Alain Gibaud

Programmer en C++

Des premiers pas à la maîtrise de C++20



ellipses

Références sciences

Programmer en C++

Des premiers pas à la maîtrise de C++20

Alain Gibaud

ellipses

Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr

ISBN 9782340-065437
©Ellipses Édition Marketing S.A., 2022
8/10 rue la Quintinie 75015 Paris

DANGER LE PHOTOCOPILLAGE TUE LE LIVRE

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5.2° et 3°a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective », et d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit constituerait une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

Je suis heureux d'avoir l'occasion de remercier ici les personnes qui ont accepté de lire et de commenter certaines parties de ce livre, à diverses étapes de son écriture. Chaque contribution, petite ou grande, m'a permis d'avancer.

Merci donc à Allison, Corentin, Françoise, Marianne et Philippe.

Merci surtout à Thérèse, qui a assuré la relecture de l'intégralité du texte. Cette tâche fastidieuse demandait beaucoup de patience et d'opiniâtreté.

≍

Je dédie ce livre à mes enfants, qui m'étonnent chaque jour, et aussi à la mémoire de mon père, qui m'a transmis le goût des sciences et des techniques dès le plus jeune âge.

Alors qu'on demandait à Confucius quelle serait sa première initiative s'il se voyait confier les rênes d'un pays, celui-ci répondit :
« *Rectifier les noms, certainement.* »

Zilu, son disciple, fut alors perplexe :
« *Les rectifier, mais pour quoi faire ?* »

« *Zilu, tu n'es qu'un rustre !* [. . .]
Quand les noms ne sont pas corrects, le langage est sans objet. Quand le langage est sans objet, les affaires ne peuvent être menées à bien. »

Confucius, *Les Entretiens*.
Traduction de Pierre Ryckmans. Gallimard (Folio Sagesses), 2016.

Table des matières

I Généralités **15**

1 À propos de ce livre **17**
1.1 À qui s'adresse-t-il ? . . . 17
1.2 Que contient-il ? . . . 17
1.3 Les exemples . . . 18
1.4 Pictogrammes utilisés . . . 20

2 Évolution des langages C et C++ **21**

3 Informations à l'usage des programmeurs débutants **23**
3.1 Qu'est-ce qu'un ordinateur ? . . . 23
3.2 Langages de programmation . . . 31

4 Les environnements de développement **39**
4.1 Choix d'un environnement . . . 39
4.2 Choix d'un compilateur . . . 40
4.3 Exemples d'utilisation du compilateur `gcc` . . . 40

II Débuter avec le langage C **43**

5 Composition des programmes en langage C **45**
5.1 La notion de fonction en programmation . . . 45
5.2 La fonction `main` . . . 45

6 Les données et leur type **47**
6.1 Variables et constantes . . . 47
6.2 Déclaration des variables . . . 47
6.3 Qu'est-ce qu'un type ? . . . 47
6.4 Les types primitifs . . . 48
6.5 Données constantes . . . 59
6.6 Données volatiles . . . 62

7 Opérateurs **63**
7.1 Opérateurs arithmétiques . . . 63
7.2 Opérateurs sur bits . . . 65
7.3 Opérateurs de décalage . . . 66
7.4 Opérateurs logiques . . . 67
7.5 Opérateurs relationnels . . . 69
7.6 Opérateurs d'affectation . . . 70
7.7 Opérateurs relatifs aux pointeurs . . . 71
7.8 Opérateurs divers . . . 72
7.9 Priorité et associativité des opérateurs . . . 75
7.10 Ordre d'évaluation des opérandes . . . 76
7.11 Expressions hétérogènes et conversion automatique de type . . . 77
7.12 Promotion entière . . . 78

8 Fonctions **79**
8.1 Fonctions et sous-programmes . . . 79
8.2 Paramètres réels et formels . . . 80
8.3 Fonctions sans paramètre . . . 83
8.4 Sous-programmes (fonctions ne retournant rien) . . . 84
8.5 Fonctions à paramètres constants . . . 85
8.6 Fonctions `inline` . . . 85

9 Entrées et sorties **87**
9.1 Qu'est-ce qu'une entrée ou une sortie ? . . . 87
9.2 Sorties : afficher à l'écran avec `printf` . . . 87
9.3 Entrées : saisir des données au clavier avec `scanf` . . . 90

10 Organiser les données **95**
10.1 Tableaux . . . 95
10.2 Structures . . . 102
10.3 Unions . . . 104
10.4 Choisir entre un tableau et une structure . . . 105
10.5 Tableaux et structures comme paramètres de fonctions . . . 106

11 Instructions **109**
11.1 Les prédicats en C . . . 109
11.2 Instructions conditionnelles (tests) . . . 109
11.3 Instructions de répétition (boucles) . . . 112
11.4 Sélection (`switch/case/default`) . . . 118
11.5 Autres instructions contrôlant le flux d'exécution . . . 121

III Approfondir le langage C **125**

12 La fonction `main` **127**
12.1 Arguments des programmes C (et C++) . . . 127
12.2 Arguments comportant des caractères blancs . . . 129
12.3 Valeur retournée par la fonction `main` . . . 129
12.4 La fonction standard `exit` . . . 129

13 Le préprocesseur du langage C **131**
13.1 Qu'est-ce que le préprocesseur ? . . . 131
13.2 Inclure un fichier avec `#include` . . . 132
13.3 Définir une macro avec `#define` . . . 133
13.4 Suppression de macro avec `#undef` . . . 138
13.5 Compilation conditionnelle avec `#if` . . . 139
13.6 Autres # directives . . . 141

14 Déclarations et définitions **143**
14.1 Concepts de déclaration et de définition . . . 143
14.2 Concevoir et analyser des déclarations complexes . . . 144

15 Visibilité et durée de vie **151**
15.1 Visibilité . . . 151
15.2 Durée de vie . . . 154

16 Pointeurs et tableaux **157**
16.1 Arithmétique des pointeurs . . . 157
16.2 Relations entre pointeurs et tableaux . . . 160
16.3 Alors, tableaux et pointeurs sont la même chose ? . . . 162
16.4 Tableaux à indices décalés . . . 162
16.5 Tableaux comme paramètres de fonction . . . 163

17 Allocation dynamique de mémoire **175**
17.1 Contexte . . . 175
17.2 La fonction `malloc` . . . 175
17.3 La fonction `free` . . . 176
17.4 La fonction `realloc` . . . 177
17.5 Erreurs relatives à l'allocation dynamique . . . 178
17.6 Exemple . . . 179

18 Récursivité **183**
18.1 Coût de la récursivité . . . 184
18.2 Quand utiliser la récursivité ? . . . 185
18.3 Exemples . . . 186
18.4 Fonctions à la fois `inline` et récursives . . . 190

IV Débuter avec le langage C++ **191**

19 Utiliser C++ sans créer de classe **193**
19.1 Différences mineures entre C et C++ . . . 193
19.2 Invocation de fonctions C à partir de code C++ . . . 194
19.3 Création de fonctions C à partir de code C++ . . . 195
19.4 Opérateurs spécifiques . . . 195
19.5 Priorité et associativité des opérateurs C++ . . . 196
19.6 Ordre d'évaluation des opérandes . . . 196
19.7 La surcharge des fonctions . . . 197
19.8 Fonctions avec arguments à valeur par défaut . . . 198
19.9 Espaces de noms (`namespace`) . . . 200
19.10 Introduction aux entrées-sorties sur flux . . . 203
19.11 Les références . . . 206
19.12 Quelques éléments de C++ moderne (C++11 à C++20) . . . 213
19.13 Allocation dynamique de mémoire . . . 222

20 Programmation par objets : les concepts **227**
20.1 Classes et objets . . . 227
20.2 Héritage . . . 228
20.3 Encapsulation . . . 230
20.4 Polymorphisme . . . 231

21 Organisation des classes en C++ **233**
21.1 Des structures aux classes . . . 233
21.2 Encapsulation . . . 233
21.3 Modélisation des données . . . 234
21.4 Modélisation des comportements . . . 234
21.5 Organisation du code des classes . . . 236

22 Modéliser un concept : la classe polynôme **239**
22.1 Concept à modéliser . . . 239
22.2 Données caractérisant ce concept . . . 239
22.3 Modéliser les responsabilités : méthodes . . . 240
22.4 Différence entre fonction et méthode . . . 242
22.5 Le pointeur `this` . . . 243
22.6 Méthodes statiques . . . 245
22.7 Méthodes constantes . . . 246
22.8 Fonctions, méthodes et classes amies (`friend`) . . . 248
22.9 Méthodes `inline` . . . 250

23 Construction et destruction **253**
23.1 Problématique de l'initialisation . . . 253
23.2 Construction . . . 253
23.3 Destruction . . . 263
23.4 Allocation dynamique de mémoire et construction . . . 264
23.5 Restitution de mémoire et destruction . . . 265

24 Accesseurs **267**
24.1 Qu'est-ce qu'un accesseur ? . . . 267
24.2 Mutateur simple (setter) . . . 267
24.3 Accesseur simple (getter) . . . 267
24.4 Accesseur et attributs calculés . . . 268
24.5 Accesseur et auto-extensibilité . . . 268
24.6 Accesseur pour objet constant . . . 271
24.7 Constance physique et logique, attribut `mutable` . . . 272
24.8 `const_cast` contre `mutable` . . . 273
24.9 Périmètre d'utilisation des accesseurs . . . 274

25 Surcharge des opérateurs **275**
25.1 Qu'est-ce que la surcharge des opérateurs ? . . . 275
25.2 Périmètre de la surcharge . . . 275
25.3 Techniques de surcharge . . . 276
25.4 Exemple de surcharge d'opérateurs par méthodes . . . 278
25.5 Exemple de surcharge d'opérateurs par fonctions . . . 279
25.6 Exemple d'utilisation des opérateurs . . . 281
25.7 Choisir le mode d'implantation d'un opérateur . . . 281
25.8 Choisir le mécanisme de passage des arguments . . . 282
25.9 Choisir le mécanisme de passage du résultat de l'opération . . . 283
25.10 Les opérateurs importants . . . 283
25.11 Autres opérateurs . . . 294

26 Conversions de types **303**
26.1 Conversion implicite . . . 303
26.2 Conversion par constructeur, spécification `explicit` . . . 305
26.3 Conversion par opérateur . . . 307

27 Héritage **311**
27.1 Réutilisation d'une classe . . . 313
27.2 Héritage public . . . 314
27.3 Héritage privé . . . 317
27.4 Héritage multiple . . . 321

28 Polymorphisme **323**
28.1 Type statique et type dynamique . . . 323
28.2 Méthodes virtuelles . . . 324
28.3 Périmètre du polymorphisme . . . 325
28.4 Opérateurs polymorphes . . . 325
28.5 Polymorphisme et destruction . . . 326
28.6 Classes abstraites et interfaces . . . 328
28.7 Le polymorphisme en action . . . 330
28.8 Du bon usage du polymorphisme . . . 336

29 Généricité **339**
29.1 Fonctions génériques . . . 339
29.2 Comment la généricité fonctionne-t-elle en C++ ? . . . 340
29.3 Périmètre de la généricité . . . 341
29.4 Instanciations implicite et explicite . . . 341
29.5 Classes et structures génériques . . . 343
29.6 Composants génériques à paramètres entiers . . . 347
29.7 Paramètres de modèles avec valeurs par défaut . . . 352
29.8 La généricité en action . . . 353

30 Exceptions **365**
30.1 Problématique de gestion des erreurs . . . 365
30.2 Gestion des erreurs basée sur les exceptions . . . 368

31 Bibliothèque d'entrées-sorties **379**
31.1 Structure et composition de la bibliothèque . . . 379
31.2 Les différents états des flux . . . 383
31.3 Les différents modes d'ouverture des flux . . . 387
31.4 `istream` . . . 387
31.5 `ifstream` . . . 392
31.6 `istringstream` . . . 393
31.7 `ostream` . . . 395
31.8 `ofstream` . . . 396
31.9 `ostringstream` . . . 396
31.10 `iostream`, `fstream` et `stringstream` . . . 397
31.11 Manipulateurs d'entrées-sorties . . . 398
31.12 Créer ses propres manipulateurs . . . 404

32 Expressions lambda (lambda-fonctions) **407**
32.1 Qu'est-ce qu'une expression lambda ? . . . 408
32.2 Expressions lambda « nommées » . . . 409
32.3 Expressions lambda génériques . . . 410
32.4 Type de retour d'une expression lambda . . . 411
32.5 Expression lambda avec paramètres par défaut . . . 411
32.6 Capture de contexte dans une expression lambda . . . 412
32.7 Exemples . . . 415

33 Bibliothèque générique standard (STL) **419**
33.1 Qu'est-ce que la STL ? . . . 419
33.2 Introduction aux conteneurs . . . 419
33.3 Introduction aux itérateurs . . . 420
33.4 Propriétés des conteneurs . . . 424

34 Conteneurs de la STL **425**
34.1 Le conteneur `vector` . . . 425
34.2 Le conteneur `deque` . . . 432
34.3 Le conteneur `array` . . . 437
34.4 Le conteneur `string` (`basic_string<char>`) . . . 439
34.5 Les conteneurs `forward_list` et `list` . . . 445
34.6 Les conteneurs `map` et `multimap` . . . 450
34.7 Les conteneurs `unordered_map` et `unordered_multimap` . . . 459
34.8 Les conteneurs `set` et `multiset` . . . 467
34.9 Les conteneurs `unordered_set` et `unordered_multiset` . . . 468
34.10 Le conteneur `stack` . . . 468
34.11 Le conteneur `queue` . . . 472
34.12 Le conteneur `priority_queue` . . . 473
34.13 Le mandataire `span` (C++20) . . . 477
34.14 Le mandataire `string_view` . . . 480
34.15 La bibliothèque `ranges` (C++20) . . . 483
34.16 Composition de vues . . . 486

35 Algorithmes de la STL **489**
35.1 Introduction aux algorithmes . . . 489
35.2 Algorithmes non modifiants . . . 491
35.3 Algorithmes modifiants . . . 495
35.4 Algorithmes de partitionnement et de tri . . . 503
35.5 Algorithmes sur intervalles triés . . . 507
35.6 Algorithmes ensemblistes . . . 508
35.7 Algorithmes sur les tas (*heaps*) . . . 512

36 Type `initializer_list` **515**
36.1 Création d'un `initializer_list` . . . 515
36.2 Interface du modèle `initializer_list` . . . 515
36.3 Exemple . . . 516

37 Liaison structurée (*Structured binding*) **517**
37.1 Collections autorisant les liaisons structurées . . . 518
37.2 Usages typiques . . . 520

38 Pointeurs sur membres **521**
38.1 Introduction . . . 521
38.2 Exemple . . . 522

39 Pratiquez le « C++ facile » **527**
39.1 Limitez l'usage des pointeurs . . . 527
39.2 Passez les paramètres par valeur ou par référence . . . 528
39.3 N'utilisez pas le système d'entrées-sorties du C . . . 528
39.4 N'utilisez pas les tableaux . . . 528
39.5 N'utilisez pas les chaînes de caractères du langage C . . . 529
39.6 Limitez l'usage du préprocesseur . . . 530
39.7 Ne gérez pas la mémoire vous-même . . . 531
39.8 Utilisez les conteneurs de la STL . . . 531
39.9 Utilisez les exceptions . . . 531
39.10 Pratiquez l'idiome RAII . . . 531

V Approfondir le langage C++ **533**

40 Construction et destruction **535**
40.1 Les différentes sortes d'objets . . . 535
40.2 Construction et destruction d'objets automatiques . . . 536
40.3 Construction et destruction d'objets statiques . . . 536
40.4 Construction et destruction d'objets dynamiques . . . 536
40.5 Construction et destruction d'objets membres ou de base . . . 536
40.6 Construction et destruction d'éléments d'un tableau . . . 537
40.7 Construction et exception . . . 538

41 Allocation dynamique **539**
41.1 Opérateur `new` avec placement . . . 539
41.2 Personnaliser `new`, `new[]`, `delete` et `delete[]` . . . 541

42 Opérateurs **547**
42.1 Opérateur `=` . . . 547
42.2 Opérateurs `++` et `--` . . . 549
42.3 Opérateur `->` . . . 554
42.4 Opérateur `""` . . . 558
42.5 Opérateur de comparaison à trois voies `<=>` (C++20) . . . 563
42.6 Repliement d'opérateur . . . 568
42.7 Les quatre manières de replier une expression . . . 569
42.8 Quelques cas d'utilisation . . . 570
42.9 Opérateurs nommés . . . 573

43 Transtypages **577**
43.1 `static_cast` . . . 577
43.2 `const_cast` . . . 578
43.3 `reinterpret_cast` . . . 578
43.4 `dynamic_cast` . . . 580

44 Identification de type à l'exécution (RTTI) **585**
44.1 L'opérateur `typeid` . . . 585
44.2 La classe `type_info` . . . 586

45 Support du polymorphisme **591**
45.1 Principe général . . . 591
45.2 Organisation de la table des méthodes virtuelles . . . 593
45.3 Invocation des méthodes virtuelles . . . 594
45.4 Obtention du type d'un objet polymorphe . . . 595
45.5 Conclusion . . . 596

46 Héritage **597**
46.1 Héritage, encapsulation et pointeurs . . . 597
46.2 Héritage et passage de paramètres . . . 600
46.3 Héritage et affectation . . . 601
46.4 Affectation polymorphe . . . 603
46.5 Héritage multiple . . . 606
46.6 Héritage dynamique . . . 615
46.7 Construction et destruction en cas d'héritage multiple ou virtuel . . . 624

47 Généricité **625**
47.1 Modèles d'alias (alias template) . . . 625
47.2 Spécialisation des modèles . . . 626
47.3 Spécialisations partielles des modèles . . . 627
47.4 Modèles de paramètres de modèles en C++17 . . . 630
47.5 Guides de déduction . . . 632
47.6 Modèles variadiques (variadic templates) . . . 636

48 Déduction de type **645**
48.1 Déduction de type dans les modèles . . . 645
48.2 Déduction de type avec `auto` . . . 648
48.3 Différence entre `auto` et `decltype` . . . 649

49 Références aux rvalues **653**
49.1 rvalue et lvalue : quelques rappels . . . 653
49.2 Passage d'une rvalue en paramètre . . . 654
49.3 Références aux rvalues . . . 655
49.4 Transmissions par déplacement . . . 656
49.5 Conversion en rvalue avec `std::move` . . . 658
49.6 Méthodes à objet support déplaçable . . . 660

50 Transmissions parfaites **663**
50.1 Réduction des références et références universelles . . . 663
50.2 Transtypage conditionnel avec `std::forward` . . . 666
50.3 Exemple . . . 667
50.4 Les transmissions parfaites dans la STL . . . 671

51 Pointeurs intelligents (smart pointers) **673**
51.1 Introduction . . . 673
51.2 Pourquoi utiliser des pointeurs intelligents ? . . . 673
51.3 `unique_ptr` . . . 673
51.4 `shared_ptr` . . . 682
51.5 `weak_ptr` . . . 688

52 Expressions lambda et closures **699**
52.1 Implantation des expressions lambda . . . 699
52.2 Type d'une closure . . . 699
52.3 Rôle des closures . . . 700
52.4 Capture généralisée . . . 701
52.5 Capture par déplacement . . . 702
52.6 Expressions lambda et références pendantes . . . 703
52.7 Capture par valeur de l'objet support . . . 705
52.8 Expressions lambda génériques et transmissions parfaites . . . 706

53 Exceptions **709**
53.1 Gestion des exceptions non interceptées . . . 709
53.2 Spécifications d'exceptions avant C++11 . . . 710
53.3 Spécifications d'exceptions à partir de C++11 . . . 711
53.4 Méthodes générées automatiquement et exceptions . . . 715
53.5 Écrire du code sûr vis-à-vis des exceptions . . . 717

54 Itérateurs **729**
54.1 Un itérateur pour le type intervalle . . . 729
54.2 Propriétés des différentes catégories d'itérateurs . . . 730
54.3 Conception de la classe `IRange` et de son itérateur . . . 730
54.4 Rendre les itérateurs compatibles avec l'écosystème C++ . . . 734

55 Programmation concurrente et threads **737**
55.1 Qu'est-ce qu'un thread ? . . . 737
55.2 Premiers pas avec les threads . . . 738
55.3 Paramètres de construction d'un thread . . . 741
55.4 Les différentes façons d'implanter un thread . . . 742
55.5 Threads et exclusion mutuelle . . . 744
55.6 Variables de condition et synchronisation . . . 749
55.7 `jthread` (C++20) . . . 752

56 Programmation concurrente et tâches **757**
56.1 Création d'une tâche avec `async` . . . 757
56.2 Détection de la terminaison d'une tâche . . . 758
56.3 Tâches et exceptions . . . 759
56.4 Fonctionnement des tâches (`promise`, `future`) . . . 760

57 Objets exécutables asynchrones **763**
57.1 Introduction aux `packaged_task` . . . 763
57.2 Le modèle `function` . . . 763
57.3 Le modèle `packaged_task` . . . 764
57.4 `packaged_task` et exécution concurrente . . . 765

58 Métaprogrammation **767**
58.1 Introduction . . . 767
58.2 SFINAE . . . 769
58.3 Introspection . . . 776
58.4 Métaprogrammation et bibliothèque standard . . . 780

59 Concepts (C++20) **783**
59.1 Utilisation d'un concept prédéfini . . . 783
59.2 Création d'un concept . . . 784

60 Modules (C++20) **791**
60.1 Qu'est-ce qu'un module ? . . . 792
60.2 Fichiers décrivant les modules . . . 793
60.3 Création d'un module simple . . . 793
60.4 Modules et bibliothèque standard . . . 795
60.5 Utilisation d'entités non importables . . . 796
60.6 Sous-modules . . . 798
60.7 Partitions . . . 800
60.8 Modularisation d'un composant préexistant . . . 803
60.9 Exportation et espaces de noms . . . 804
60.10 Conception des modules . . . 805

Première partie

Généralités

1 | À propos de ce livre

1.1 À qui s'adresse-t-il ?

Ce livre a été écrit pour accompagner les étudiants d'université ou d'école d'ingénieurs dès le début de leur cursus, mais il conviendra aussi à toute personne désirant compléter ou actualiser ses compétences de programmeur C++. En effet, bien qu'il débute par l'introduction de notions de base, cet ouvrage n'est pas pour autant spécifiquement destiné aux débutants. Par le traitement de sujets de difficulté croissante, il se propose de vous guider de manière progressive vers la maîtrise de ce langage très puissant qu'est C++, dans sa plus récente version : C++20.

1.2 Que contient-il ?

Cet ouvrage est essentiellement consacré au langage C++. Toutefois, ce dernier est construit sur le langage C, et la relation qui existe entre ces langages est davantage qu'une simple filiation : C++ est un *surensemble* de C. Ceci signifie que l'étude de C++ inclut forcément celle de C. Le choix qui a été fait dans ce livre est de traiter explicitement le langage C, ce qui permet de faire la distinction entre les fonctionnalités provenant de ce langage et celles qui sont spécifiques à C++.

Il faut toutefois signaler que certaines caractéristiques du langage C sont évoquées parce qu'elles peuvent encore être rencontrées, mais ne sont plus utiles en C++ moderne. Elles peuvent même être parfois considérées comme nuisibles, et sont alors signalées comme telles.

1.2.1 Langage C

Il est impossible de décrire un langage d'une manière à la fois détaillée et accessible aux débutants. Il est en effet fréquent qu'une description approfondie fasse référence à des informations ou à des concepts qui n'ont pas encore été présentés. Pour cette raison, la présentation du langage C a été scindée en deux parties. La première de ces parties, « *Débuter avec le langage C* », contient une description du langage largement suffisante pour écrire des programmes intéressants.
Parce qu'elle s'adresse à des programmeurs débutants, cette partie est toutefois un peu plus qu'une description du langage. Elle comporte des rappels concernant les manières classiques de structurer les programmes et d'organiser les données. Ceci permet à des personnes n'ayant jamais suivi de cours d'algorithmique de lire ce livre avec profit.

La seconde partie, « *Approfondir le langage C* », présente une description plus poussée de certains aspects du langage, et met l'accent sur les points généralement peu abordés, donc souvent mal compris.

1.2.2 Langage C++

La même démarche a été suivie pour le langage C++, dont la présentation occupe plus de 80% de ce livre. Celle-ci comporte également deux parties, nommées « *Débuter en langage C++* » et « *Approfondir le langage C++* ». Certains chapitres portent le même nom dans la première et la seconde partie. Ils traitent du même sujet, mais à des niveaux de détail différents.

La première de ces deux parties débute par une description de ce qu'est la *programmation par objets*, et des concepts associés à ce type d'approche. L'objectif de cette présentation est de montrer que la programmation par objets ne s'appuie pas uniquement sur l'usage de « boîtes à outils » logicielles : c'est aussi une démarche d'analyse des problèmes permettant de concevoir l'architecture de leurs solutions.

Une introduction progressive des fonctionnalités de base de C++ vient ensuite. Elle s'appuie sur des exemples d'abord rudimentaires, puis plus élaborés au fil des chapitres.

La seconde partie traite de sujets un peu plus ardus, et même parfois difficiles si l'on s'intéresse aux aspects les plus avancés. Il faut s'attendre à ce que certaines de ces pages nécessitent plusieurs lectures pour être assimilées. Mais ne paniquez surtout pas, car vous n'êtes pas obligés de *tout* savoir sur le langage C++ pour écrire des programmes intéressants et très performants.

1.3 Les exemples

Apprendre un langage de programmation est une tâche frustrante si chaque point abordé n'est pas illustré par un exemple immédiatement utilisable. Cependant, les premiers exemples sont difficiles à introduire, car il n'est pas possible de présenter simultanément toutes les connaissances requises pour les comprendre. On est donc parfois amené à présenter du code comportant des instructions ou utilisant des concepts qui n'ont pas encore été introduits. Je vous demande donc de vous contenter des explications succinctes fournies, en sachant que des explications plus détaillées suivront.

Les exemples courts ont l'avantage d'illustrer de manière facilement compréhensible les fonctionnalités présentées. Toutefois, ils ne permettent généralement pas d'en démontrer l'intérêt concret. C'est pour cela que ce livre contient beaucoup d'exemples plus ambitieux qui replacent le sujet dans un contexte pratique. Ceux-ci sont plus difficiles à comprendre, car ils tiennent compte de détails qui, bien qu'ils débordent parfois du cadre du sujet traité, rendent le code réaliste.

Il existe dans ce livre trois sortes d'exemples, tous imprimés avec une police à chasse fixe[1], mais avec une présentation spécifique :

(a) Un *fragment de code* est un court extrait destiné à illustrer un point précis. Les fragments ne sont pas utilisables en l'état, car ils sont incomplets : il leur manque généralement un contexte, par exemple des déclarations ou initialisations de variables. Voici un exemple de fragment de code :

```
/* Schéma d'HORNER */
y = ((a * x) + b ) * x + c  ;
```

Dans tous les exemples, les commentaires avec points de suspension tels que :

```
// ...
/* ... */
// ... suite de l'exemple précédent
```

indiquent l'emplacement d'un code qui n'est pas en relation directe avec le sujet, et qui a été volontairement omis pour simplifier la présentation.

(b) Les *fonctions* (ou les *classes*) constituent des outils presque complets, et sont donc généralement utilisables dans un programme par simple copier-coller. Pour des raisons de simplicité et de brièveté, certains éléments sont cependant omis dans les exemples présentés : c'est généralement le cas des inclusions de fichiers, pourtant nécessaires à la compilation de la fonction. Par exemple, une fonction qui réalise des entrées-sorties en langage C++ devrait être précédée de l'inclusion du fichier d'en-tête `<iostream>`, mais ce n'est pas forcément le cas dans ce livre afin d'éviter de le rendre encore plus volumineux.

1. Dans une police à chasse fixe, tous les caractères ont la même largeur. Ceci facilite l'alignement du code et le rend donc plus facilement compréhensible.

Voici un exemple de fonction :

```
// ...
/*
  Calcule la valeur d'un polynôme du second degré,
  de coefficients a,b,c au point x
*/
double poly2(double x, double a, double b, double c)
{
  double y ;
  /* Schéma d'HORNER */
  y = ((a * x) + b ) * x + c  ;
  return y ;
}
// ...
```

(c) Les *programmes* sont des exemples autonomes, car ils sont utilisables tels quels, par simple copie. Certains programmes sont parfois présentés en plusieurs parties pour pouvoir introduire des explications intermédiaires. Dans ce cas, un commentaire `// ...` ou `/* ... */` montre comme d'habitude que du code a été omis. Voici un programme très court écrit en langage C, qui met en œuvre la fonction précédente.

Programme poly2.c

```

#include <stdio.h>
/*
  Calcule la valeur d'un polynome du second degré,
  de coefficients a,b,c au point x
*/
double poly2(double x, double a, double b, double c)
{
  double y ;
  /* Schéma d'HORNER */
  y = ((a * x) + b ) * x + c  ;
  return y ;
}

int main()
{
  double x,a,b,c ;
  printf("Entrez a b c puis x : ") ;
  scanf("%lf%lf%lf%lf", &a, &b, &c, &x) ;
  printf("Valeur du polynome = %f\n", poly2(x,a,b,c)) ;
  return 0 ;
}
```

(d) Finalement, le texte affiché par un programme sera représenté dans ce livre avec une barre latérale de la façon suivante :

```
Valeur du polynome = 23.56
```

1.4 Pictogrammes utilisés

Le texte de ce livre utilise plusieurs sortes de pictogrammes permettant d'attirer l'attention sur un point particulier.


Introduit des explications concernant le code qui précède.


Résumé 22
Introduit un résumé de ce qui précède, et rappelle les principaux points à assimiler.


Introduit un renvoi vers la partie avancée du livre. Ceci permet au lecteur d'approfondir immédiatement le sujet en train d'être traité, s'il le désire.


Introduit des explications techniques concernant l'implantation d'une fonctionnalité particulière. Lire ces explications permet d'avoir une connaissance plus profonde du langage.


Signale une difficulté particulière, ou un piège à éviter.


Signale une fonctionnalité du langage C qu'il est recommandé de ne pas utiliser en C++.

2 | Évolution des langages C et C++

Le langage C est apparu au début des années 70. Il a principalement été créé par Dennis Richie et Ken Thompson, sur la base d'un autre langage utilisé à l'époque, le langage B. À l'origine, le langage C a été conçu pour supporter l'écriture du système d'exploitation UNIX. Or, ce type d'application exige l'écriture de code manipulant les composants de bas niveau des ordinateurs. Pour parvenir à ce résultat, l'usage était alors d'utiliser un langage spécifique à chaque type d'ordinateur, ce qui rendait les programmes ainsi réalisés inutilisables sur une machine d'un autre type. Pour résoudre ce problème, on a été conduit à utiliser des langages de plus haut niveau, qui présentent l'avantage d'être indépendants de la machine cible. Le langage C est largement le plus utilisé d'entre eux : grâce à lui, il a été possible de faire d'UNIX un système d'exploitation portable.

Toutefois, le domaine d'application du C ne se résume pas à la réalisation de logiciels système. En effet, ce langage réalise une excellente synthèse entre des fonctionnalités de bas niveau et des possibilités plus évoluées autorisant l'écriture de toutes sortes d'applications. Une énorme quantité de logiciels a été écrite en C, et les machines pour lesquelles ce langage n'est pas disponible sont rarissimes. Si vous utilisez un ordinateur, un téléphone portable, une tablette, un téléviseur, une automobile ou une machine à laver le linge, sachez qu'il est très probable que le « cerveau » de votre appareil contienne des programmes écrits en C, ou dans un langage dont la mise en œuvre s'appuie sur des morceaux de code écrits en C.

Le langage C est « nu », en ce sens qu'il ne fournit que les fonctionnalités de base permettant d'écrire des programmes. Les autres fonctionnalités, *si elles sont nécessaires*, sont fournies par des bibliothèques de fonctions. Cette approche modulaire contribue à la flexibilité du langage, puisqu'elle permet de faire fonctionner des programmes écrits en C dans toutes sortes d'environnements, éventuellement très pauvres en ressources. Les systèmes embarqués basés sur des microcontrôleurs en sont un exemple.

Le C a fortement inspiré la syntaxe de nombreux autres langages, tels que Java, D, Objective C, C#, php, Javascript, et bien sûr C++. Il leur a aussi fourni un support d'exécution, car certains de ces langages (et bien d'autres dont la syntaxe ne ressemble pas forcément à celle du C) s'appuient sur des bibliothèques support rédigées en C.

Le langage C est *normalisé*. Ceci signifie qu'il existe un document officiel décrivant sa syntaxe et le comportement de chaque fonctionnalité offerte. Pourtant, la première « norme » n'en était pas réellement une : il s'agissait du livre écrit par Dennis Richie lui-même, et par un de ses collègues, Brian Kernighan. Ce livre est le célèbre « The C programming language », aussi connu sous le nom de « K&R ». La norme « K&R » n'est aujourd'hui plus utilisée, mais les normes suivantes le sont. Elles ont été créées par un comité de normalisation ISO/ANSI [1]. On ne retiendra que les normes les plus importantes : C89 (dite aussi C-ANSI) de 1989 qui spécifie la forme « classique » du langage, et les plus récentes C99 (1999), C11 (2011) ou C17 (2017).

Venons-en maintenant au langage C++.
Celui-ci est né des réflexions de Bjarne Stroustrup, qui a eu l'idée d'allier l'efficacité du langage C à la puissance d'expression permise par la programmation par objets. C++ a donc été conçu au départ comme un surensemble du langage C, ce qui a permis de conserver le bénéfice de l'énorme logithèque de ce dernier.

Au fil des années, le langage a été doté de nombreuses fonctionnalités importantes : *l'héritage multiple*, la gestion des *exceptions* et surtout la *généricité*, qui a radicalement changé la manière de l'utiliser, et qui l'a rendu particulièrement extensible.

1. *International Organization for Standardization* et *American National Standards Institute*.

À cause de sa genèse et des similitudes syntaxiques, C++ est considéré par ceux qui ne le connaissent pas comme un « super C », mais c'est une erreur. Non seulement le C++ classique est très différent de C, mais le C++ moderne (c'est-à-dire à partir de C++11) permet une approche de la programmation très différente de celle que proposait son prédécesseur.

On dit parfois que C++ est un langage *multiparadigmes*, ce qui signifie qu'il supporte plusieurs façons d'aborder les problèmes et de créer les solutions correspondantes : il est à la fois *procédural* [2] (comme le C), *à objets* [3], et *générique* [4]. Ces différentes façons de voir les choses peuvent être combinées selon le type de problème à résoudre : on peut ainsi écrire des programmes dans un style *procédural pur*, ou qui seraient *à objets et génériques*, ou *procéduraux et génériques*, ou enfin combiner les trois paradigmes, ce qui est la possibilité la plus utilisée. Enfin, grâce à son approche de la généricité, C++ permet une certaine forme de *métaprogrammation* [5].

C++ est un langage évoluant en permanence, et son comité de normalisation est très actif. Les formes classiques du langage s'appellent C++98 et C++03. Les versions suivantes constituent le *C++ moderne* : il s'agit de C++11, C++14, C++17 et C++20. Cette dernière norme n'était d'ailleurs pas complètement supportée par tous les compilateurs lors de la réalisation de ce livre.

Malgré les différences, il y a un domaine dans lequel la philosophie du C++ est identique à celle du C : chaque fonctionnalité est conçue pour favoriser l'efficacité, et vous ne payez pas pour une fonctionnalité que vous n'utilisez pas. Pour cette raison, C++ est bien adapté aux projets complexes pour lesquels d'excellentes performances sont requises. C'est le cas par exemple des programmes et environnements graphiques, que nous utilisons tous les jours sur nos ordinateurs.

C++ est un langage très puissant dont les fonctionnalités de base sont abordables, mais il est tellement riche et vaste qu'il y a toujours des subtilités à y découvrir, même pour un programmeur expérimenté. C'est pourquoi, si à l'issue de la lecture de ce livre (ou de tout autre ouvrage traitant du même sujet), vous pensez avoir tout compris du C++, il est probable qu'on vous ait mal expliqué.

Bonne lecture !

2. La programmation procédurale est basée sur des assemblages de procédures (aussi appelées sous-programmes ou fonctions) qui implantent des algorithmes.

3. Ce sujet est traité au chapitre 20.

4. Ce sujet est traité au chapitre 29.

5. La métaprogrammation est une manière de programmer permettant dans certains cas d'obtenir des résultats à la compilation, donc sans exécuter de code. Ce sujet est traité au chapitre 58.

3 | Informations à l'usage des programmeurs débutants

3.1 Qu'est-ce qu'un ordinateur ?

Qu'est-ce qu'un ordinateur ? Chacun peut répondre à cette question en fonction de l'usage qu'il en fait. Par exemple, pour moi, en ce moment, c'est un appareil permettant de saisir des textes. Pour le programmeur, un ordinateur est un assemblage de composants qui vont permettre l'exécution de ce qu'on appelle un programme. Bien entendu, écrire ce programme nécessite que l'on ait un minimum de connaissances concernant le fonctionnement de ces composants. L'objet de ce chapitre est de fournir à des programmeurs débutants les éléments permettant de comprendre ce qu'ils manipulent, sans entrer dans d'inutiles détails.

On peut considérer qu'un ordinateur est composé de trois éléments principaux :

(a) La *mémoire volatile* : il s'agit d'un composant électronique qui mémorise instructions et données durant l'exécution d'un programme. Une donnée est une information quelconque manipulée par le programme.

(b) La *mémoire permanente* : réalisée dans une technologie différente [1] de celle de la mémoire volatile, elle permet une mémorisation à long terme, c'est-à-dire lorsque la machine est hors tension.

(c) Le *processeur* : c'est également un composant électronique. Extrêmement complexe, il est capable de reconnaître et d'exécuter les instructions qui constituent les programmes.

Cette description est simplifiée à l'extrême, mais la caractéristique suivante est importante : les informations circulent de façon permanente entre ces différents composants. Pour ce faire, ceux-ci sont interconnectés par des ensembles de fils électriques qu'on appelle des *bus*. Le processeur est la « plaque tournante » par laquelle les informations véhiculées par les bus transitent, sont transformées ou sont générées. La vitesse de circulation des informations dépend de la technologie des bus qui réalisent les connexions : l'information circule très vite à l'intérieur du processeur, beaucoup moins vite entre le processeur et la mémoire volatile, et encore nettement moins vite si l'un des partenaires est la mémoire permanente.

Ces précisions montrent qu'un ordinateur n'a rien d'un support abstrait. Il est au contraire le siège d'opérations physiques qui ont nécessairement un coût. *La capacité d'évaluer ce coût constitue une des compétences du programmeur*.

3.1.1 Mémoire volatile

La mémoire volatile d'un ordinateur (qu'on appellera simplement *mémoire* dans la suite du texte) est un composant électronique permettant d'enregistrer très rapidement une information, et de la restituer à la demande. Le détail du fonctionnement de ce composant importe peu : l'image que doit en avoir un programmeur est celle de la figure 3.1. Comme vous pouvez le voir, il s'agit d'un ensemble de cases, analogues aux tiroirs d'un meuble. Chaque case contient une information très simple, sur laquelle nous reviendrons plus tard.

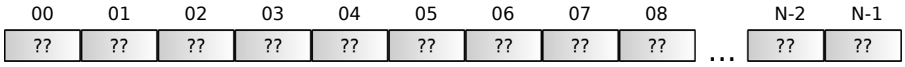


Figure 3.1 – Représentation simplifiée de la mémoire d'un ordinateur

1. Disque magnétique, optique, mémoire flash, etc.

Pour l'instant, deux choses doivent attirer l'attention :

(a) La première est le numéro qui surplombe chaque case, et dont le rôle est d'identifier cette case de manière unique. La première case porte le numéro `0`, et la dernière le numéro `N-1`. Il y a donc `N` cases dans la mémoire représentée ici. Comme, en réalité, `N` peut être très grand (de l'ordre de 10^9), j'ai dû (un peu) simplifier le dessin en ne représentant pas les cases intermédiaires.

Le numéro d'une case s'appelle une *adresse* : il va nous permettre d'accéder aux informations, un peu comme les numéros permettent d'accéder aux maisons, dans une rue. Loin d'être anecdotique, le concept d'adresse est absolument fondamental en programmation, même lorsqu'il n'apparaît pas explicitement. Il est très utilisé en C, et un tout petit peu moins en C++, mais il est bien toujours là.

Concrètement, la mémoire est utilisée de la façon suivante : si nous décidons par exemple qu'une information `X` doit être stockée à l'adresse `4`, il suffit de présenter simultanément l'adresse `4`, l'information `X` et l'ordre de stockage à la mémoire pour que cette information soit mémorisée. Pour la retrouver, il suffit de présenter simultanément l'adresse `4` et l'ordre de consultation pour que la mémoire restitue l'information `X`.

Rassurez-vous, vous n'aurez pas besoin de « dialoguer » avec la mémoire aussi explicitement : tout ceci se fait de manière transparente lors de l'exécution des programmes. Mais il est tout de même très important de savoir ce qui se passe.

(b) Le second point important concerne le contenu des cases. Sur le schéma, il est symbolisé par `??`, ce qui indique une valeur *inconnue*. Ceci ne signifie nullement que la case est vide, mais qu'elle contient une information inconnue (et éventuellement aléatoire), ce qui est très différent. On touche ici aux limites de la métaphore, car contrairement aux tiroirs mentionnés ci-dessus, une case mémoire n'est *jamais vide*.
Jusqu'à présent, nous n'avions pas fait de supposition sur la nature des informations stockées en mémoire. Pour les besoins de la présentation, nous allons maintenant supposer que ces informations sont des nombres, ce qui est de toute façon presque vrai.

Voici deux instructions rédigées dans un langage imaginaire permettant de donner à certaines cases mémoire un contenu défini :

```
mettre 23 dans la case 4
mettre le contenu de la case 4 dans la case 8
```

ce qui s'écrirait dans un langage un peu plus « informatique » :

```
mettre #23,4
mettre 4,8
```

Vous pouvez voir que les nombres qui apparaissent dans cet exemple sont en réalité des adresses, sauf celui qui débute par le caractère « `#` ». C'est par ce moyen qu'on indique avec ce langage qu'une valeur représente réellement un nombre, et non une adresse.

Ne paniquez pas, les autres exemples de ce livre ne ressembleront pas du tout à celui-ci ! Le seul objectif de cet exemple est de vous montrer qu'un programme ne peut pas modifier l'état de la mémoire[2] sans utiliser explicitement ou implicitement les adresses des cases à modifier. De plus, cet exemple montre que les adresses permettent un accès aux informations dans un *ordre arbitraire*. On parle alors d'accès *direct* ou *aléatoire*.

Le schéma de la figure 3.2 montre le contenu de la mémoire après l'exécution de ce code.

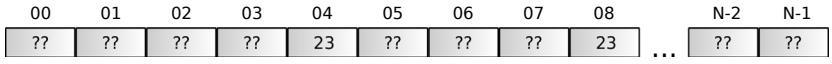


Figure 3.2 – Nouveau contenu de la mémoire

2. Ce qui est le rôle fondamental de tout programme.

Nous allons maintenant parler de la nature exacte des informations stockées dans chaque case de la mémoire. Il ne faut pas perdre de vue la nature physique des ordinateurs : ce sont des machines électroniques. Il s'avère que dans ce domaine technologique, il est facile de représenter une information pouvant prendre deux états, par exemple une *présence* ou *absence* de tension électrique. On appelle cela une information *binaire*.

À première vue, on ne peut pas faire grand-chose avec ce type d'information, sauf bien sûr représenter des informations naturellement binaires telles que « oui » ou « non ». Par convention, on appelle les deux états possibles « `1` » et « `0` ». C'est bien plus court que « oui » et « non », et cette interprétation nous sera utile plus tard.

Comment représenter des informations plus subtiles que `oui` et `non` avec des `1` et des `0` ? Il suffit d'y consacrer « suffisamment » de `0` et de `1`, et de définir des *conventions de codage*. Le tableau de la figure 3.3 montre par exemple comment coder une réponse évasive en utilisant deux informations binaires. Précisons maintenant quelques points de terminologie : les informations binaires stockées

Réponse à coder	Code correspondant
Non	`00`
Peut-être non	`01`
Peut-être oui	`10`
Oui	`11`

Figure 3.3 – Codage d'une réponse pouvant être évasive

dans les cases mémoire s'appellent des *bits*. Ce mot est l'abréviation anglaise de « **bi**nary digi**t** », qui signifie chiffre binaire. D'ailleurs, dans le jargon informatique, le terme anglais *digit* est souvent utilisé au lieu de chiffre. De plus, retenez qu'on utilise plutôt le terme *mot* que « case mémoire ».

Chaque mot contient un nombre fixe de bits, par exemple 8, 16, 32 ou 64 bits selon les machines. Pour simplifier, nous supposerons ici que la mémoire est composée de mots de 8 bits, qu'on appelle aussi des *octets* (*bytes* en anglais).

Combien d'informations différentes peut-on représenter avec 8 bits ? Pour le savoir, il suffit de compter toutes les combinaisons possibles de `0` et de `1` :

```
00000000
00000001
00000010
...
11111101
11111110
11111111
```

Il y a en tout 256 combinaisons possibles.

Bien sûr, on pouvait anticiper ce résultat, car on sait que 8 chiffres binaires permettent de coder $2^8 = 256$ états différents.

Bon, d'accord, pensez-vous. Mais un ordinateur est censé être un champion dans la manipulation des nombres. Comment les nombres sont-ils représentés ? C'est ici que la convention selon laquelle chaque bit représente 0 ou 1 va nous être utile. Elle nous permet de considérer qu'un mot contient un nombre « écrit » en base 2.

Par exemple, pour un mot de 8 bits, ce nombre est de la forme

$$b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$$

où chaque bit b_i peut valoir 0 ou 1. La valeur de ce nombre est donc :

$$N = b_7 \times 2^7 + b_6 \times 2^6 + \cdots + b_1 \times 2 + b_0 = \sum_{i=7}^{0} b_i \times 2^i$$

Pour fixer les idées, le schéma de la figure 3.4 montre la structure interne de deux mots en mémoire. Le mot d'adresse 7 a un contenu inconnu, chaque bit est donc symbolisé par un « **?** », qui peut être soit **0**, soit **1**.

Traditionnellement, les bits sont numérotés de gauche à droite de b_7 à b_0. Leur numéro est simplement l'exposant de 2 dans la formule précédente. Pour cette raison, on dit que le bit de droite est le *bit de poids le plus faible* : il représente les unités. De même, le bit de gauche (b_7, ou *bit de poids le plus fort*) a un poids de $2^7 = 128$. Toutefois, les termes gauche et droite n'ont de sens que lorsque le nombre est exhibé pour lecture : en mémoire, il n'y a ni gauche ni droite.

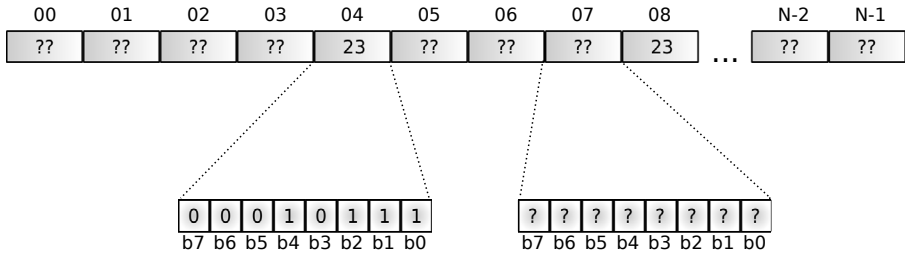


Figure 3.4 – Le contenu de deux mots en mémoire

Vérifions que le mot d'adresse 4 vaut bien 23. En développant la formule, on obtient :

$$0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 16 + 4 + 2 + 1 = 23$$

Ouf ! Le compte est bon.

Il faut insister sur le fait que ceci n'est que le résultat d'une convention de représentation. Il existe d'autres façons de représenter les nombres entiers, mais celle-ci est de loin la plus utilisée.

On a vu précédemment que chaque octet permettait de stocker un nombre compris entre 0 et 255. Mais comment stocker un nombre plus grand ? La réponse est simple : il suffit d'utiliser plusieurs octets consécutifs pour constituer un mot de plus grande taille. Grâce à cette technique, il est donc possible de stocker de très grandes valeurs, mais comme un nombre peut être arbitrairement grand, l'éventualité qu'une valeur ne puisse pas être représentée existe toujours. Lorsque cela arrive, on parle de *débordement*, et la valeur n'est pas représentée correctement [3].



La conséquence d'un débordement pouvant être catastrophique, il appartient au programmeur de s'assurer que les nombres manipulés soient toujours représentables.

Mais le monde ne se réduit pas aux nombres entiers positifs. Comment représenter les autres sortes d'informations ? Voici une réponse générale : il suffit d'y consacrer suffisamment d'octets, et de développer les conventions de représentation qui conviennent. Tous les langages de programmation permettent par exemple de représenter des entiers relatifs, des nombres réels ou des caractères. Bien entendu, ceci peut ne pas être suffisant. Dans ce cas, vous pouvez inventer vos propres types

3. Le stockage de la valeur 256 provoque par exemple un débordement *sur un octet*. Le résultat de ce débordement est la valeur 0.

de données, et définir leur représentation ainsi que les opérations qui les concernent. Vous verrez que cette activité est d'ailleurs courante lorsqu'on programme en C++, car elle est à la base de la programmation par objets.

3.1.2 Mémoire permanente

La mémoire permanente permet de conserver des informations lorsque l'ordinateur est hors tension. Tous les utilisateurs d'ordinateurs savent que le support de cette information est un périphérique tel qu'un disque magnétique, ou un dispositif électronique comportant une mémoire permanente [4]. Ils savent aussi que cette mémoire est structurée en *répertoires* (ou *dossiers*), qui contiennent des *fichiers*.

Mais qu'est-ce qu'un fichier pour un programmeur ? En simplifiant, il s'agit essentiellement d'un ensemble d'octets repérés par un numéro. Présenté comme ceci, la similitude avec la mémoire volatile est frappante. Pourtant, un fichier est très différent d'un bloc d'octets résidant en mémoire, car les technologies sous-jacentes sont totalement différentes. Voici une liste des différences les plus importantes :

(a) Le périphérique qui supporte le fichier n'est pas directement relié au processeur, mais piloté par un dispositif électronique dédié, nommé *contrôleur*.

(b) On ne peut y accéder qu'en exécutant des morceaux de programmes spéciaux fournis par le langage de programmation utilisé, ou éventuellement par le système d'exploitation [5].

(c) L'accès aux informations n'est pas direct. Pour simplifier, on peut considérer que l'accès à un octet quelconque d'un fichier n'est possible qu'après avoir lu (ou écrit) les octets précédents.

La conséquence de tout ceci est que la vitesse d'accès aux informations qui résident en mémoire permanente est de plusieurs ordres de grandeur plus faible que lorsque le processeur accède à la mémoire volatile.



Cette caractéristique est tellement pénalisante qu'on a développé un ensemble de techniques permettant d'améliorer les performances des programmes qui utilisent des fichiers. La plus classique consiste à utiliser des tampons (ou *buffers*).

Un tampon est simplement une zone mémoire qui sert au stockage temporaire des données à écrire (ou à lire) dans un fichier. Voici pourquoi on les utilise : les échanges avec la mémoire permanente (sur laquelle résident les fichiers) sont beaucoup plus performants lorsqu'on transfère de grandes quantités de données en une seule opération. Pour profiter de cette caractéristique, il est donc logique de regrouper les données à transférer. C'est précisément ce que permet l'usage d'un tampon.

La gestion des tampons est automatique, et le programmeur a peu à s'en soucier. Alors, pourquoi en parler ? Supposez que votre programme rencontre une erreur et s'arrête brutalement juste entre l'instant où les données sont écrites dans le tampon et celui où elles sont recopiées dans le fichier. Dans cette situation, le fichier sera corrompu, dans le sens où il ne contiendra pas la totalité des données écrites. Vous serez certainement confronté un jour à cette situation surprenante, alors autant être averti.

3.1.3 Le processeur et son fonctionnement

Le processeur est le composant de l'ordinateur qui *reconnaît* et *exécute* les instructions qui composent un programme. Une instruction doit être considérée dans ce contexte comme un ordre qui indique au processeur la prochaine action à entreprendre. Les processeurs sont extraordinairement complexes, aussi on n'en fera ici qu'une description simpliste. On insistera toutefois sur les points qui permettent de mieux comprendre certains concepts des langages C et C++, et de nombreux autres langages de programmation.

4. Par exemple une mémoire en technologie flash.
5. Il s'agit de fonctions de *lecture* et d'*écriture*.

Chaque processeur contient de petits espaces de mémoire volatile qu'on appelle des *registres*. Fondamentalement, ces registres ne sont pas très différents des cases mémoires, telles qu'elles sont décrites dans la section 3.1.1. Ils servent généralement à stocker des résultats intermédiaires, et le programmeur n'a en principe pas à s'en soucier tant qu'il ne programme pas en langage machine. Toutefois, certains registres jouent un rôle tellement important dans le fonctionnement du processeur qu'il est bon de savoir comment ils sont exploités, même si vous ne les manipulez jamais explicitement. On va donc examiner les deux plus importants d'entre eux, qui s'appellent généralement PC et SP.

3.1.3.1 Rôle de PC dans l'exécution séquentielle des instructions

Au lancement d'un programme [6], le registre PC (Program Counter) est chargé avec l'adresse de la première instruction à exécuter. Après ce chargement, le processus suivant est entrepris :

- ■1 La valeur contenue dans PC est communiquée par le processeur à la mémoire volatile, qui lui fournit en retour l'instruction se trouvant à cette adresse,
- ■2 le processeur ajoute 1 à la valeur du registre PC,
- ■3 le processeur exécute l'instruction qui lui a été fournie par la mémoire,
- ■4 le processus est repris à l'étape ■1.

Durant la phase d'exécution ■3, PC contient donc toujours l'adresse de la *prochaine* instruction à exécuter [7], qui est celle de l'instruction en cours d'exécution, plus 1. C'est par ce mécanisme automatique de progression vers les adresses croissantes que les différentes instructions qui sont stockées en mémoire sont exécutées, les unes à la suite des autres, dans l'ordre où elles sont rencontrées.

Toutefois, ce principe astucieux ne permet qu'une exécution séquentielle des instructions. Or, vous savez qu'un programme a la capacité de réagir en fonction des données qui lui sont présentées. Pour faire ceci, il faut qu'il puisse changer (dans une certaine mesure) son comportement, c'est-à-dire rompre le caractère séquentiel du flux des instructions. Vous devez entrevoir le moyen d'y parvenir : il suffit qu'il existe une instruction capable de modifier la valeur de PC. Cette opération extrêmement fréquente s'appelle un *saut*, ou un *branchement*.

Ce mécanisme conceptuellement très simple permet de réaliser les *traitements conditionnels ou répétitifs* qui sont à la base de tout programme. Le schéma de la figure 3.5 montre par exemple un morceau de programme qui réalise un traitement répétitif (ou *boucle*) grâce à une altération de la valeur de PC.

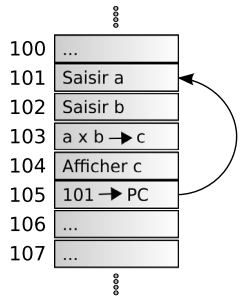


Figure 3.5 – Exemple de réalisation d'une boucle par modification du registre PC.

6. Nous supposons ici que le programme a déjà été copié dans la mémoire de l'ordinateur.

7. Ce comportement est semblable à celui d'une personne lisant un texte à haute voix : ses yeux sont toujours en avance par rapport au texte qu'il est en train d'énoncer. On verra plus loin en quoi ce fonctionnement est intéressant.

Dans ce schéma, chaque case représente un mot mémoire, et contient une instruction d'un langage imaginaire. La valeur figurant à gauche de la case est simplement l'adresse de cette case. Ce petit bout de code commence à l'adresse 101. Il comporte à l'adresse 105 une modification du registre PC qui conduit le processeur à systématiquement continuer le traitement à l'adresse 101. Il est donc prévisible que ce programme ne se terminera jamais, puisqu'il revient toujours au même point : il comporte ce qu'on appelle une *boucle infinie*.

3.1.3.2 Rôle de SP dans l'exécution des sous-programmes

Le registre SP (Stack Pointer) joue également un rôle capital dans le fonctionnement des processeurs, et son exploitation est encore plus astucieuse que celle de PC. Il permet de gérer une zone de la mémoire dédiée au stockage *temporaire* de données. Il serait prématuré de préciser de quelles données il s'agit, mais on va exposer d'une manière générale comment ce registre est exploité.

En début de programme, SP est initialisé avec l'adresse d'une zone mémoire dédiée au stockage des données temporaires. Chaque fois que l'on veut stocker une donnée, on ajoute 1 à la valeur de SP, puis l'on fournit sa valeur et la donnée à stocker à la mémoire. Celle-ci joue alors son rôle : elle stocke la donnée à l'adresse spécifiée. À tout moment, SP contient donc l'*adresse de la dernière donnée* qu'il a permis de sauvegarder.
Pour stocker une autre donnée, le même processus est exécuté. Cette nouvelle donnée sera donc rangée à l'adresse suivante, et ainsi de suite pour celles qui seront stockées ultérieurement. Les données sont donc virtuellement « empilées » les unes derrière les autres selon une organisation qu'on appelle une *pile*. C'est de là que vient le nom *stack pointer* c'est-à-dire « pointeur de pile ».

À titre d'exemple, le schéma de la figure 3.6 montre une pile qui a été arbitrairement implantée à l'adresse **58**. Les valeurs **26**, **43** et **77** ont été empilées dans cet ordre, et SP contient donc 60, qui est l'adresse de la dernière valeur empilée. Ce schéma est redondant, puisqu'il comporte une flèche qui met en évidence le fait que SP *pointe* sur le sommet de pile. Comme d'habitude, les autres mots mémoire ont un contenu inconnu, ce qui est représenté par des points d'interrogation.

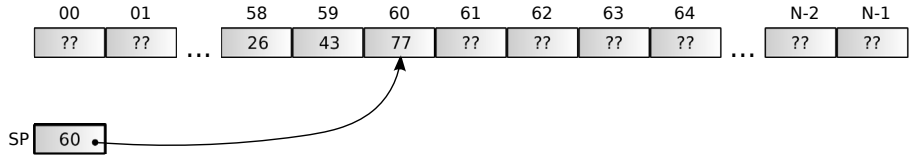


Figure 3.6 – État de la pile après empilement des valeurs **26**, **43** et **77**

Afin de rendre le processus d'empilement moins abstrait, imaginez par exemple que chaque donnée représente un livre empilé sur une étagère. Ce type de rangement permet d'accéder facilement au livre qui se trouve en sommet de pile, mais beaucoup moins facilement aux autres. Toutefois, ceci n'est pas un inconvénient si l'on désire juste faire du stockage.

Pour exploiter les données stockées dans une pile, les deux opérations suivantes sont les plus fréquentes :

(a) Obtenir la donnée située en sommet de pile : le processeur a juste à fournir le contenu de SP à la mémoire, qui fournit en retour la donnée située à cette adresse.

(b) Supprimer la donnée située en sommet de pile : il faut non seulement « l'oublier » , mais en plus rendre disponible l'espace mémoire correspondant, ce qui permettra éventuellement de s'en servir à nouveau. Avec une pile, rien n'est plus facile : il suffit d'enlever 1 à SP (décrémentation), et rien d'autre. La prochaine donnée empilée prendra donc la place de celle qui vient d'être supprimée.

Le schéma de la figure 3.7 montre l'état de la pile après le retrait de la dernière valeur empilée. Remarquez que ce retrait est purement logique, car la valeur **77** se trouve toujours en mémoire. Elle

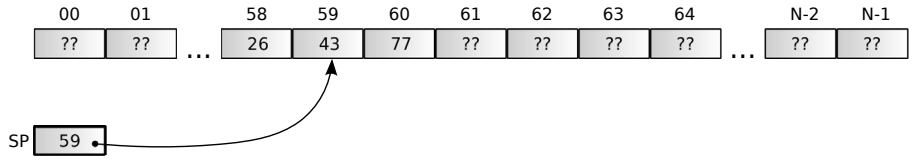


Figure 3.7 – État de la pile après le retrait de la valeur **77**

n'est toutefois plus accessible, car elle se trouve au-delà du sommet de la pile. Maintenant que le fonctionnement d'une pile est connu, on va voir comment celle-ci est utilisée lors de la gestion des sous-programmes. Un *sous-programme* est un morceau de code dédié à un traitement particulier [8] susceptible d'être exécuté en divers points d'un programme. Après chaque exécution, le déroulement du programme principal doit pouvoir être repris *là où il avait été interrompu*.

Pour provoquer l'exécution d'un sous-programme, il suffit qu'il existe dans le programme principal une instruction qui mette l'adresse de la première instruction du sous-programme dans le registre PC, selon le principe exposé à la section précédente. Malheureusement, on ne peut pas appliquer la même stratégie pour revenir au programme principal, car le point de retour peut être différent pour chaque exécution du sous-programme. Le schéma (a) de la figure 3.8 montre ce qui se passe si l'on ne tient pas compte de cela : l'instruction exécutée après le sous-programme est toujours la même.

Pour qu'un sous-programme puisse être exploité correctement , il faut donc lui fournir d'une manière ou d'une autre l'adresse de l'instruction à exécuter lorsqu'il sera terminé. C'est ici que le registre SP in-

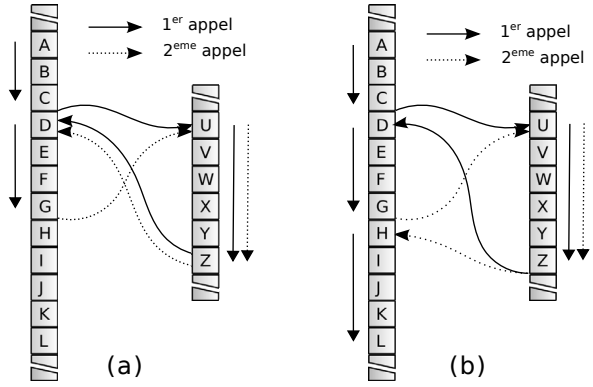


Figure 3.8 – Gestion incorrecte (a) et correcte (b) des adresses de retour.

tervient : il est utilisé pour sauvegarder la valeur de PC avant l'exécution de chaque sous-programme. Le nouveau scénario de gestion des exécutions de sous-programmes est le suivant :

- 1 Exécutons une instruction (souvent nommée `call`) qui commence par empiler PC, selon le mécanisme décrit précédemment. Comme PC est incrémenté *avant* l'exécution de chaque instruction, la valeur empilée est *l'adresse de l'instruction suivante*, c'est-à-dire précisément ce dont nous avons besoin. Après cela, cette instruction met l'adresse de la première instruction du code à exécuter dans PC.
- 2 Exécutons la séquence de code située à partir de cette adresse, c'est-à-dire le sous-programme.
- 3 À la fin de cette séquence, exécutons une instruction dont le rôle est de retirer la valeur située en sommet de pile, et de la ranger dans PC. Cette instruction s'appelle souvent `return`.

8. Imaginez par exemple un calcul d'usage général, tel que l'extraction d'une racine carrée.

Grâce à ce procédé simple et très astucieux, le programme continue son exécution à l'instruction qui suit l'appel du sous-programme, comme le montre le schéma (b) de la figure 3.8. Le grand intérêt de ce mécanisme est qu'il fonctionne également si le sous-programme exécute lui-même un autre sous-programme, et ainsi de suite. Il n'y a pas de limite à la profondeur d'imbrication des appels, si ce n'est la taille de la mémoire dédiée à la pile. En pratique, des centaines d'appels peuvent être imbriqués, voire davantage si l'on utilise une technique de programmation nommée *récursivité*, qui est traitée au chapitre 18.

Résumé 1

(a) La mémoire est composée de « cases » identifiées de manière unique par un numéro que l'on appelle *adresse*.

(b) Ces cases peuvent contenir des *données* ou des *instructions*.

(c) Les instructions sont des ordres élémentaires prédéfinis, que le processeur sait reconnaître et exécuter.

(d) Le processeur comporte des *registres* qui permettent de mémoriser temporairement des informations quelconques. Certains, comme PC et SP, ont toutefois un rôle particulier.

(e) PC contient à tout moment l'adresse de la prochaine instruction à exécuter, ce qui permet au processeur de connaître cette instruction. La modification de PC permet de changer explicitement l'ordre d'exécution des instructions.

(f) SP autorise la décomposition des programmes en sous-programmes. Dans ce contexte, il permet (entre autres) la mémorisation des adresses de retour.

En conclusion de cette section, il faut reconnaître qu'il est possible de commencer à programmer sans aborder le fonctionnement des processeurs. Toutefois, posséder ces notions permet de comprendre immédiatement (et donc d'assimiler facilement et durablement) certaines caractéristiques des langages de programmation, en particulier celles qui concernent l'usage des fonctions. C'est ainsi que les notions de *variables locales* (chapitre 15), de *passage de paramètres* (section 8.2.1) et de *récursivité* (chapitre 18) sont intimement liées à la pile d'exécution.

3.2 Langages de programmation

3.2.1 Langage machine et langage d'assemblage

```
00001110 00011100
11001111 11101000 11110000 00101100
```

Le texte précédent montre un morceau de programme sous la forme d'un ensemble de bits, tel qu'il est implanté dans la mémoire d'un processeur. Pour en améliorer la lisibilité, j'ai regroupé ces bits par paquets de 8, donc par octets, mais cela ne rend pas le programme plus compréhensible. C'est pourtant sous cette forme que les premiers programmeurs développaient leurs œuvres à partir de la fin des années 40.

Afin de rendre la lecture plus facile, il est possible de travailler en base 16 (*hexadécimal*) plutôt qu'en base 2. Voici donc le même morceau de programme, rédigé à la manière hexadécimale.

```
0E1C
CFE8 F02C
```

Le code obtenu est plus compact, mais à moins d'être initié, il reste obscur. J'insiste sur le fait qu'en mémoire, ce code est exactement le même que le précédent : seule l'écriture externe en est différente. Par exemple, pour le dernier octet, vous pourrez vérifier que $\texttt{00101100}_2$ (`00101100` en base 2) est bien égal à $\texttt{2C}_{16}$ (`2C` en base 16).

Ce code est difficile à comprendre, car il est rédigé en *langage machine*. Les programmeurs des temps héroïques ont donc rapidement compris qu'il fallait développer des langages basés sur des symboles, qui sont plus faciles à manipuler par l'esprit humain. Le premier langage symbolique a été le *langage d'assemblage*. Il s'agissait d'un équivalent textuel [9] du langage machine, tel que celui-ci :

```
movlw 28         ; mettre la valeur 28 dans le registre W
movff WREG,C18_i ; recopier le contenu de W dans le mot C18_i
```

Bien que ce texte nous semble plus familier que la suite de `0` et de `1` précédente, il réclame une explication pour être compris : j'ai donc ajouté après chaque instruction un texte explicatif, ou *commentaire* [10], débutant par un point-virgule.

Comme vous savez ce qu'est un registre ou un mot mémoire, vous pouvez comprendre avec l'aide des commentaires ce que fait ce code : il s'agit de mettre la valeur 28 dans le mot nommé `C18_i`. Remarquez que `C18_i` est un nom arbitraire qui représente une adresse particulière. C'est ce qu'on appelle un *symbole*, ou plus simplement un *nom de variable*.
L'intérêt d'utiliser des symboles est que nous n'avons plus besoin de nous préoccuper des adresses des mots : nous donnons un nom à ceux qui nous intéressent, et nous n'avons plus qu'à utiliser ce nom pour désigner la (ou les) case(s) correspondante(s).

Ceci est intéressant pour le programmeur, mais on n'a fait ici que déplacer le problème. En effet, le processeur ne peut pas comprendre le texte symbolique, car il est essentiellement spécialisé dans les `0` et les `1`. Il est donc nécessaire de traduire ce texte pour lui fournir quelque chose qu'il peut traiter. Ceci est le rôle d'un programme spécifique qu'on appelle un *assembleur*. Ainsi, chaque instruction écrite en langage d'assemblage est traduite en une instruction en langage machine, comme dans le tableau de la figure 3.9, où vous reconnaîtrez notre code machine initial.

Langage d'assemblage		*Langage machine*
`movlw 28`	→	`00001110 00011100`
`movff WREG,C18_i`	→	`11001111 11101000 11110000 00101100`

Figure 3.9 – Traduction par un assembleur du langage d'assemblage en langage machine.

Toutefois, malgré l'aide de l'assembleur, écrire de gros programmes de cette manière est long et fastidieux. De plus, chaque type de processeur possède son propre langage machine (donc son propre langage d'assemblage) ce qui implique que les programmes obtenus seront *non portables*, c'est-à-dire non réutilisables dans un autre environnement. Il a donc fallu inventer d'autres types de langages qui ne présentent pas cet inconvénient.

3.2.2 Langages compilés

Afin d'accélérer le processus de développement des programmes, on a rapidement inventé des langages permettant d'écrire des programmes sans faire explicitement référence aux instructions disponibles sur le processeur cible. C'est par exemple le cas des langages C et C++ présentés dans ce livre. Pour chaque langage créé, on a défini des notations compactes qui ont souvent été inspirées par les notations mathématiques existantes, et qui permettent d'indiquer de manière concise ce que l'on souhaite faire.
Bien entendu, on a aussi développé les traducteurs correspondants, appelés *compilateurs*. Fondamentalement, les compilateurs jouent le même rôle que les assembleurs, mais ils sont beaucoup plus complexes et génèrent un grand nombre d'instructions-machine pour chaque instruction à traduire. Le tableau de la figure 3.10 montre que, même pour traduire des instructions très simples, un compilateur peut générer beaucoup d'instructions-machine.

9. Par opposition à numérique.
10. Vous verrez par la suite que, quel que soit le langage de programmation utilisé, les commentaires permettent de faciliter la lecture des programmes. Ils sont donc extrêmement utiles aux programmeurs qui ont à modifier ou à corriger des programmes préexistants.

Langage source		*Traduction en langage d'assemblage*
`i=28;`	→	`movlw 28` `movff WREG,C18_i`

`j=j+i;`	→	`movff C18_i,PREINC0`
		`movff C18_j,WREG`
		`addwf INDF0,F,0`
		`movff POSTDEC0,C18_j`

Figure 3.10 – Résultat de la compilation de deux instructions simples.

Précisons un point de terminologie : le programme à traduire (à gauche dans le tableau) se nomme *programme source*, alors que le résultat de la traduction (à droite) est le *programme cible*. Dans cet exemple, le langage cible est le langage d'assemblage, mais certains compilateurs génèrent du code machine directement à partir du code source.

Utiliser un langage compilé est très intéressant, car le développement des programmes est beaucoup plus facile et rapide, avec moins de chances de se tromper. En effet, comme vous pouvez le constater sur cet exemple, le code source est beaucoup plus facile à comprendre et à relire que le code cible. Un autre avantage de ce mode de traduction est que les programmes sont portables, à condition de disposer d'un compilateur pour la machine cible [11].

Le programme en langage d'assemblage fabriqué par le compilateur est ensuite traduit en code machine, comme cela est expliqué aux sections précédentes. Celui-ci est stocké sur un support permanent, ce qui permettra de l'exécuter plus tard, et aussi souvent que désiré. Pour cela, il suffira de le recopier dans la mémoire de l'ordinateur, comme le montre le schéma de la figure 3.11.

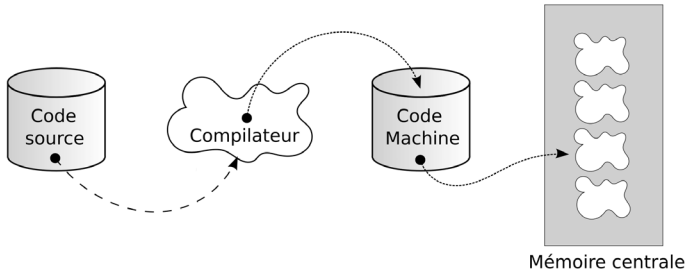


Figure 3.11 – Représentation simplifiée du processus de développement en mode compilé.

Toutefois, cette description du processus de développement est une simplification de la réalité, comme nous allons le voir tout de suite.

3.2.3 Édition de liens

3.2.3.1 Résolution des références externes

Le schéma de la figure 3.11 décrit de manière simplifiée le processus de développement d'un programme en mode compilé. En réalité, il faut tenir compte du fait que le code source des programmes n'est presque jamais entièrement contenu dans un seul fichier. En pratique, il arrive que ce code soit réparti dans des centaines de fichiers. Lorsque ces fichiers sont compilés, on obtient pour chacun d'entre eux un autre fichier contenant du code machine, qu'on appelle *fichier objet*. Le

11. En pratique, cette restriction est rarement contraignante.

schéma de la figure 3.12 montre la situation juste après la compilation d'un programme composé de trois fichiers sources. Le point important de ce processus est le suivant : les fichiers objets sont gé-

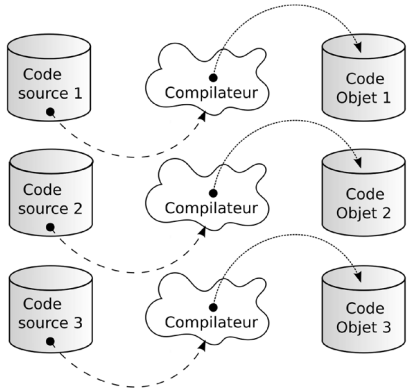


Figure 3.12 – Compilation d'une application répartie sur trois fichiers sources.

néralement incomplets. En effet, souvenez-vous que les langages de programmation travaillent avec des symboles. Pour rester très général, un symbole est un nom qui représente l'adresse d'une case mémoire particulière [12]. Lorsque cette case et le symbole correspondant sont définis dans le même fichier source, il est facile de les associer. Par conséquent, chaque fois que le symbole est utilisé par le programmeur, le compilateur est capable de remplacer ce symbole par l'adresse correspondante. Cependant, il est très fréquent qu'on utilise des symboles qui correspondent à quelque chose qui est défini dans un autre fichier source : ces symboles sont alors des *références externes*. Dans ce cas, le compilateur ne peut pas connaître l'adresse associée au symbole, car il n'a pas de vision globale de l'ensemble des fichiers sources composant l'application. Il en est donc réduit à fabriquer un code incomplet, dans le sens où il comporte des adresses provisoirement indéfinies.

Pour sortir de cette situation fâcheuse, on utilise un autre programme nommé *éditeur de liens* (ou *linker*), qui va épauler le compilateur. Le rôle de l'éditeur de liens est double : il va tout d'abord fusionner les différents fichiers objets pour fabriquer un fichier unique. Mais durant cette fusion, il examine aussi chaque fichier et tente de résoudre ses références externes.
Par exemple, si le premier fichier utilise un symbole x qui est une référence externe, l'éditeur de liens va analyser les autres fichiers pour voir s'ils comportent une zone mémoire portant ce nom. Ici, trois situations peuvent se présenter :

(a) Cette zone est trouvée, et l'éditeur de liens peut terminer le travail que le compilateur n'avait pas pu mener à terme : il remplace le symbole par l'adresse correspondante. Ceci est le cas normal, et le plus courant.

(b) Cette zone n'est pas trouvée : c'est ce qu'on appelle une *référence indéfinie*.

(c) Plusieurs zones portent ce nom, et elles sont dans des fichiers différents : on est dans une situation ambiguë de *références multiples*.

Les deux derniers cas génèrent une erreur fatale qui interdit la fabrication du fichier exécutable final. Le schéma de la figure 3.13 illustre le processus de compilation et d'édition de liens dans le cas où tout se passe bien.

Ne soyez pas effrayés par l'apparente complexité du processus, car en réalité les opérations de compilation et d'édition de liens sont généralement déclenchées par une unique commande. Si

12. Cette case peut correspondre au début d'un ensemble de données (c'est le cas le plus fréquent), ou au début d'un morceau de programme (sous-programme ou fonction).

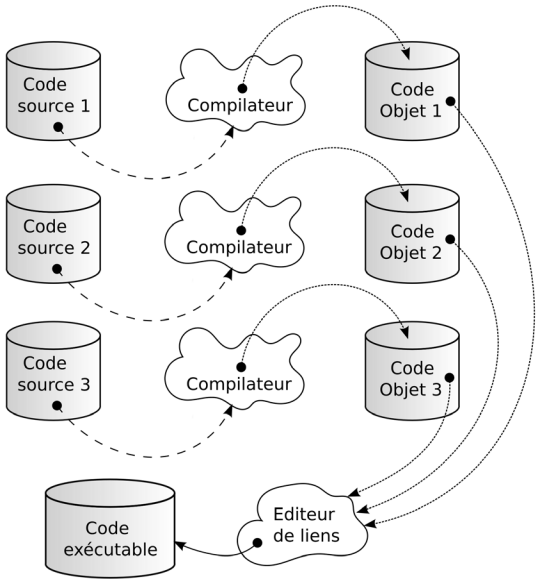


Figure 3.13 – Compilation et édition de liens d'une application répartie sur trois fichiers sources.

vous utilisez un environnement de développement intégré (ou IDE), l'appui sur un seul bouton peut réaliser toutes ces opérations. Toutefois, même dans ce cas, il est très utile de connaître l'ensemble de la chaîne de production du programme exécutable, car cela permet de comprendre l'origine des messages d'erreur auxquels vous serez forcément confrontés un jour ou l'autre.

3.2.3.2 Utilisation de bibliothèques

Il est très fréquent qu'un programme utilise des fonctions très générales, par exemple le calcul d'une racine carrée, la résolution d'un système d'équations ou le tri d'un ensemble de données. Réécrire, et mettre au point le code correspondant chaque fois qu'il est nécessaire est bien évidemment une perte de temps et d'énergie considérable. Pour cette raison, il est possible d'utiliser une version déjà compilée et testée de ce code. Celui-ci a été écrit par une tierce personne, et il est reconnu comme étant de bonne qualité.

Pour des raisons de facilité d'utilisation, ce code est fourni dans de gros fichiers nommés *bibliothèques*, qui regroupent un ensemble de *modules objets*[13]. Par exemple, les fabricants de compilateurs fournissent systématiquement des bibliothèques permettant certains calculs mathématiques ou encore des opérations de lecture ou d'écriture dans des fichiers. Bien entendu, les modules de bibliothèques doivent être incorporés aux programmes qui s'en servent pour être réellement utilisables, et ce travail revient aussi à l'éditeur de liens. Le schéma de la figure 3.14 montre par exemple la chaîne de production d'un programme exécutable comportant deux fichiers sources, et qui utilise deux modules extraits d'une bibliothèque. En C et en C++, utiliser des bibliothèques est une pratique banale, car de nombreuses fonctionnalités importantes de ces langages (comme les entrées-sorties) en font partie.

Une bibliothèque peut être *statique* ou *dynamique*. Dans le premier cas, les modules extraits de la bibliothèque sont *recopiés* dans l'exécutable. Dans le second, ces modules sont simplement *référencés* par l'exécutable. Leur chargement en mémoire a lieu lors de l'exécution. La seconde approche

13. Un module objet doit être vu comme un morceau de code déjà compilé, et dont le rôle est dûment identifié et documenté.

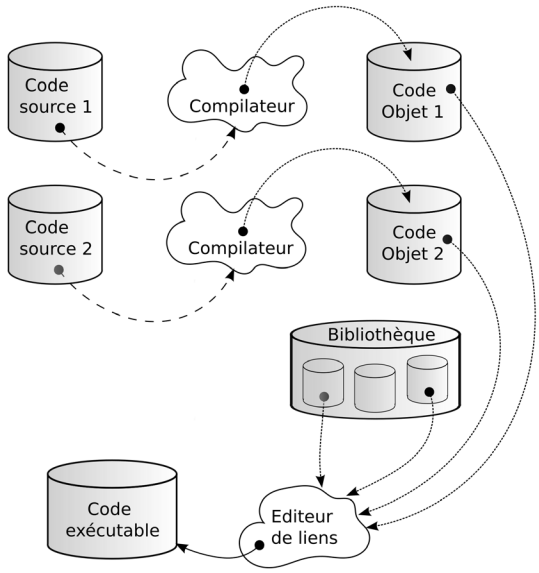


Figure 3.14 – Compilation et édition de liens d'une application répartie sur deux fichiers sources, et utilisant deux modules extraits d'une bibliothèque.

est la plus fréquente et présente les avantages suivants : (a) les exécutables sont plus petits, (b) les modules ne sont pas chargés s'ils sont déjà en cours d'utilisation par un autre programme : ils deviennent alors *partagés*, ce qui permet d'économiser beaucoup de mémoire.

3.2.4 Langages interprétés

Imaginez que vous débarquiez à l'aéroport de Shanghaï, sans connaître un seul mot de la langue locale. Étant prévoyant, vous avez emporté dans vos bagages un papier rédigé par un collègue chinois, qui y a inscrit dans sa langue natale quelque chose comme : *« Conduisez-moi à l'hôtel du Lotus Bleu, s'il vous plaît »*. Ce papier va vous permettre de vous faire comprendre du chauffeur de taxi qui, normalement, vous conduira à l'endroit voulu.
Un autre scénario de votre arrivée pourrait être que votre collègue Tchang fasse partie du voyage. Comme il comprend le français, vous pouvez lui demander en montant dans le taxi d'informer le chauffeur de votre destination.
Dans les deux cas, le résultat est le même, puisque vous rejoignez votre hôtel. Pourtant les moyens mis en œuvre pour y parvenir sont très différents. Dans le premier cas, Tchang a joué un rôle de *traducteur*, et dans le second d'*interprète*.

La même distinction existe dans le domaine des langages de programmation : pour vous faire comprendre de l'ordinateur, vous pouvez bien sûr utiliser une approche par traduction. Celle-ci est réalisée par un compilateur, comme la section 3.2.2 l'explique. Mais vous pouvez également utiliser un *interpréteur*. Un interpréteur est l'équivalent informatique de l'interprète humain : il s'agit d'un programme qui reçoit le code source de votre programme [14], l'analyse pour en comprendre le sens, et l'exécute directement, *sans générer le moindre code machine*.

Ce fonctionnement est peut-être difficile à comprendre, aussi vais-je en donner un exemple. Supposons que l'instruction à exécuter soit la suivante [15] :

14. Ceci est une simplification de ce qui se passe en réalité.
15. Il s'agit d'une instruction d'un langage imaginaire trivial.

```
afficher_sur_ecran "bonjour"
```

En mode compilé, l'affichage d'un texte sur l'écran est réalisé en fabriquant et en intégrant dans votre programme un morceau de code qui « envoie » les différents caractères du message à l'écran.

En mode interprété, ce code *existe déjà* dans l'interpréteur, car sans cela, `afficher_sur_ecran` ne serait pas une instruction valide du langage interprété. Il est donc possible d'activer ce code, immédiatement après avoir analysé (et reconnu) l'instruction d'affichage.

Il y a au moins trois différences entre les approches compilées et interprétées :

(a) Pour exécuter un programme déjà compilé, le compilateur n'est plus utile. Le programme est autonome, comme le voyageur qui dispose du papier lui permettant de rentrer à l'hôtel en taxi, aussi souvent qu'il le désire. À l'opposé, les programmes interprétés sont inutilisables sans interpréteur, de la même manière que vous dépendez de votre collègue Tchang si vous l'utilisez comme interprète durant votre séjour en Chine.

(b) Les programmes compilés sont *beaucoup* plus rapides que les programmes interprétés, car ils sont constitués d'instructions très bien agencées immédiatement compréhensibles par l'ordinateur. À l'opposé, les programmes interprétés sont exécutés au prix d'une alternance de phases d'analyse et d'exécution, car l'interpréteur doit comprendre le sens de ce que vous demandez. Et comme il n'a pas le temps de chercher la meilleure façon de procéder, la machine est généralement exploitée de manière sous-optimale.

(c) Supposons maintenant que vous voyagiez seul, et que l'hôtel du Lotus Bleu soit temporairement fermé pour travaux. Votre chauffeur de taxi le sait, mais ne peut pas vous expliquer pourquoi il ne veut pas vous y conduire. Il va donc être très difficile de résoudre le problème.

    Si en revanche vous voyagez avec Tchang, un simple échange entre lui et le chauffeur vous permettra de choisir rapidement un autre hôtel. Il en va de même des interpréteurs : ils disposent de facilités d'interaction qui rendent la mise au point des programmes interprétés plus aisée que celle des programmes compilés [16]. Heureusement, la grande majorité des compilateurs disposent d'un mode de traduction spécial (dit mode *debug*) qui permet également une mise au point facile.

Il n'y a pas de raison définitive impliquant qu'un programme écrit dans un langage donné soit exécuté dans un mode plutôt que l'autre. Par exemple, le langage Basic très en vogue dans les années 80 a été utilisé dans les deux modes. Toutefois, pour des raisons historiques et techniques, la plupart des langages ne sont disponibles que dans un seul mode. Les langages C et C++ sont par exemple utilisés en mode compilé pour des raisons d'efficacité.

La compilation des programmes est un processus très complexe, qui nécessite une machine assez puissante. Les machines actuelles conviennent tout à fait pour cela, mais ce n'était pas le cas il y a trois ou quatre décennies. C'est pourquoi le mode d'exécution interprété a perdu la majeure partie de son intérêt au fil du temps, et qu'il est surtout utilisé pour l'exécution de commandes en mode interactif.

3.2.5 Langages à machine virtuelle

Nous avons vu que le mode d'exécution interprété ne permettait pas la réalisation de programmes rapides. Il existe toutefois une variante de ce mode qui souffre moins de ce handicap : cette variante est basée sur l'utilisation d'une *machine virtuelle*.

Une machine virtuelle peut être vue comme un processeur analogue à celui qui équipe votre ordinateur : il est en particulier capable de reconnaître et d'exécuter un certain nombre d'instructions de bas niveau, et il est possible de le programmer en langage d'assemblage. Il présente toutefois une

16. Pour être convaincu de ceci, il suffit de considérer que lors de la compilation, les *symboles* définis dans le programme source disparaissent et sont remplacés par les adresses des cases mémoire correspondantes. De la même manière, il n'existe pas de lien entre le code machine et le code source qui lui a donné naissance. Ceci rend la compréhension du code machine particulièrement ardue.

originalité de taille : il n'existe pas physiquement. Plutôt que de le réaliser avec des transistors et autres composants électroniques, on a décidé de le faire *simuler* par un programme. Une machine virtuelle peut donc à la fois être considérée comme un processeur et un programme.

Si on réfléchit à la nature de ce programme, on se rend compte qu'il s'agit en fait d'un interpréteur [17]. La seule différence entre cet interpréteur et ceux dont j'ai déjà parlé est qu'il reconnaît un langage extrêmement simple, qui est celui de la machine simulée. Par conséquent, le temps requis pour l'analyse du langage traité est beaucoup plus faible que dans les interpréteurs traditionnels.

Toutefois, le langage machine de la machine virtuelle (parfois nommé *byte-code* ou *P-code*) n'est pas plus facile à utiliser par les programmeurs que celui d'une machine physique. Par conséquent, on a développé des compilateurs qui génèrent du code pour la machine virtuelle, selon une logique identique à celle du mode compilé.

La chaîne conduisant à l'exécution d'un programme basé sur une machine virtuelle est représentée par le schéma de la figure 3.15. Vous pouvez voir que ce mode d'exécution requiert à la fois un compilateur classique (qui génère du byte-code), et un interpréteur, qui est la machine virtuelle qui exécute le byte-code. Bien entendu, le compilateur n'est requis que pour le développement du programme, et non pour son exécution.

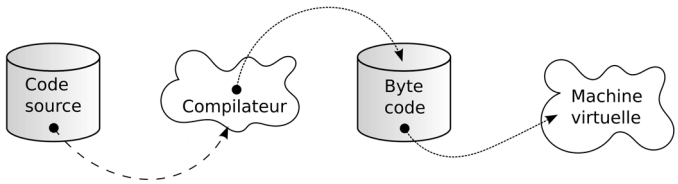


Figure 3.15 – Exécution d'un programme sur une machine virtuelle.

L'intérêt de ce mode d'exécution est que les programmes qui l'utilisent peuvent être exécutés sans recompilation sur n'importe quelle machine physique, à condition que celle-ci dispose de la machine virtuelle correspondante. En pratique, cette dernière condition est assez facile à satisfaire, car les machines virtuelles sont des programmes souvent écrits en C ou en C++ [18].

17. Une machine virtuelle est en réalité un peu plus qu'un interpréteur, car elle fournit aussi un environnement d'exécution.
18. La machine virtuelle développée pour le langage Java est par exemple écrite en C++.

4 | Les environnements de développement

4.1 Choix d'un environnement

Le choix d'un environnement de développement est important lorsqu'on étudie un langage de programmation.

Les débutants sont naturellement attirés par les environnements de développement intégrés [1] (IDE en anglais), qui semblent d'un abord facile. Malheureusement, cette apparence peut être trompeuse, car ces environnements fournissent beaucoup de fonctionnalités annexes (peu utiles à un débutant) qui nécessitent un effort pour être maîtrisées. Cet effort s'ajoute donc à celui d'apprendre le langage, et parfois à celui d'apprendre à programmer. Il est par exemple souvent arrivé qu'on me soumette des problèmes « de programmation » qui se sont révélés être de simples problèmes de configuration de projet ou d'utilisation de l'IDE.

L'autre défaut des IDE est qu'ils cachent beaucoup de détails du processus technique de développement. Or cette propriété, qui est en général utile, peut s'avérer contre-productive pour les débutants. Par exemple, certains programmeurs débutants utilisant un IDE ne savent pas ce qu'est une édition de liens, puisque celle-ci est réalisée automatiquement de façon cachée. Ceci n'est pas grave si tout se passe bien, mais lorsque ce n'est pas le cas (ce qui finit toujours par arriver) ces programmeurs auront beaucoup de mal à comprendre certains messages d'erreur, car ils ne sauront pas de quel point de la chaîne de développement ils proviennent.

Pour apprendre à programmer, ma préférence va aux environnements de type console texte, avec une interface en ligne de commande et un éditeur de texte dédié à la programmation [2] : cela permet de comprendre ce que l'on fait, et d'en apprécier immédiatement les conséquences. L'expérience montre qu'il ne faut pas plus de quelques minutes pour s'approprier un tel environnement. En outre, et contrairement à une idée répandue, cela permet de travailler très rapidement.

Il faut tout de même relativiser cette belle théorie : si votre application est (un peu) complexe, un IDE est très utile parce qu'il permet de naviguer facilement dans le code source.

Par ailleurs, le travail en mode ligne de commande est commode sous Linux ou MacOS, mais peu agréable sous Windows. Vous serez donc plus ou moins contraint d'utiliser un IDE dans cet environnement. Si vous ne disposez pas de l'IDE payant de Microsoft, je recommande d'utiliser un IDE tel que `Code::Blocks`. Celui-ci est gratuit, assez simple et de bonne qualité. Mais une meilleure alternative est d'utiliser `Creator`, l'IDE fourni avec la bibliothèque graphique Qt. Qt est également gratuit pour les applications non commerciales et constitue un outil excellent [3], que je recommande particulièrement.



Ce livre ne comporte pas de copies d'écran indiquant pas à pas les fonctionnalités à utiliser pour faire telle ou telle chose avec un IDE donné. En effet, ces informations seraient spécifiques à cet IDE, sortiraient du sujet et risqueraient d'être périmées avant d'être publiées. Vous devez vous reporter à la documentation de l'IDE en question pour les obtenir.



Pour terminer cette section, j'aimerais dissiper un doute révélé par une question que l'on me pose souvent : quel que soit l'outil de développement ou la machine que vous utilisez, le C (ou le C++) est le même. Ceci découle de la normalisation de ces langages. Par conséquent, si l'un de vos

1. Un environnement intégré est un programme graphique qui comporte un éditeur de texte perfectionné, des fonctions de navigation puissantes et un ensemble de menus/boutons qui permettent d'accéder rapidement aux actions de développement classiques : compilation, édition de liens, mise au point, etc.

2. Il existe de très nombreux éditeurs de ce type, sur toutes les plateformes.

3. Le toolkit Qt comporte de nombreuses fonctionnalités, et il est multiplateforme. Il vous permet donc d'avoir le même outil de développement (nommé `creator`) sur plates-formes Linux, Unix, MacOS et Windows.

programmes ne fonctionne plus à la suite d'un changement d'environnement, il est très probable qu'il comporte une erreur, ou qu'il utilise une fonctionnalité non normalisée.

4.2 Choix d'un compilateur

Choisir un environnement de développement n'est pas suffisant, vous devez également disposer d'un compilateur. Vous ne devez pas confondre l'environnement de développement et le compilateur, car ce sont deux outils qui n'ont pas le même rôle. Vous pouvez utiliser le même compilateur dans des environnements différents, et un environnement peut vous permettre d'utiliser des compilateurs différents.

Il existe bien entendu de nombreux compilateurs C et C++. Toutefois, le choix est assez facile à faire ici, tant un compétiteur cumule d'avantages. Je recommande d'utiliser `gcc`[4], qui fait partie de la suite d'outils du projet GNU de la Free Software Fundation. Ce compilateur est gratuit, et il est très utilisé, y compris dans le développement de très nombreuses applications commerciales. Il est capable de générer du code pour presque tous les processeurs et fonctionne sur presque tous les systèmes d'exploitation. Il fabrique un code de très bonne qualité et respecte bien les différentes normes du langage.

Une alternative à `gcc` est le compilateur gratuit `clang`, qui est d'une qualité équivalente. L'utilisation de `clang` (et `clang++`) est, dans les grandes lignes, identique à celle de `gcc` (et `g++`). Il faut aussi signaler que la version de `gcc` pour Windows s'appelle `MinGW`.

4.3 Exemples d'utilisation du compilateur `gcc`

Cette section est une très courte introduction à l'utilisation de la commande `gcc` sous Unix/Linux, dans le cadre des exemples de ce livre. Cette commande correspond en fait à un programme qui exécute au besoin le compilateur C, le compilateur C++ ou l'éditeur de liens : `gcc` est ce qu'on appelle un *front-end*[5]. Une variante de `gcc` s'appelle `g++` : il s'agit simplement d'une version de `gcc` qui a été préconfigurée pour compiler du code C++. Les informations données ici sont donc valables aussi bien pour les langages C que C++.

Cette introduction suppose que l'on travaille en ligne de commande[6]. Si vous avez des difficultés avec les termes de *compilation*, *édition de liens* , *code source*, *code objet* ou *exécutable*, lisez ou relisez les sections 3.2.2 et 3.2.3 de ce livre.

La commande `gcc` utilise la notion *d'extension de nom de fichier* pour indiquer (ou déduire) le type de données que contient ce fichier. On appelle extension un suffixe du nom qui débute par la lettre « . ». Par exemple, pour le nom de fichier « `abc.x` », l'extension est « `.x` ».

Le tableau de la figure 4.1 montre les extensions les plus couramment utilisées lorsqu'on développe en C ou C++.

4. `gcc` signifie « GNU C Compiler ».
5. `gcc` peut aussi invoquer d'autres compilateurs, mais ceci sort du cadre de ce livre.
6. Si vous utilisez un IDE, celui-ci générera les commandes lui-même.

Extension	Type du fichier correspondant
`.c`	code source en langage C
`.cpp` ou `.cc` ou `.cxx`	code source en langage C++
`.o` ou `.obj`	code objet
`.exe`	fichier exécutable Windows
pas d'extension	fichier exécutable Unix/Linux
`.dll`	bibliothèque dynamique Windows
`.lib`	bibliothèque statique Windows
`.so`	bibliothèque dynamique Unix/Linux
`.a`	bibliothèque statique Unix/Linux/MacOS
`.dylib`	bibliothèque dynamique MacOS

Figure 4.1 – Les extensions utilisées en C et C++

4.3.1 Compilation d'un seul fichier

La commande

```
gcc -c x.c
```

compile le fichier source en langage C « `x.c` ». Le fichier objet produit est « `x.o` ».
L'option « `-c` » signifie ici « **c**ompiler uniquement ».

La commande

```
g++ -c x.cpp
```

compile le fichier source en langage C++ « `x.cpp` ». Le fichier objet produit est « `x.o` ». Les commandes `gcc` et `g++` s'utilisent donc de la même manière.

4.3.2 Compilation de plusieurs fichiers

La commande

```
gcc -c x.c y.c
```

compile les fichiers sources en langage C « `x.c` » et « `y.c` » et. Les fichiers objets produits s'appelleront « `x.o` » et « `y.o` ». Vous pouvez compiler autant de fichiers que vous voulez en une seule commande.

La commande

```
g++ -c x.cpp y.cpp
```

compile les fichiers sources en langage C++ « `x.cpp` » et « `y.cpp` ». Les fichiers objets produits sont « `x.o` » et « `y.o` ».

4.3.3 Édition de liens

La commande

```
gcc -o ex1 x.o y.o
```

réalise l'édition de liens des fichiers objets « `x.o` » et « `y.o` ». Le fichier de sortie (programme exécutable) s'appellera « `ex1` ». L'option « `-o` » (comme **o**utput) permet de spécifier le nom du programme exécutable à générer. L'utilisation de `g++` est identique pour les programmes réalisés en C++.

4.3.4 Compilation et édition de liens en une seule commande

La commande

```
gcc -o ex1 x.c y.c
```

réalise la compilation des fichiers sources « `x.c` » et « `y.c` ». Cette variante de la commande crée des fichiers objets temporaires qui sont automatiquement détruits après l'édition de liens. Le fichier de sortie (programme exécutable) s'appellera « `ex1` ». L'utilisation de `g++` est identique pour les programmes réalisés en C++.

4.3.5 Compilation et édition de liens avec une bibliothèque

La commande

```
gcc -o ex1 x.c y.c -lm
```

réalise la compilation des fichiers sources « `x.c` » et « `y.c` », puis l'édition de liens comme dans la commande précédente. Les symboles indéfinis lors de l'édition de liens sont recherchés dans une bibliothèque qui s'appelle `libm`. Les noms des bibliothèques commencent toujours par `lib`, ce qui explique que la commande omette ce préfixe. Par exemple si la bibliothèque s'appelait `libABC`, l'option correspondante serait « `-lABC` » [7]. On peut référencer autant de bibliothèques qu'on le désire, en spécifiant plusieurs options « `-l` ». L'utilisation de `g++` est identique pour les programmes réalisés en C++.

Cette commande est plus commode que les précédentes, puisqu'elle permet de déclencher successivement toutes les opérations de la chaîne de production d'un programme exécutable. Elle n'est pas optimale dans le sens ou elle recompile systématiquement *tous* les fichiers sources composant l'application, même s'ils n'ont pas été modifiés depuis la dernière compilation. Ceci n'est toutefois pas pénalisant pour les petits programmes, compte tenu de la rapidité des machines actuelles. En outre, cela offre l'avantage de garantir que les fichiers objets correspondent toujours à la dernière version du code source, ce qui évite qu'une modification de ce code source ne soit pas prise en compte. Pour cette raison, je vous recommande donc d'utiliser ce genre de commande lors de vos premiers pas de programmeur.

Enfin, il est possible avec les commandes `gcc` et `g++` de choisir la version du langage que l'on désire utiliser. Pour cela, vous devez ajouter aux commandes l'option `-std=`*norme*, ou *norme* peut être :

- `c89`, `c90`, `c99`, `c11` ou `c17` pour le langage C,
- `c++98`, `c++11`, `c++14`, `c++17` ou `c++20` pour le langage C++.

Par exemple :

```
g++ -o prog main.cpp -std=c++17
```

Bien entendu, la liste des différentes normes disponibles est susceptible de changer lorsque celles-ci apparaissent ou sont supportées par le compilateur. La norme prise en compte par défaut dépend de la version du compilateur que vous utilisez : pour la version 9 de `g++`, il s'agit par exemple de C++17.

7. Attention : bien lire `-l` (lettre L minuscule), et non `-1`, (chiffre 1). Ces deux caractères sont très ressemblants.

Deuxième partie

Débuter avec le langage C

5 | Composition des programmes en langage C

5.1 La notion de fonction en programmation

5.1.1 Fonction classique

Un programme rédigé en langage C est composé de *fonctions*. Une fonction est un groupe d'instructions qui réalisent un traitement à partir de données qu'on lui fournit. L'effet de ce traitement est généralement la production d'un résultat, comme pour une fonction mathématique. Par exemple, la fonction définie en mathématiques comme

$$f(x, y) \rightarrow x + y$$

reçoit deux nombres, en calcule la somme et retourne cette valeur.

5.1.2 Fonction sans paramètre

Il peut aussi arriver qu'une fonction produise un résultat sans qu'on lui fournisse explicitement de données, comme :

$$heure() \rightarrow heureCourante$$

5.1.3 Fonction sans résultat

Par extension du concept de fonction, le langage C permet aussi de définir des fonctions qui ne produisent pas de résultat. On dit dans ce cas que leur résultat est *vide* (ou `void`). Ceci pourrait être représenté comme :

$$g(x) \rightarrow$$

Une fonction à résultat vide peut être très utile en programmation. En effet, ses instructions, (comme d'ailleurs celles de toutes les autres fonctions) peuvent produire un *effet*[1] : elles peuvent par exemple lancer une fusée vers la lune, éteindre une lampe, afficher un message ou plus simplement modifier l'état de la mémoire. La notion de fonction en programmation (et particulièrement en langage C) est donc assez différente de ce qu'elle est en mathématiques.

5.1.4 Invocation des fonctions

Bien entendu, pour qu'une fonction soit utile, il faut qu'elle soit *appelée*[2], ce qui déclenche son exécution. C'est l'interaction entre les différentes fonctions qui composent le programme qui constitue l'activité de ce programme.
Le schéma de la figure 5.1 représente un programme très simple composé de 5 fonctions. Dans ce schéma, les flèches représentent des relations d'appel : la fonction `main` peut appeler les fonctions `b` et `d`. On notera que `d` n'appelle aucune autre fonction, mais que `b` peut appeler `a` qui peut appeler `c`, qui elle-même peut appeler `d`. On remarque donc que `d` peut être appelée de façon *directe* ou *indirecte*. Remarquez aussi que `a` est susceptible de s'appeler elle-même : il s'agit d'un *appel récursif*. Ce genre d'appel est traité au chapitre 18.

5.2 La fonction `main`

Comme la totalité du code constituant un programme se trouve dans des fonctions, il est nécessaire qu'une de ces fonctions soit lancée automatiquement et appelle les autres, directement ou

1. On peut aussi utiliser le terme *effet de bord*.
2. On utilise aussi le terme *invocation* pour désigner un appel de fonction.

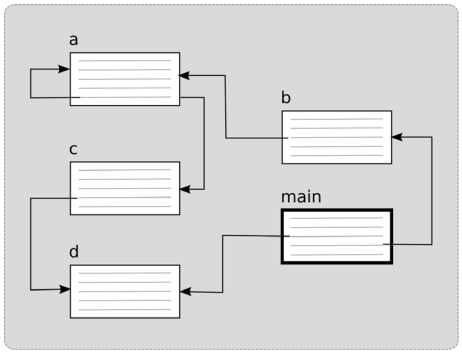


Figure 5.1 – Un programme simple composé de cinq fonctions.

indirectement. Il s'agit d'une fonction conventionnellement nommée **main**, et elle peut être la seule du programme. Voici donc ce que vous devez retenir : si un programme écrit en langage C ou C++ ne comporte pas de fonction **main**, l'éditeur de liens refusera de générer le fichier exécutable.

Toutes les fonctions **main** sont construites sur le modèle de celle qui suit. Pour l'instant, je ne donnerai pas d'autre explication sur les fonctions (qui seront traitées au chapitre 8), car nous avons beaucoup d'autres choses à voir auparavant. La fonction **main** nous servira donc uniquement à accueillir les différents exemples, au fur et à mesure de leur présentation.

NeFaitRien/ne-fait-rien.c

```
1
2 #include <stdio.h>
3 // -------------- Exemple de fonction main vide (commentaire C++)
4 int main()
5 {
6   /* Le code de la fonction doit se trouver ici (commentaire C) */
7   return 0 ;
8 }
```

La fonction **main** débute ici à la ligne 4, et se termine à la ligne 8. Elle ne comporte aucune instruction, mais les lignes 3 et 6 comportent des *commentaires*. Ceux-ci n'ont aucun effet sur l'exécution des programmes, mais ils sont extrêmement importants, car ils permettent d'éclaircir ou de rappeler un point qui rendrait le programme difficile à comprendre. N'hésitez jamais à ajouter un commentaire, même si vous êtes la seule personne à lire votre programme.
Pour l'instant, ne vous préoccupez pas du reste du code : contentez-vous de recopier cet exemple tel quel avec votre éditeur de texte favori. Enregistrez ce code source dans le fichier **ne-fait-rien.c**, puis générez le programme exécutable. Par exemple, si vous travaillez en mode texte dans une console Linux, vous devez taper :

```
gcc -o ne-fait-rien   ne-fait-rien.c
```

Exécutez maintenant le programme en tapant :

```
./ne-fait-rien
```

S'il ne se passe rien, c'est plutôt bon signe. Bravo ! Vous venez d'exécuter votre premier programme en langage C, celui qui ne fait rien.

Vous trouverez davantage des renseignements sur la fonction **main** au chapitre 12.

6 | Les données et leur type

Ce qu'on appelle *donnée* dans le contexte des langages de programmation n'est autre que la « matière » sur laquelle travaillent les programmes. C'est pourquoi, avant de parler des outils mis à notre disposition par un langage de programmation, il faut préciser les concepts en relation avec les données.

6.1 Variables et constantes

Une donnée peut être une *constante* ou une *variable*. Ces deux termes rappellent de façon claire que la valeur d'une constante ne peut pas être modifiée, contrairement à celle d'une variable. Les variables sont désignées par un nom, tel que `hauteur` ou `xflr13`. Dans le vocabulaire informatique, on trouve souvent les termes *identificateur* ou *identifiant*, comme synonymes de nom. Les constantes n'ont pas de nom : elles peuvent être des valeurs intermédiaires résultant d'un calcul, ou des *littéraux* tels que `12.5`.

6.2 Déclaration des variables

La très grande majorité des langages compilés (tels C ou C++) exigent que les variables soient déclarées avant d'être utilisées. Une déclaration ressemble à une présentation : *« Compilateur, je te présente* `x`*, c'est une variable contenant une donnée de type entier »*. Grâce à cela, le compilateur pourra entreprendre les actions aboutissant à la création de la variable. Il saura aussi comment la manipuler lorsqu'il rencontrera l'identificateur `x`.

⚠ En C, déclarer une variable ne l'initialise pas, à moins que le programmeur le demande explicitement. Vous ne devez donc jamais supposer qu'une variable possède *par défaut* une valeur particulière. En C++, on verra qu'il est possible de garantir que tous les *objets* (que vous pouvez considérer dans un premier temps comme des variables améliorées) soient initialisés.

6.3 Qu'est-ce qu'un type ?

Les variables et constantes ont un type. Si vous n'êtes pas familier avec ce concept, lisez cette section. Sinon, vous pouvez passer directement aux sections suivantes qui décrivent les différents types disponibles en langage C.

Nous avons vu dans le chapitre consacré à la structure et au fonctionnement des ordinateurs que toutes les informations stockées en mémoire sont représentées de la même manière [1]. Si l'on s'intéresse à un octet particulier pris au hasard, il est absolument impossible de dire ce qu'il représente, car il ne comporte aucune information pouvant nous renseigner.

Or, selon que cet octet contribue à représenter une adresse, un entier, un réel ou autre chose, le programme devra le manipuler de différentes façons. Par exemple, la façon de multiplier deux réels est différente de celle qui permet de multiplier deux entiers.
L'information décrivant la nature d'une entité informatique s'appelle un *type*. Connaître le type d'une entité peut être utile au moment de l'exécution du programme, mais dans le cas des langages compilés, c'est surtout lors de la compilation que le type est exploité :

(a) Le type permet au compilateur de faire en sorte que la place nécessaire et suffisante pour stocker chaque entité soit réservée. Corollairement, ceci lui permet d'associer à chaque identificateur utilisé (`x`, `y`, `masse`, `vitesse`, `prix`, etc.) une *adresse* en mémoire.

1. Avec les technologies actuelles, il s'agit d'une représentation binaire.

(b) Le type permet au compilateur de fabriquer le code machine manipulant correctement cette entité. C'est tout de même le point le plus important !

(c) Enfin, le type permet aussi au compilateur de vérifier que le programmeur utilise bien les données de façon cohérente. Par exemple, l'addition de deux nombres réels semble cohérente. Par contre, additionner un nombre réel avec un texte est probablement une erreur.


Le dernier point cité est très important : la notion de type a pour effet *d'introduire du sens* dans vos programmes, ce qui permet au compilateur d'en vérifier (dans une certaine mesure) la cohérence. Dans le jargon informatique, on utilise le terme *sémantique* pour désigner le sens attaché à un type.


La sémantique des types permet de détecter automatiquement beaucoup d'erreurs de programmation, mais ne vous faites pas d'illusions : il existe tant de façons de se tromper qu'il reste très facile d'écrire des programmes faux. Par conséquent, le fat qu'un programme ait été compilé sans messages d'erreur n'est absolument pas une garantie que ce programme soit correct, malgré l'apport du typage.

6.4 Les types primitifs

On dira qu'une entité est d'un type *fondamental* (ou *primitif simple*) si son type ne s'appuie pas sur l'existence d'un autre type. Il existe en C deux sortes de *types primitifs simples* : les *entiers*, les *réels*[2]. Il existe aussi des types *primitifs composés*, tels que les *pointeurs*, les *tableaux* et les *structures*. Le premier sera traité ici, mais les autres seront traités au chapitre 10.

Qu'elles soient d'un type simple ou composé, les entités primitives ont en commun de pouvoir être manipulées par des opérateurs prédéfinis du langage.


La sémantique des types (quels qu'ils soient) peut être définie par :

(a) le langage (c'est le cas des types primitifs),

(b) la bibliothèque standard du langage (en C++ uniquement),

(c) l'utilisateur (également en C++).

Les deux derniers points contribuent à l'extensibilité du langage.

6.4.1 Entiers

Les nombres entiers utilisés dans les ordinateurs ne représentent qu'approximativement des nombres entiers que l'on connaît en mathématiques. Leur représentation est imparfaite dans le sens où on ne sait représenter qu'un nombre fini d'entiers. Le langage C n'impose aucune représentation particulière de ces nombres, et il utilise généralement celle de la machine cible. Heureusement, presque tous les ordinateurs utilisent la même représentation, ce qui limite les surprises.
En C comme en C++, les entiers peuvent être signés (entiers relatifs) ou non signés (entiers naturels). Les entiers signés sont presque toujours représentés par un procédé nommé *complément à deux*. À partir de C++20, ce procédé est même obligatoire.

Le langage C ne propose pas moins de cinq types entiers qui s'appellent `char`, `short`, `int`, `long` et `long long`. Sans spécification explicite, ils sont tous signés, à l'exception de `char`, qui peut être signé ou non selon le compilateur. Remarquez toutefois que la plupart des compilateurs considèrent que les `char` sont signés par défaut.

Le tableau de la figure 6.1 indique les intervalles de valeurs représentables par des entiers codés sur N bits. `signed`, mais à part pour les `char`, ceci est redondant.

2. Le langage C++ fournit un troisième type primitif : le type *booléen*.

Type d'entier	Plus petite valeur	Plus grande valeur
Entiers non signés	0	$(2^N) - 1$
Entiers signés	$-(2^{N-1})$	$(2^{N-1}) - 1$

Figure 6.1 – Intervalle des valeurs représentables par des entiers codés sur N bits

Pour obtenir la version non signée de ces types, il suffit de préfixer leur nom par le mot-clé **`unsigned`**, comme dans **`unsigned long`**. La taille de l'entier résultant est la même que celle de l'entier signé correspondant. Pour mettre en évidence le fait qu'un entier est signé, vous pouvez aussi utiliser le préfixe Le langage impose quelques contraintes concernant la quantité de mémoire utilisée pour représenter les entiers : les **`char`** doivent être par définition codés sur un octet, et la taille minimale des **`int`** est de 2 octets. À part cela, les relations suivantes entre les différentes tailles doivent être vérifiées :

$$taille(\texttt{char}) \leq taille(\texttt{short}) \leq taille(\texttt{int}) \leq taille(\texttt{long}) \leq taille(\texttt{long long})$$

Pour fixer les idées, le tableau de la figure 6.2 indique les tailles typiquement utilisées, dans deux environnements classiques. Ces informations sont purement indicatives : il est bon de consulter les spécifications de votre compilateur avant d'utiliser un type plutôt qu'un autre. Cependant, il est probable que ces informations correspondent exactement à votre système.

Type C de l'entier	Ordinateur personnel	Microcontrôleur
`char`	1	1
`short` (ou **`short int`**)	2	2
`int`	4	2
`long` (ou **`long int`**)	8	4
`long long`	8	4

Figure 6.2 – Tailles typiques (en octets) des entiers supportés par les compilateurs C, dans deux environnements très différents.



Avant de passer à un exemple, voici quelques précisions importantes concernant les types **`char`** et **`int`**, qui sont très utilisés.

- Le nom **`char`** est l'abréviation de « *character* ». Le choix de ce nom est sans doute une des décisions de conception les plus funestes des auteurs du langage, car il fait invariablement penser aux débutants que ce type de donnée sert forcément à représenter des caractères. En fait, les **`char`** peuvent effectivement servir à cela, mais aussi à tout autre chose. Le **`char`** est simplement un entier, sur lequel vous pouvez faire exactement les mêmes opérations que sur les autres entiers. Sa seule spécificité est d'être toujours codé sur 8 bits [3], ce qui ne permet de représenter que 256 valeurs. Retenez donc qu'il n'y a pas de type caractère en C.
- Le type **`int`** est très souvent équivalent à **`short`** ou **`long`**, selon la machine. On peut donc se demander à quoi il sert, puisqu'il semble redondant. Ce type a été créé pour supporter de façon efficace les calculs en nombres entiers sur la machine cible. Pour cela, les variables **`int`** ont, en principe, exactement la même taille que les registres de données du processeur, ce qui assure théoriquement un fonctionnement optimal sur celui-ci. Ceci explique sa fréquente utilisation.

3. En théorie, les **`char`** pourraient cependant être codés sur plus de 8 bits, mais les architectures de processeurs exploitant cette possibilité sont extrêmement rares.