101
1.2.4 L'opérateur unaire NON : !	102
1.2.5 Priorités des opérateurs NON et comparaison	103
1.3 Mise en pratique : opérateurs de comparaison et NON	104
2. Sauts conditionnels	104
2.1 L'instruction if	104
2.2 Le couple d'instructions if-else	106
2.3 La forme contractée du if-else, opérateur conditionnel ?	107
2.4 La cascade d'instructions if-else if-else	107
2.5 Expérimentation : les sauts conditionnels (les trois if)	108
2.6 Mise en pratique : les sauts conditionnels	111
3. Branchements	113
3.1 Branchement sélectif : switch, case et break	113
3.2 Rupture de séquence : goto avec étiquette	115
3.3 Expérimentation : branchement sélectif switch	116
3.4 Mise en pratique : l'aiguillage switch	118
4. Les tests multiconditions (ET/OU)	119
4.1 Conjonction ET : opérateur &&	119
4.1.1 ET avec deux expressions membres	119
4.1.2 ET avec plus de deux expressions membres	120
4.2 Disjonction OU, opérateur 	121
4.2.1 OU avec deux expressions membres	121
4.2.2 OU avec plus de deux expressions membres	122
4.3 ET prioritaire sur OU	122
4.4 Priorité avec les autres opérateurs	123
4.5 Mise en pratique : les opérateurs logiques ET, OU	123

5.	Boucles	126
5.1	Boucle TANT QUE : le while	126
5.2	Boucle FAIRE {...} TANT QUE : le do-while	127
5.3	Boucle comptée POUR : le for	128
5.4	Boucles imbriquées	129
5.5	Sortie et saut forcés dans une boucle	130
5.5.1	Sortir avec l'instruction break	130
5.5.2	Passer à l'itération suivante avec l'instruction continue	131
5.5.3	Sortir d'une ou de plusieurs boucles imbriquées avec l'instruction goto	131
5.6	Mise en pratique : les boucles while, do-while et for	132
6.	Utilisations typiques de boucles	135
6.1	Créer un menu utilisateur	135
6.2	Boucle d'événements dans un jeu vidéo	137
6.2.1	Récupérer les entrées clavier : kbhit() et getch()	137
6.2.2	Boucle d'événements simple	138
6.2.3	Contrôler le temps d'exécution	138
6.3	Commencer la création d'un jeu en mode console	139
6.4	Mise en pratique : menus, boucles d'événements	143
7.	Fonctions	145
7.1	Qu'est-ce qu'une fonction ?	145
7.2	Écrire sa fonction	146
7.2.1	Où écrire sa fonction ?	146
7.2.2	Conditions à remplir	147
7.2.3	Exemple de fonction sans retour ni paramètre	147
7.2.4	Exemple de fonction avec retour et sans paramètre	147
7.2.5	Exemple de fonction sans retour avec un paramètre	148
7.2.6	Exemple de fonction avec retour et paramètre	148
7.2.7	Conclusion : quatre cas d'écriture de fonction	149
7.3	Utiliser sa fonction	149
7.3.1	Appel de la fonction (bonjour1)	149
7.3.2	Récupération de la valeur de retour (bonjour2)	150
7.3.3	Passage de valeurs aux paramètres (bonjour 3 et 4)	150
7.3.4	Précision sur le passage par valeur	151
7.3.5	Visibilité et déclaration de la fonction	152

6 **Du C au C++**

De la programmation procédurale à l'objet

7.4	Fonctions avec liste variable de paramètres	153
7.4.1	Liste variable de paramètres de même type	154
7.4.2	Liste variable de paramètres de types différents	155
7.4.3	Transformer printf()	157
7.5	Fonctions récursives	158
7.6	Mise en pratique : fonctions.	160
7.6.1	Identifier les composants d'une fonction	160
7.6.2	Déclaration de fonctions	160
7.6.3	Procédures sans paramètre	161
7.6.4	Fonctions sans paramètre	161
7.6.5	Fonctions avec paramètres	162
8.	Gestion des variables	166
8.1	Visibilité des variables.	166
8.1.1	Profondeur de la déclaration	166
8.1.2	Portée des variables	167
8.1.3	Masquage d'une variable	167
8.2	Durée de vie des variables.	168
8.2.1	Variables globales	168
8.2.2	Variables locales (auto)	169
8.2.3	Variables static.	169
8.3	Choix méthodologiques	170
8.4	Mise en pratique : gestion des variables	171
8.5	Expérimentations : circuit de voiture	173
8.5.1	Principe du circuit	173
8.5.2	Structure de données du circuit	174
8.5.3	Structure de données de la voiture	174
8.5.4	Étapes de l'algorithme, fonctions nécessaires.	175
8.5.5	Programme commenté.	175
8.5.6	Conclusion	180
8.6	Mise en pratique : recherches algorithmiques.	180
8.6.1	Ascenseur de la tour infernale.	180
8.6.2	Tracer des lignes.	181
9.	Style, commentaires et indentations	181
9.1	Pourquoi soigner le style ?	181
9.2	Typographie et choix des noms	182
9.3	Indentations rigoureuses et accolades	183
9.4	Parenthèses pour dissiper les ambiguïtés.	185

9.5 Commentaires pertinents 185
9.6 Mise en pratique : style, indentations, commentaires 186

Chapitre 3
Les structures

1. Structure 189
1.1 Qu'est-ce qu'une structure ? 189
1.2 Disposer d'une structure dans un programme 190
1.2.1 Définir un type de structure 190
1.2.2 Où placer sa définition de structure 190
1.2.3 Déclarer ses variables structure. 191
1.2.4 Utiliser un typedef. 191
1.3 Utiliser une structure 193
1.3.1 Accès aux éléments avec l'opérateur point. 193
1.3.2 Priorité de l'opérateur point 194
1.3.3 Une structure comme champ dans une structure. 194
1.3.4 Initialiser une structure à la déclaration. 195
1.3.5 Copier une structure 197
1.4 Mise en pratique : définir, déclarer, initialiser des structures 197
2. Structures et fonctions 198
2.1 Retourner une structure. 198
2.2 Structures en paramètre de fonction. 199
2.3 Mise en pratique : structures et fonctions 201
2.4 Expérimentation 202
2.4.1 Fourmi 1 : une fourmi mobile à l'écran 203
2.4.2 Voiture 2 : structure circuit, structure voiture 206

Chapitre 4
Les tableaux

1. Tableaux statiques. 215
1.1 Qu'est-ce qu'un tableau ? 215
1.2 Disposer d'un tableau statique dans un programme 216
1.2.1 Définir et déclarer un tableau 216
1.2.2 Utiliser des #define pour les tailles 216

1.3	Utiliser un tableau	217
1.3.1	Accéder aux éléments du tableau avec l'opérateur []	217
1.3.2	Priorité de l'opérateur []	217
1.3.3	Débordement de tableau	218
1.3.4	Initialiser un tableau à la déclaration	218
1.3.5	Parcourir un tableau avec une boucle for	218
1.3.6	Trier un tableau	219
1.4	Précisions sur enum et #define	220
1.4.1	Utiliser un #define	220
1.4.2	Utiliser un enum	222
1.5	Tableaux à plusieurs dimensions	223
1.5.1	Déclarer un tableau à plusieurs dimensions	223
1.5.2	Initialiser la déclaration	225
1.5.3	Parcourir un tableau à plusieurs dimensions	225
1.6	Expérimentation : tableaux statiques	226
1.7	Mise en pratique : opérations de base sur les tableaux statiques (non dynamiques)	230
1.7.1	Déclaration de tableaux, accès aux éléments	230
1.7.2	Initialisation de tableaux à la déclaration	231
1.7.3	Tableaux à plusieurs dimensions	231
1.7.4	Boucles et tableaux	232
2.	Exemples d'utilisations de tableaux	234
2.1	Chaînes de caractères	234
2.2	Image bitmap	236
2.3	Stocker des données localisées	236
2.4	Expérimentation : utilisation de chaînes de caractères	237
2.5	Mise en pratique : tableaux	239
2.5.1	Chaînes de caractères	239
2.5.2	Image, terrain de jeu	241
2.5.3	Localisation de données via plusieurs dimensions	243
3.	Tableaux et structures	243
3.1	Tableau comme champ dans une structure	243
3.2	Tableau de structures	244
3.3	Différences entre tableaux et structures	247
3.4	Mise en pratique : tableaux de structures	247

3.5	Expérimentations : amélioration du programme voiture 2	251
3.5.1	Voiture 3 : une structure trait et un tableau de traits	251
3.5.2	Voiture 4 : tableau de structures voiture	258
3.5.3	Voiture 5 : un circuit par voiture	260
3.5.4	Voiture 5b : un circuit différent pour chaque voiture	265
4.	Tableaux et fonctions	272
4.1	Utiliser un tableau déclaré en global	272
4.2	Tableau en paramètre de fonction	274
4.2.1	Précision sur le type tableau	274
4.2.2	La variable pointeur	274
4.2.3	En paramètre, conversion du tableau en pointeur	275
4.2.4	Choix pour l'écriture des tableaux en paramètre	277
4.2.5	Modification des données via un passage par adresse	278
4.3	Quelques fonctions de traitement de chaînes de caractères	280
4.3.1	Récupérer une chaîne entrée par l'utilisateur	280
4.3.2	Obtenir la taille d'une chaîne	281
4.3.3	Copier une chaîne	281
4.3.4	Comparer deux chaînes	282
4.3.5	Concaténer deux chaînes	282
4.4	Expérimentation : tableaux et fonctions	283
4.5	Mise en pratique : tableaux et fonctions	288
4.5.1	Appels de fonctions, tableaux en paramètre	288
4.5.2	Manipulations sur les chaînes	291
4.6	Fourmi 2 : une colonie de fourmis	293
4.7	Fourmi 3 : plusieurs colonies de fourmis	296

Chapitre 5 Structuration d'un programme

1.	Structuration d'un programme, étude d'un automate cellulaire	301
1.1	Clarifier et définir ses objectifs	301
1.1.1	Principe de l'automate cellulaire	302
1.1.2	Fonctionnement envisagé	302
1.2	Trouver une structure de données valable	304
1.3	Identifier les fonctions principales	304
1.4	Choisir le niveau des variables fondamentales	305

1.5	Écrire les fonctions	306
1.5.1	Fonction d'initialisation.	306
1.5.2	Fonction d'affichage.	307
1.5.3	Fonction de calcul	308
1.5.4	Fonction de comptage des voisins.	309
1.5.5	Fonction de recopie	309
1.5.6	Montage dans le main().	310
1.6	Intégrer une librairie personnelle	311
1.7	Répartir son code sur plusieurs fichiers C	313
1.7.1	Code réparti en quatre fichiers C	314
1.7.2	Problème de redéfinition	317
1.8	Mise en pratique : structuration d'un programme	320
1.8.1	Simulation d'un feu de forêt	320
1.8.2	Tristus et rigolus	320
1.8.3	Simulation d'une attaque de microbes dans le sang	320
1.8.4	Bancs de poissons, mouvements de populations	320
1.8.5	Élection présidentielle	321
1.8.6	Chenille.	321
1.8.7	Système de vie artificielle, colonies de fourmis	321
1.8.8	Boutons et pages	322
1.8.9	Panneaux de bois et entrepôts.	323
1.8.10	Nenuphs et clans	323
1.8.11	Neige 1	324
1.8.12	Neige 2	324
1.8.13	Neige 3	324
1.8.14	Casse-brique simple	324
1.8.15	Casse-brique guru.	325
1.8.16	Space Invaders simple	325
1.8.17	Space Invaders more	325
1.8.18	Space Invaders guru	325
1.8.19	Pacman débutant	325
1.8.20	Pacman intermédiaire	325
1.8.21	Pacman guru	325
1.8.22	Jeu de miroirs	326
1.8.23	Simulations football.	326

Chapitre 6
Les pointeurs

- 1. Principe du pointeur 327
 - 1.1 Qu'est-ce qu'un pointeur ?..... 327
 - 1.1.1 Mémoire RAM..... 327
 - 1.1.2 Une variable pointeur 328
 - 1.1.3 Quatre opérateurs 328
 - 1.1.4 Trois utilisations fondamentales des pointeurs..... 329
 - 1.2 Déclarer un pointeur dans un programme 329
 - 1.3 Fonctionnement des quatre opérateurs..... 330
 - 1.3.1 Opérateur adresse : & 330
 - 1.3.2 Opérateur étoile : * 331
 - 1.3.3 Opérateur flèche : ->..... 332
 - 1.3.4 Opérateur crochet : []..... 333
 - 1.3.5 Priorité des quatre opérateurs 334
 - 1.4 Allouer dynamiquement de la mémoire 334
 - 1.4.1 La fonction malloc() 334
 - 1.4.2 Libérer la mémoire allouée : la fonction free()..... 335
 - 1.4.3 Le pointeur générique void* 336
 - 1.4.4 La valeur NULL 337
 - 1.5 Attention à la validité d'une adresse mémoire 338
 - 1.5.1 Validité d'une adresse mémoire 338
 - 1.5.2 Pourquoi caster le retour des fonctions d'allocation ?..... 340
 - 1.6 Pointeurs et constantes 341
 - 1.6.1 Pointeur variable sur un objet constant..... 341
 - 1.6.2 Pointeur constant sur un objet variable..... 342
 - 1.6.3 Pointeur constant sur un objet constant 343
 - 1.7 Cas des tableaux de pointeurs 344
 - 1.7.1 Une structure de données très utile 344
 - 1.7.2 Un tableau de chaînes de caractères 345
 - 1.7.3 Utiliser les arguments de lignes de commandes 346
 - 1.8 Cas des pointeurs de fonction 348
 - 1.8.1 Une fonction est une adresse 348
 - 1.8.2 Reconnaître un type de fonction 349
 - 1.8.3 Appeler une fonction via un pointeur 350
 - 1.8.4 Pourquoi des pointeurs de fonction ?..... 351

1.9	Expérimentation : base pointeurs	352
1.10	Mise en pratique : base sur les pointeurs	355
1.10.1	Définir et manipuler des pointeurs	355
1.10.2	Tests tableaux/pointeurs	356
1.10.3	Base allocation dynamique	358
1.10.4	Attention aux erreurs	359
1.10.5	Tableaux de chaînes	360
2.	Allocation dynamique de tableaux	361
2.1	Allouer un tableau avec un pointeur	361
2.2	Allouer une matrice avec un pointeur de pointeur	362
2.3	Différences entre tableaux statiques et dynamiques	364
2.4	Autres fonctions d'allocation dynamique	365
2.4.1	Fonction calloc()	365
2.4.2	Fonction realloc()	366
2.5	Mise en pratique : allocation dynamique	367
2.5.1	Allouer dynamiquement des tableaux	367
2.5.2	Allouer dynamiquement des matrices	369
2.5.3	Allocation dynamique calloc() et realloc()	370
3.	Pointeurs en paramètres de fonction	371
3.1	Passage par référence	371
3.1.1	Cas général d'une variable quelconque	371
3.1.2	Exemple : une fonction qui retourne l'heure	373
3.1.3	Passage par référence d'une structure	374
3.1.4	Passage par référence d'une variable pointeur	375
3.2	Tableaux dynamiques en paramètres	379
3.3	Mise en pratique : passage par référence	381
3.3.1	Passage par référence, base	381
3.3.2	Passage par référence, opérateurs bit à bit	383
3.3.3	Passage de pointeurs par référence	384
3.3.4	Passage de tableaux dynamiques	385
4.	Fichiers (type FILE*)	386
4.1	Notions de base	386
4.1.1	Le type FILE*	386
4.1.2	Ouverture et fermeture d'un fichier	387
4.1.3	Spécifier un chemin d'accès	388

- 4.2 Fichiers binaires. 389
 - 4.2.1 Écriture et lecture en mode binaire. 389
 - 4.2.2 Détecter la fin d'un fichier binaire 391
 - 4.2.3 Déplacements dans un fichier. 391
- 4.3 Écriture et lecture en mode texte. 392
 - 4.3.1 Détecter la fin d'un fichier : EOF et feof() 392
 - 4.3.2 Lecture/écriture de caractères 392
 - 4.3.3 Lecture/écriture de chaînes 394
 - 4.3.4 Lecture/écriture formatées 395
- 4.4 Sauvegarde d'éléments dynamiques 396
 - 4.4.1 Sauvegarder et récupérer un tableau dynamique. 396
 - 4.4.2 Récupérer des données via des pointeurs. 398
- 4.5 Expérimentation : récapitulation sauvegardes binaires 399
- 4.6 Mise en pratique : fichiers 404

Chapitre 7

La dimension objet, le C++

- 1. C inclus en C++ 409
 - 1.1 Un projet console en C++ 410
 - 1.1.1 Bibliothèque <iostream> 410
 - 1.1.2 L'instruction using namespace std; 411
 - 1.2 Un programme C compilé en C++ 411
- 2. C augmenté en C++ 416
 - 2.1 Entrée-sortie console : cout et cin 416
 - 2.1.1 Utiliser cout et cin 416
 - 2.1.2 Instructions de formatage en sortie 417
 - 2.2 Variables et constantes. 420
 - 2.2.1 Déclarations plus souples 420
 - 2.2.2 Type bool 420
 - 2.2.3 Type long long ou unsigned long long 420
 - 2.2.4 Type caractère unicode : wchar_t. 421
 - 2.2.5 Typedef inutile pour les structures. 422
 - 2.2.6 Type référence (pointeur masqué) 422
 - 2.2.7 Type pointeur, opérateurs new et delete 426
 - 2.2.8 Type auto (C++ 2011). 427

2.2.9	Constantes (const) et énumération (enum) plutôt que #define	427
2.3	Conversions de types	428
2.3.1	static_cast<type>	428
2.3.2	const_cast<type>	429
2.3.3	reinterpret_cast<type>	429
2.3.4	dynamic_cast<type>	430
2.4	Boucle for (:) "pour chaque"	430
2.5	Fonctions	431
2.5.1	Fonctions embarquées "inline"	431
2.5.2	Retourner une référence	431
2.5.3	Valeurs par défaut de paramètres	433
2.5.4	Surcharge des fonctions	435
2.5.5	Fonctions génériques (template).	436
2.5.6	Fonctions comme champs de structures	438
2.6	Gestion des exceptions (base)	439
2.6.1	Instruction throw	439
2.6.2	Instruction de saut try-catch.	440
2.6.3	Instruction throw et appels de fonctions.	441
2.6.4	Instruction throw sans valeur de retour.	443
2.6.5	Exception non identifiée	443
2.6.6	Bloc catch(...) par défaut	444
2.7	Espaces de noms (namespace) et raccourcis (using)	445
2.8	Intégrer d'autres langages dans le code C++	450
3.	Classes, objets	451
3.1	Une classe, des objets	451
3.1.1	Qu'est-ce qu'une classe ?	451
3.1.2	Qu'est-ce qu'un objet ?	452
3.1.3	Définir une classe	452
3.1.4	Déclarer un objet	453
3.1.5	Droits d'accès	454
3.1.6	Un programme C muté en classe et objet	454
3.2	Constructeur	459
3.2.1	Paramétrer un objet à sa déclaration	459
3.2.2	Le pointeur this	461
3.2.3	Constructeur sans paramètre	462
3.2.4	Constructeurs avec paramètres.	463

3.2.5	Constructeur et initialiseur	464
3.2.6	Constructeur et copie d'objet	470
3.3	Destructeur	474
3.4	Propriétés encapsulées « property »	477
3.4.1	Principe de la « property »	477
3.4.2	Lire une variable private ou protected	477
3.4.3	Modifier une variable private ou protected	478
3.4.4	Lire et modifier une variable private ou protected	479
3.4.5	Intérêt d'un appel de fonction.	480
3.5	Surcharge des opérateurs	481
3.5.1	Fonction operator globale hors classe.	482
3.5.2	Fonction operator localisée dans une classe.	483
3.5.3	Fonction operator et données dynamiques	486
3.5.4	Objet-fonction (ou fonction-objet)	490
3.6	Classes et membres "static"	491
3.6.1	Qualificatif static en C	491
3.6.2	Qualificatif static et objets	492
3.7	Classes génériques (template ou patron ou modèle)	494
3.7.1	Principe.	494
3.7.2	Syntaxe de base	494
3.7.3	Syntaxe des constructeurs	495
3.7.4	Syntaxe avec plusieurs types génériques	495
3.7.5	Paramétrage d'une classe générique	496
3.7.6	Exemple d'implémentation d'une pile générique.	501
3.7.7	Spécialisation de fonction sur un type donné	503
3.7.8	Spécialiser une classe entière.	504
3.8	Classes et fonctions "amies" (friend)	507
4.	Associations entre objets.	509
4.1	Principes des associations pour les relations entre objets	509
4.1.1	Association simple.	509
4.1.2	Agrégation	509
4.1.3	Composition	510
4.1.4	Problème syntaxique en C++	510
4.2	Associations simples : messages en paramètres de méthodes	511
4.2.1	Liaison non réciproque entre deux objets	511
4.2.2	Liaison réciproque entre deux objets	513

4.3	Agrégations : pointeur d'objet en propriété	516
4.3.1	Liaison à sens unique	516
4.3.2	Partage des objets pointés	518
4.3.3	Liaison réciproque	524
4.4	Composition : interdire ou limiter le partage et la recréation d'objets internes	536
4.4.1	De l'agrégation à la composition, comment trancher ?	536
4.4.2	Pointeur d'objet encapsulé en propriété	537
4.4.3	Objet en propriété	540
4.4.4	Référence d'objet en propriété	545
5.	Héritage	550
5.1	Définir une classe dérivée	551
5.2	Appeler explicitement un constructeur de la classe de base	553
5.3	Redéfinition de données ou de fonctions	553
5.4	Spécifier un membre de la classe de base	555
5.5	Droits d'accès locaux de la classe héritée	555
5.6	Droits d'accès globaux de la classe héritée	558
5.7	Héritage multiple	561
5.8	Héritage multiple avec une base virtuelle	565
5.9	Distinction entre héritage et association	569
6.	Polymorphisme, virtualité	570
6.1	Accès pointeurs par défaut aux fonctions redéfinies	570
6.2	Accès pointeur aux fonctions virtuelles redéfinies	571
6.3	Intérêt des accès pointeurs aux fonctions virtuelles	573
6.4	Classes abstraites, fonctions virtuelles pures	575
6.5	Interface	580

Chapitre 8

Récurtivité

1.	Fonctions récursives	583
1.1	Qu'est-ce que la récursivité ?	583
1.2	Une fonction récursive basique	584
1.3	Pile d'appels et débordement	586
1.4	Retourner une valeur	587
1.5	Représentation et analyse du fonctionnement	589

1.5.1	Analyse descendante	590
1.5.2	Analyse ascendante	591
1.6	Choisir entre itératif ou récursif	591
2.	Exemples classiques de fonctions récursives	592
2.1	Calculs	592
2.1.1	Afficher les chiffres d'un entier	592
2.1.2	Produit factoriel	593
2.1.3	Suite de Fibonacci	593
2.1.4	Changement de base arithmétique d'un nombre	595
2.1.5	Puissance	596
2.1.6	PGCD, algorithme d'Euclide	597
2.2	Dessins	600
2.2.1	Tracé d'une règle graduée : "diviser pour résoudre"	600
2.2.2	Tracé de cercles	603
2.2.3	Tracé de carrés	604
2.2.4	Tracé d'un arbre	605
2.3	Créations et jeux	608
2.3.1	Trouver un chemin dans un labyrinthe	608
2.3.2	Création d'un labyrinthe	612
2.4	Les tours de Hanoï	614
2.5	Tri rapide d'un tableau de nombres	616
3.	Mise en pratique : récursivité	619

Chapitre 9

Listes en C

1.	Listes chaînées dynamiques	625
1.1	Qu'est-ce qu'une liste chaînée ?	625
1.1.1	Une chaîne constituée de maillons	625
1.1.2	Trois types de listes chaînées	626
1.1.3	Les actions sur une liste chaînée	627
1.1.4	Listes chaînées contre tableaux	628
1.2	Implémenter une liste simple	628
1.2.1	Structure de données d'un maillon	629
1.2.2	Début et fin de la liste	629
1.2.3	Initialiser un maillon	630

1.2.4	Ajouter un maillon au début d'une liste	630
1.2.5	Insérer un maillon dans une liste	632
1.2.6	Parcourir la liste	636
1.2.7	Supprimer un maillon au début de la liste	637
1.2.8	Supprimer un élément sur critère	638
1.2.9	Détruire la liste.	640
1.2.10	Sauvegarder la liste.	641
1.3	Implémenter une liste simple circulaire.	643
1.3.1	Structure de données d'une liste circulaire	643
1.3.2	Liste vide.	643
1.3.3	Début et fin de la liste	643
1.3.4	Initialiser un maillon	644
1.3.5	Ajouter un maillon.	644
1.3.6	Parcourir la liste	644
1.3.7	Supprimer un maillon	645
1.3.8	Détruire la liste.	646
1.4	Implémenter une liste symétrique.	646
1.4.1	Structure de données	647
1.4.2	Liste vide.	647
1.4.3	Début et fin de la liste	647
1.4.4	Initialiser un élément.	647
1.4.5	Ajouter un élément au début	648
1.4.6	Ajouter un élément à la fin	648
1.4.7	Parcourir et afficher la liste	649
1.4.8	Supprimer un élément	649
1.4.9	Détruire la liste.	650
1.4.10	Copier une liste	650
1.5	Mise en pratique : listes chaînées.	651
2.	Piles	654
2.1	Principes de la pile.	654
2.1.1	Modèle de données pile	654
2.1.2	Implémentation statique ou dynamique	654
2.1.3	Primitives de gestion des piles.	655
2.1.4	Applications importantes des piles	656
2.2	Implémentation d'une pile en dynamique	656
2.2.1	Structure de données	656
2.2.2	Pile vide, pile pleine	656

2.2.3	Initialisation	657
2.2.4	Empiler	657
2.2.5	Lire le sommet	657
2.2.6	Dépiler	658
2.2.7	Vider, détruire	658
2.2.8	Affichage	658
2.2.9	Test dans le main()	659
2.3	Implémentation d'une pile en statique (tableau)	660
2.3.1	Structure de données	660
2.3.2	Initialisation	660
2.3.3	Pile vide, pile pleine	661
2.3.4	Empiler	661
2.3.5	Lire le sommet	662
2.3.6	Dépiler	662
2.3.7	Vider, détruire	662
2.3.8	Affichage	663
2.3.9	Test dans le main()	664
2.4	Mise en pratique : les piles	665
3.	Files	667
3.1	Principes de la file	667
3.1.1	Modèle de données file	667
3.1.2	Implémentation statique ou dynamique	667
3.1.3	Primitives de gestion des files	668
3.1.4	Applications importantes des files	669
3.2	Implémentation d'une file en dynamique	669
3.2.1	Structure de données	669
3.2.2	File vide, file pleine	669
3.2.3	Initialisation	670
3.2.4	Enfiler	670
3.2.5	Lire la tête, lire la queue	671
3.2.6	Défiler	671
3.2.7	Vider, détruire	672
3.2.8	Affichage	672
3.2.9	Test dans le main()	673
3.3	Implémentation d'une file en statique (tableau)	674
3.3.1	Structure de données	674
3.3.2	File vide, file pleine	675

3.3.3	Initialisation	675
3.3.4	Enfiler	676
3.3.5	Lire la tête, lire la queue.	676
3.3.6	Défiler	677
3.3.7	Vider, détruire	677
3.3.8	Affichage.	678
3.3.9	Test dans le main()	678
3.4	Mise en pratique : les files	679
4.	Arbres	682
4.1	Généralités sur les arbres	682
4.1.1	Principe	682
4.1.2	Exemples d'utilisation des arbres	683
4.1.3	Nomenclature des arbres	685
4.2	Types d'arbre	686
4.2.1	Arbre binaire.	686
4.2.2	Arbre n-aire.	686
4.2.3	Transformer un arbre n-aire en arbre binaire.	686
4.3	Représentations en mémoire	687
4.3.1	Arbre n-aire.	687
4.3.2	Arbre binaire.	689
4.3.3	Structures de données statiques ou dynamiques.	691
5.	Contrôler un arbre binaire.	691
5.1	Créer un arbre binaire.	691
5.1.1	Créer un arbre à partir d'un schéma descriptif.	691
5.1.2	Créer un arbre à partir des données aléatoires d'un tableau	694
5.1.3	Créer un arbre en insérant des éléments ordonnés	696
5.2	Parcourir l'arbre	697
5.2.1	Parcours en profondeur	697
5.2.2	Parcours en largeur, par niveau	701
5.3	Afficher l'arbre.	702
5.3.1	Afficher un arbre avec indentation	702
5.3.2	Dessiner l'arbre sans les liens.	703
5.4	Obtenir des propriétés de l'arbre binaire	704
5.4.1	Connaître la taille.	704
5.4.2	Connaître la hauteur	705
5.4.3	Savoir si un nœud est une feuille	705
5.4.4	Compter le nombre des feuilles de l'arbre.	706

5.4.5	Lister toutes les feuilles	706
5.4.6	Faire la somme des valeurs des nœuds	707
5.4.7	Comparer des valeurs des nœuds de l'arbre	708
5.4.8	Ramener un nœud de l'arbre à partir d'une valeur	708
5.5	Dupliquer l'arbre	709
5.6	Détruire l'arbre	710
5.7	Conversion statique-dynamique d'un arbre binaire	710
5.7.1	Conversion d'un arbre statique en dynamique	710
5.7.2	Conversion d'un arbre dynamique en statique	711
5.8	Sauvegarde et chargement d'un arbre binaire	712
5.8.1	Sauvegarder un arbre dynamique	712
5.8.2	Charger (load) un arbre dynamique	712
5.9	Arbres binaires sur fichiers	713
5.9.1	Structure de données	713
5.9.2	Lecture d'un nœud à partir de son numéro d'enregistrement	713
5.9.3	Adaptation des fonctions pour les arbres binaires dynamiques ou statiques	714
5.10	Mise en pratique : arbre binaire	714
6.	Arbres binaires de recherche	719
6.1	Définition	719
6.2	Structure de données	720
6.3	Insérer un élément dans l'arbre selon sa clé	720
6.3.1	Comparer deux clés	721
6.3.2	Insérer un élément à la bonne place	721
6.4	Rechercher dans l'arbre un élément selon sa clé	722
6.5	Supprimer un élément dans l'arbre de recherche	723
6.5.1	Trois cas	724
6.5.2	Fonction de recherche du nœud max	724
6.5.3	Fonction de suppression	725
6.6	Lister tous les éléments de l'arbre (parcours en largeur)	726
6.7	Afficher l'arbre	728
6.8	Test dans le main()	729
6.9	Mise en pratique : arbres	730

Chapitre 10

Listes en C++ (conteneurs)

1.	Principes des conteneurs	731
1.1	Trois catégories de conteneurs	731
1.1.1	Conteneurs séquentiels	731
1.1.2	Conteneurs associatifs	732
1.1.3	Les « presque conteneurs »	732
1.2	Récapitulation des bibliothèques conteneurs	732
1.3	Opérations sur les conteneurs	735
1.3.1	Templates, constructeurs, destructeurs	735
1.3.2	Accès aux éléments	736
1.3.3	Itérateurs	737
1.3.4	Pointeurs	738
1.3.5	Affectation	738
1.3.6	Opérations de liste, pile et file	738
1.3.7	Gestion des tailles et capacités	739
1.3.8	Fonctions de comparaison	740
2.	Conteneurs séquentiels	740
2.1	La classe array	740
2.1.1	Template, constructeurs, destructeurs	740
2.1.2	Accès éléments, itérateurs	740
2.1.3	Opérations de liste	743
2.1.4	Taille et capacité	744
2.1.5	Comparaisons	744
2.2	La classe vector	744
2.2.1	Template, constructeurs, destructeurs	744
2.2.2	Accès aux éléments, itérateurs	745
2.2.3	Opérations de liste	748
2.2.4	Opérations de pile et file	752
2.2.5	Taille et capacité	753
2.2.6	Comparaisons entre vecteurs	757
2.2.7	Vecteur spécialisé de booléens	758
2.3	La classe list<>	761
2.3.1	Template, constructeurs, destructeurs	761
2.3.2	Accès aux éléments, itérateurs	762
2.3.3	Opérations de liste (pile et file compris)	764

2.3.4	Comparaisons entre listes	774
2.3.5	Taille et capacité	775
2.4	La classe deque	776
2.4.1	Template, constructeurs, destructeurs.	776
2.4.2	Accès aux éléments, itérateurs	777
2.4.3	Opérations de liste.	779
2.4.4	Taille et capacité	783
2.4.5	Comparaisons entre files	785
3.	Conteneurs séquentiels spécialisés	786
3.1	La classe stack	786
3.1.1	Templates, constructeurs	786
3.1.2	Méthodes	787
3.1.3	Comparaisons	788
3.2	La classe queue	789
3.2.1	Template, constructeurs	789
3.2.2	Méthodes	790
3.2.3	Comparaisons	791
3.3	La classe priority_queue	792
3.3.1	Template, constructeurs	792
3.3.2	Méthodes	796
3.3.3	Fonctions-objets de comparaison	797
4.	Conteneurs associatifs.	799
4.1	La classe map.	800
4.1.1	Templates, constructeurs	800
4.1.2	Itérateurs, accès aux éléments	803
4.1.3	Opérations de liste.	808
4.1.4	Taille et capacité	812
4.1.5	Comparaisons	813
4.2	La classe multimap	814
4.2.1	Templates, constructeurs	814
4.2.2	Itérateurs, accès aux éléments	817
4.2.3	Opérations de liste.	822
4.2.4	Taille et capacité	827
4.2.5	Comparaisons	828
4.3	La classe set	828
4.3.1	Template, constructeurs	828
4.3.2	Itérateurs, accès aux éléments	831

4.3.3	Opérations de liste	834
4.3.4	Taille et capacité	839
4.3.5	Comparaisons.	840
4.4	La classe multiset	840
4.4.1	Template, constructeurs	841
4.4.2	Itérateurs, accès aux éléments.	844
4.4.3	Opérations de liste	846
4.4.4	Taille et capacité	850
4.4.5	Comparaisons.	851
4.5	La classe pair	852
4.5.1	Template, constructeurs	852
4.5.2	Méthodes	853
5.	Conteneurs « presque conteneurs »	855
5.1	La classe string.	855
5.1.1	Template.	855
5.1.2	Constructeurs.	856
5.1.3	Accès, itérateurs	857
5.1.4	Opérations sur chaînes de caractères	859
5.1.5	Taille et capacités.	879
5.1.6	Comparaisons.	882
5.2	La classe bitset.	883
5.2.1	Template, constructeurs	883
5.2.2	Méthodes	884
5.2.3	Opérateurs bits à bits, comparaison	887

Annexe

Priorité et associativité des opérateurs	889
--	-----

Index.	891
----------------	-----

Chapitre 2

Les contrôles des blocs d'instructions

1. Blocs d'instructions et conditions

1.1 Qu'est-ce qu'un bloc d'instructions ?

1.1.1 Définition

- Un bloc d'instructions est UNE instruction composée de plusieurs instructions qui se suivent.
- En C (et tous les langages dérivés) il est délimité avec les opérateurs { } (accolades ouvrante et fermante).
- Un bloc peut contenir d'autres blocs imbriqués.
- Dans un fichier source, il ne peut pas y avoir d'instruction en dehors d'un bloc (sauf directives macro-processeur et déclarations de variables globales ou de fonctions). Pour être valides, toutes les instructions doivent être localisées dans un bloc. Le bloc au sommet est celui de la fonction main() qui réunit in fine toutes les instructions du programme.

1.1.2 Exemple

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

int main()
{ //----- ouv bloc main B1
  int x, pasx; // déclarations de variables locales au bloc main
```

```

// elles sont visibles (accessibles) dans ce bloc
// et tous les sous-blocs

{ //-----ouv B2
  int c;
  x=0;
  c=rand()%256;
  pasx=rand()%5;
} // -----ferm B2
x=640;

{ // -----ouv B3
  //c=10; // provoque erreur, c non visible dans ce bloc
  x/=2;
  pasx=15;

} // -----ferm B3
x+=pasx;
printf("x vaut : %d\n",x); // affichage ?

Return 0;

} //-----ferm bloc main

```

Les blocs posent la question de la visibilité des variables. En fait la durée de vie des variables en générale est associée au bloc dans lequel elles sont déclarées. C'est pourquoi elles sont visibles dans tous les sous-blocs imbriqués et invisibles en revanche dans les blocs de niveau supérieur ou de même niveau mais séparés.

1.1.3 Utilité d'un bloc d'instructions

Dans l'exemple ci-dessus les blocs sont inutiles : ils n'influencent en rien sur le déroulement linéaire du programme, il n'y a aucune modification de la succession des opérations et les supprimer revient au même sauf pour la déclaration de la variable `c`.

L'utilité des blocs est de permettre de rompre avec cette linéarité et d'introduire grâce à des instructions données par le langage :

- Des sauts de bloc, ce sont les trois instructions `if`, `if-else`, `if-else if-else`.
- Des branchements, c'est l'instruction `switch`.
- Et des répétitions, ce sont les trois instructions de boucles `while`, `do-while` et `for`.

Les blocs permettent également de généraliser des portions de code qui se répètent dans un programme avec éventuellement des valeurs différentes. C'est ce que nous appelons une fonction. Une fonction est un bloc d'instructions doté d'une entrée de valeur et d'une sortie de valeur.

1.2 Définir une condition

1.2.1 Pourquoi une condition ?

Les sauts conditionnels permettent d'exécuter ou de ne pas exécuter un bloc d'instructions selon qu'une condition est remplie ou non. De même pour les boucles, un bloc d'instructions est répété tant qu'une condition fixée pour sa répétition reste vraie.

Par exemple pour l'instruction `if`, première forme du saut conditionnel, l'exécution du bloc est soumise à une condition de la façon suivante :

```
Si (condition vraie) Alors
    faire bloc d'instructions
FinSi
```

Si et seulement si la condition est remplie, c'est-à-dire est "vraie", alors les instructions du bloc qui en dépend sont exécutées.

■ Remarque

Une condition vraie, c'est une expression dont la valeur est différente de 0. À l'inverse une condition est considérée comme fausse si l'expression vaut 0.

1.2.2 Comment définir une condition ?

Définir une condition, c'est écrire une expression qui sera évaluée comme vraie ou fausse. La plupart des conditions sont élaborées à partir de comparaisons de variables. Les comparaisons bits à bits sont parfois utilisées également.

1.2.3 Les opérateurs de comparaison

Il est possible de comparer entre elles des variables ou n'importe quelles expressions du point de vue de leurs valeurs. Soient deux expressions `a` et `b` (des variables ou des opérations), elles ont chacune une valeur et les comparaisons suivantes sont possibles :

<code>a > b</code>	a strictement supérieur à b
<code>a < b</code>	a strictement inférieur à b
<code>a >= b</code>	a supérieur ou égal à b
<code>a <= b</code>	a inférieur ou égal à b
<code>a == b</code>	a égal b (test d'égalité)
<code>a != b</code>	a différent de b (test d'inégalité)

Le résultat de chaque expression vaut :

0 si c'est faux
1 si c'est vrai

À votre avis, qu'imprime le programme suivant ?

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
int a=10, b=5;
printf("%d ", a < b);
printf("%d ", a+b > 12);

a = 5;
printf("%d ", a >= b);
printf("%d ", a == b);

b = 4;
printf("%d ", a <= b);
printf("%d ", a == b);
printf("%d ", a+3 != b*2);
return 0;
}

```

Réponse :

0 1 1 1 0 0 0

Autre exemple :

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
int a,b;
printf("%d ", 56 < rand() ); // 1 si vrai, 0 si faux

printf("%d ", (a=rand()) + (b=rand()) >= 1000 );
// 1 si vrai, 0 si faux

printf("%d ", a*b == 45); // 1 si oui, 0 si non
return 0;
}

```

Les résultats dépendent des retours de la fonction rand().

1.2.4 L'opérateur unaire NON : !

L'opérateur ! placé à gauche d'une expression donne vrai si l'expression vaut 0 (faux) et faux si l'expression est différente de 0 (vraie). Par exemple :

```

#include <stdio.h>
#include <stdlib.h>

```

```
int main()
{
    int a=10, b=20;
    printf("%d ", !(a < b)); // imprime 0
    printf("%d ", !(a > b)); // imprime 1
    return 0;
}
```

Il est souvent utilisé dans des conditions à la place d'une égalité à 0, par exemple :

```
int a=10;

printf("%d ", a==0); // imprime 0

// et peut être remplacé par l'expression :

printf("%d ", !a); // qui imprime également 0
```

1.2.5 Priorités des opérateurs NON et comparaison

Toutes les opérations arithmétiques ainsi que décalages, conversion de type, sizeof, NON et complément à 1 sont prioritaires par rapport aux comparaisons. En revanche les comparaisons sont prioritaires sur ET, OU inclusif et exclusif et affectations combinées (Annexe Priorité et associativité des opérateurs).

Par exemple :

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

int main()
{
    int a=10, b=0, c=2;
    printf("a+b < c*10 vaut %d\n", a+b < c*10);
    printf("c*10 >= 66 vaut %d\n", c*10 >= 66);
    printf("a+b < c*10 >= 66 == 80 vaut %d\n",
           a+b < c*10 >= 66 == 80);
    return 0;
}
```

La première ligne donne 1

la seconde 0

la troisième 0

1.3 Mise en pratique : opérateurs de comparaison et NON

Exercice 1

Faire un programme qui affiche les résultats de ce qui suit :

```
int a, b, c;
    srand(5);
    Imprimez les résultats de :
    a=rand()%256;
    b=rand()%256;
    c=rand()%256;

    a<128
    b>128
    c==223
    a < b >= rand()%2 == 1
    a= b !=c +rand()%50;
    b= a==c
    c= rand()%10 < rand()%10 >= rand()%10 != rand()%10 ==rand()%10
```

Exercice 2

Qu'imprime le programme suivant ?

```
#include <stdio.h>

int main()
{
    int a=10, b=0, c=2;
    printf("a+b < c*10 vaut %d\n", a+b < c*10);
    printf("c*10 >= 15 vaut %d\n", c*10 >= 15);
    printf("a+b < c*10 >= 15 == 1 vaut %d\n",
           a+b < c*10 >= 15 == 1);
    printf("a+b < c*10 <= 15 == 1 vaut %d\n",
           a+b < c*10 <= 15 == 1);
    return 0;
}
```

2. Sauts conditionnels

2.1 L'instruction if

SI et seulement si l'expression est vraie ALORS le bloc des instructions associées au if est exécuté. Il ne peut y avoir qu'un seul bloc associé au if.

```
if ( expression1 vraie){
```