Jean-Michel Léry

Le langage C

Apprendre à programmer

Avec plus de 250 exemples et exercices corrigés

Licence
Master
Autoformation



COMPLÉMENTS
TÉLÉCHARGEABLES

ellipses

Le langage C

Apprendre à programmer

Références sciences

Licence
Master
Autoformation

Le langage C

Apprendre à programmer

Avec plus de 250 exemples et exercices corrigés

Jean-Michel Léry

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 9782340-085565
© Ellipses Édition Marketing S.A., 2024
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

Table des matières

Introduction 1

Les concepts fondamentaux 7

1. L'environnement de programmation 7

1.1. Le langage binaire et les logiciels 7
1.2. L'évolution des langages 8
1.3. Les étapes du développement 9
1.4. Les recommandations 19

2. Les outils de développement 20

2.1. Sous Windows 20
2.2. Sous Linux 22
2.3. Sous MacOS 26

3. La structure et les éléments du langage 28

3.1. Structure d'un programme C 28
3.2. Notions sur les variables 30
3.3. Notions sur les constantes 36
3.4. Notions sur les instructions 37
3.5. Les commentaires 39
3.6. Les éléments du langage 39
3.7. Les directives de compilation 41

Résumé 43

Questions de révisions 44

Exercices 45

Solutions 45

Les types simples 47

1. L'affichage et la saisie des variables 47

1.1. La procédure d'affichage *printf()* 47
1.2. La procédure de saisie *scanf()* 50
1.3. Les codages des types de données 52

2. Le type entier 52

2.1. Définition 52
2.2. Les différents types entiers 52
2.3. Représentation binaire d'un entier 54
2.4. Incidences sur la programmation 56
2.5. Déclaration des variables entières 57
2.6. Format de saisie et d'affichage 58
2.7. Syntaxe des constantes entières 61
2.8. Les opérateurs 62
2.9. Exemple 64
2.10. Les fonctions sur les entiers 65

3. Le type réel 66
3.1. Définition 66
3.2. Les différents types réels 66
3.3. Représentation binaire d'un réel 66
3.4. Incidences sur la programmation 68
3.5. Déclaration des variables réelles 68
3.6. Format de saisie et d'affichage 68
3.7. Syntaxe des constantes réelles 69
3.8. Les opérateurs 69
3.9. Les fonctions sur les réels 70
3.10. Exemples 71
4. Le type caractère 73
4.1. Définition 73
4.2. Les différents types caractères 73
4.3. Forme binaire d'un caractère 73
4.4. Les tables de codage des caractères 74
4.5. Incidences du codage sur la programmation 80
4.6. Déclaration des variables caractères 81
4.7. Format de saisie et d'affichage 81
4.8. Syntaxe des constantes caractères 81
4.9. Les opérateurs 82
4.10. Exemple 83
4.11. Les fonctions sur les caractères 83
4.12. Le caractère de validation de saisie 85
4.13. Fonctions annexes d'Entrée/Sortie 88
4.14. Incidence des *signed* et *unsigned char* 88
5. Le type booléen 89
5.1. Définition 89
5.2. Les types booléens 89
5.3. Le type entier comme support 89
5.4. Les opérateurs de comparaison 90
5.5. Les opérateurs booléens et leur table de vérité 90
5.6. Les traitements logiques 91
5.7. Les traitements binaires 94
5.8. Le type *_Bool* ou *bool* 97
6. Le type énumération 98
6.1. Qu'est-ce que le type énumération ? 98
6.2. Déclaration 98
6.3. Exemple 99
Résumé 100
Questions de révisions 100
Exercices 103
Solutions 105

Les instructions 113
1. Les instructions simples 113
1.1. L'affectation 113
1.2. L'appel de procédure 117
1.3. L'instruction *goto* 121

2. Les instructions composées..... 122
2.1. La séquence d'instructions..... 122
2.2. Les instructions conditionnelles..... 123
2.3. Les instructions répétitives..... 130
2.4. Les instructions de contrôle..... 146
Résumé..... 147
Questions de révisions..... 148
Exercices..... 149
Solutions..... 153

Les types structurés..... 179
1. Les tableaux..... 179
1.1. Présentation..... 179
1.2. Déclaration..... 180
1.3. Représentation mémoire..... 180
1.4. Accès à un élément du tableau..... 181
1.5. Format de saisie et d'affichage..... 182
1.6. Types de tableaux..... 182
1.7. Traitement d'un tableau..... 183
1.8. Tableau et pointeur..... 194
2. Les chaînes de caractères..... 196
2.1. Présentation..... 196
2.2. Le type chaîne de caractères..... 196
2.3. Représentation mémoire..... 196
2.4. Déclaration des variables..... 197
2.5. Format de saisie et d'affichage..... 197
2.6. Les constantes..... 198
2.7. Les autres fonctions d'Entrée/Sortie..... 198
2.8. Les fonctions de manipulation de chaînes..... 200
2.9. Les méthodes de gestion de chaînes..... 203
2.10. Tableau de chaînes..... 207
3. Les structures..... 210
3.1. Présentation..... 210
3.2. Déclaration de structures..... 210
3.3. Déclaration de type..... 211
3.4. Représentation mémoire..... 211
3.5. Format de saisie et d'affichage..... 212
3.6. Accès à un élément de la structure..... 212
3.7. Traitement d'une structure..... 212
3.8. Champs binaires..... 214
3.9. Tableau de structures..... 216
3.10. Structures imbriquées..... 218
3.11. La définition de type par *typedef*..... 219
4. Les unions..... 221
4.1. Présentation..... 221
4.2. Déclaration d'une union..... 222
4.3. Représentation mémoire..... 222
4.4. Exemple..... 222

5. Les fichiers 224
5.1. Rôle d'un fichier 224
5.2. Principe d'utilisation 224
5.3. Nature des fichiers 225
5.4. Méthodes d'accès 227
5.5. Déclaration d'un fichier 228
5.6. Relation entre une variable fichier et le fichier sur disque 229
5.7. Les fichiers textes 230
5.8. Les fichiers binaires 235
6. Les nombres complexes 240
6.1. Présentation 240
6.2. Déclaration des variables complexes 242
6.3. Format de saisie et d'affichage 243
6.4. Affectation des parties réelles et imaginaire 244
6.5. Syntaxe des constantes 244
6.6. Les opérateurs arithmétiques 246
6.7. Les fonctions 247
Résumé 250
Questions de révisions 250
Exercices 253
Solutions 258

La programmation structurée 283
1. Les procédures et les fonctions 283
1.1. Présentation 283
1.2. Syntaxe 284
1.3. Création d'une procédure 285
1.4. Déclaration préliminaire 287
1.5. Contraintes syntaxiques 289
1.6. Passage des paramètres 291
1.7. Création d'une fonction 295
1.8. Retour d'une fonction 296
1.9. Paramètres évolués 298
1.10. Variable locale, globale, statique 305
1.11. Le mot-clef *inline* 312
1.12. Le mot-clef *_Noreturn* 313
1.13. Classes des variables et de fonctions 314
2. La récursivité 316
2.1. Principe 316
2.2. Description 316
2.3. Exemple 318
3. La généricité 319
3.1. Principe 319
3.2. Description 320
3.3. Exemples 320
4. La compilation séparée et les librairies 321
4.1. Compilation séparée 321
4.2. Construction d'une librairie 325

5. Le préprocesseur C 329
5.1. Qu'est-ce qu'une directive de compilation ? 329
5.2. Les directives de définition 329
5.3. La directive *#include* 332
5.4. Les directives conditionnelles 332
5.5. Les directives dédiées au compilateur 335
5.6. Les options du compilateur 337
Résumé 339
Questions de révisions 340
Exercices 341
Solutions 345

La gestion dynamique de la mémoire 387

1. Les pointeurs 387
1.1. Présentation 387
1.2. Déclaration 388
1.3. Format d'affichage 388
1.4. Utilisation d'un pointeur 388
1.5. Allocation dynamique de la mémoire 389
1.6. Les opérateurs 392
1.7. Gestion de blocs de mémoire 393
1.8. Tableau ou pointeur ? 394
1.9. Pointeur sur *void* 397
1.10. Pointeur de fonction 397
1.11. Tableau de pointeurs 399
1.12. Pointeur de pointeur 401
2. Les tableaux dynamiques 403
2.1. Taille limite des tableaux statiques 403
2.2. Allocation dynamique d'un tableau 405
2.3. Allocation dynamique case par case 407
3. Les listes chaînées 409
3.1. Tableaux ou liste chaînées ? 409
3.2. Principe de la liste 411
3.3. Gestion de la liste 412
3.4. L'opérateur `->` 415
3.5. Exemple 415
Résumé 417
Questions de révisions 418
Exercices 419
Solutions 421

La programmation système 443

1. L'interface avec le système d'exploitation 443
1.1. Présentation 443
1.2. Arguments de la ligne de commande 443
1.3. Gestion des erreurs 444
1.4. Exécution d'une commande UNIX 449
1.5. Les variables d'environnement 451

2. La programmation des signaux 455
2.1. Notions sur les signaux 455
2.2. Description des fonctions 456
2.3. Exemple 458
3. La programmation des processus 461
3.1. Notions sur les processus 461
3.2. Description des fonctions 461
3.3. Exemple 462
4. La date et l'heure du système 465
4.1. Représentation 465
4.2. La structure *tm* 465
4.3. Description des fonctions 466
4.4. Exemple 469
5. La gestion des fichiers et des répertoires 470
5.1. Le système de fichiers 470
5.2. Description des fonctions 470
5.3. Exemple 474
6. La fonction aléatoire 478
6.1. Principe 478
6.2. Description des fonctions 478
6.3. Exemple 479
Résumé 480
Questions de révisions 480
Exercices 481
Solutions 481

Index 485

Introduction

Ce livre offre une approche très pédagogique du langage C, à partir d'exemples et d'exercices corrigés. L'expérience de l'enseignement montre que les concepts fondamentaux sont mieux assimilés quand ils sont accompagnés d'exemples qui mettent en pratique les techniques décrites, et d'exercices qui incitent à une réflexion personnelle sur l'utilisation de ces techniques. C'est pourquoi une place très importante est également accordée aux programmes C.

Cet ouvrage n'est pas pour autant un recueil d'exercices, mais bien une présentation du langage C, accompagnée de la mise en pratique de ce langage. Les étudiants y trouveront la description complète du langage et des grands processus de programmation comme les boucles, ainsi que des schémas explicatifs. Même si le public visé est celui des étudiants en informatique, cet ouvrage conviendra parfaitement aux débutants ainsi qu'aux programmeurs confirmés qui désirent approfondir leurs connaissances en C. Les enseignants y trouveront des exercices et des schémas, ce qui fait de cet ouvrage un excellent support de cours.

Bref rappel historique

Le langage C a été inventé en 1972 par *Denis Ritchie* dans les *laboratoires Bell.* Il fait suite au langage B développé par *Ken Thompson.*

Le langage C a été utilisé en 1973 pour réécrire le système UNIX développé à la même époque sur mini-ordinateur PDP-7 par *Ken Thompson* et *Denis Ritchie*. Cette réécriture d'UNIX dans un langage évolué a permis son déploiement sur d'autres ordinateurs.

Par la suite, *Brian Kernigham* aida au développement et à la popularisation du langage C, via la parution en 1978 du livre qu'il a co-écrit avec *Denis Ritchie* : « *The C Programming Language* ».

Le langage a ensuite été normalisé par l'organisme *ANSI* (*American National Standards Institute*) aboutissant en 1989 à la première norme *ANSI C* ou *C89*.

Cette norme a également été adoptée par l'organisation internationale de normalisation *ISO* (*International Standards Organisation*) en 1990 sous le nom de *C ISO* ou *C90*. Quelques correctifs et améliorations ont ensuite produit les versions *C94/C95.* En 1999 la version *C99* apporte des nouveautés tels que les tableaux de

taille variable, les nombres complexes, les fonctions *inline* ou encore les commentaires du C++. En 2011 la version *C11* ajoute le support à la programmation *multi-thread* et à *Unicode*. En 2018 la version *C17/C18* apporte quelques correctifs. Une nouvelle version C23 est en cours de préparation.

Atouts du langage C

Bien qu'assez ancien, le langage C reste de nos jours le langage le plus rapide en temps d'exécution.

Il est à la fois évolué et structuré, ce qui en fait un excellent support pour l'apprentissage de la programmation en général. Il est également utilisé dans le monde scientifique ou industriel.

Enfin, le langage C a inspiré les concepteurs d'autres langages comme le C++, le PHP ou le Java. Certaines syntaxes du langage C, comme les boucles ou les tests, se retrouvent dans ces autres langages. Apprendre à programmer avec le langage C facilite l'apprentissage de ces autres langages.

Programmation procédurale

Le langage C est de type *procédural*. Cela signifie que l'écriture d'un programme consiste à écrire les traitements qui s'appliquent aux données. Les traitements sont généralement regroupés en sous-programmes appelés *procédures* ou *fonctions*, d'où le nom de ce type de programmation.

Cette programmation s'oppose à la *programmation orientée objet* ou *POO*. Dans celle-ci les traitements et les données ne sont pas séparés. Ainsi un objet contient à la fois ses données et ses traitements.

Il est usuel dans l'apprentissage de la programmation de commencer par un langage procédural pour apprendre les grands mécanismes de programmation ainsi que les méthodes de traitements comme les boucles, les tests, les appels de procédures, avant d'appréhender la programmation orientée objet.

Langage compilé

Le langage C est un langage *compilé*. Cela signifie qu'il faut d'abord écrire un fichier texte dans ce langage, le fichier *source*, contenant tous les traitements que l'on souhaite faire exécuter à l'ordinateur. Ce fichier est ensuite traduit dans le langage *binaire* de l'ordinateur, produisant ainsi le fichier *exécutable*. Seul ce dernier est compris de l'ordinateur. C'est le *logiciel*.

Les langages compilés nécessitent deux étapes distinctes : l'écriture du programme source (le fichier texte) ; la compilation qui produit le fichier exécutable.

Cela s'oppose aux langages *interprétés* comme le PHP, pour lesquels on n'effectue que la première étape : l'écriture du fichier texte source. Le fichier source n'étant pas écrit dans le langage binaire de l'ordinateur, il ne peut pas être exécuté directement. Pour cela, on utilise un logiciel qui va interpréter les syntaxes qu'il contient et indiquer à l'ordinateur les actions à effectuer. Cette étape d'interprétation se déroule à chaque nouvelle exécution du programme.

Ce n'est pas le cas des langages compilés pour lesquels la traduction en fichier exécutable (langage binaire de l'ordinateur) n'est effectuée qu'une seule fois, lors de la compilation. L'exécution est alors plus rapide.

Public concerné

Le langage C – Apprendre à programmer s'adresse aussi bien aux débutants qu'aux utilisateurs avertis. Toutefois, il est souhaitable d'avoir un minimum de connaissances en informatique, comme par exemple, savoir installer son ordinateur personnel.

Conventions typographiques

L'emploi de polices différentes permet de distinguer les syntaxes C ou de faire ressortir un nouveau terme ou un nouveau concept.

- Les instructions du langage C sont dans une police à espacement fixe, par exemple `main()`.
- Les termes nouveaux ou importants sont en italique, par exemple *compilation*.
- Les syntaxes C nouvelles ou importantes sont en police à espacement fixe et en gras comme la syntaxe suivante : **`car=getchar() ;`**

Des encadrés apportent des informations complémentaires au fil des sections et des développements théoriques ou pratiques comme l'encadré « à noter » ci-dessous.

À NOTER

Cette rubrique attire l'attention du lecteur sur des points intéressants.

L'environnement de programmation

Les exemples et exercices corrigés fournis dans cet ouvrage sont développés en respectant le langage C normalisé. Ils sont exploitables dans n'importe quel environnement de programmation.

Le chapitre 1 détaille les outils de programmation sous Windows, Linux ou MacOS. Les chapitres suivants utilisent les syntaxes de compilation et d'exécution

communes à UNIX (Linux ou MacOS). Les menus de compilation et d'exécution sous Windows, détaillés dans le chapitre 1, ne sont pas reproduits à chaque exemple de programme, mais ils s'appliquent parfaitement à tous les programmes.

Les exemples

Leur objectif est double :

- Montrer l'utilisation des syntaxes vues dans le chapitre.
- Montrer comment appliquer les concepts énoncés, dans un programme plus complet. Les exemples sont alors plus conséquents.

Cet ouvrage est conçu pour faire évoluer le lecteur vers un niveau de programmation de plus en plus complexe.

Les exemples du début sont volontairement simples et pourront sembler sans rapport les uns avec les autres. Ils permettent en réalité de constituer une collection de techniques de programmation de base sur les différentes données élémentaires (entiers, réels, caractères…), utilisées ensuite avec les données structurées.

Dès que le niveau de programmation sera suffisant et que les grands mécanismes auront été abordés (tests, boucles, sous-programmes), un exemple plus poussé est utilisé comme fil conducteur. Il est enrichi au fur et à mesure de l'apprentissage de nouvelles techniques.

Chaque programme est suivi par le résultat de son exécution. Les commandes tapées à ce niveau sont précédées par l'invite système (`$`).

À NOTER

Le caractère `$` est la forme courante de l'invite dans le système UNIX ou Linux. Tous les exemples ont été développés dans cet environnement.

Le programme `premier_prog.c` présente cette convention typographique.

premier_prog.c

```
/* Mon premier programme */
#include <stdio.h>
int main()
{
 printf("Hello World\n");
}
```

Une fois traduit en binaire exécutable (compilé), l'exécution de ce programme produit le résultat suivant :

```
$ ./premier_prog
Hello World
```

Les explications nécessaires à la compréhension de ce premier programme C, ainsi que les commandes de compilation (traduction en binaire exécutable) sont fournies dans les chapitres suivants. Il ne s'agit ici que d'une présentation typographique.

Questions de révision

Les questions de révision permettent au lecteur de vérifier sa compréhension des concepts abordés dans le chapitre. Cela fait partie des méthodes de travail de nombreux enseignants que de poser des questions ciblées à la fin d'un cours pour vérifier l'assimilation des points essentiels.

Les exercices corrigés

Les exemples montrent l'utilisation des techniques de programmation, mais rien ne remplace l'apprentissage par la manipulation. L'écriture d'un même programme diffère d'un développeur à l'autre. Les erreurs des uns ne sont pas celles des autres. Un bon apprentissage consiste à apprendre « de ses erreurs ».

À la fin de chaque chapitre nous proposons des exercices qui reprennent les points traités. S'il n'est pas indispensable de réaliser tous les exercices, il est toutefois conseillé de s'atteler à quelques-uns, car rien ne remplace l'apprentissage par l'expérience.

Contenu de l'ouvrage

L'apprentissage d'un langage consiste tout d'abord à assimiler les différents types de données (les types simples et les types structurés), puis à étudier les instructions qui les manipulent (instructions simples et composées).

L'étude des données structurées nécessite la connaissance des boucles (instructions composées). L'ordre logique qui en découle est d'aborder en premier les types simples, puis les instructions, et en dernier les données structurées.

Cet ouvrage est structuré en sept chapitres.

- **Chapitre 1** : **Les concepts fondamentaux**. Il décrit l'environnement de programmation (outils et principes) et la structure du langage.
- **Chapitre 2** : **Les types simples**. Il présente les entiers, les caractères, les réels, les booléens et les énumérations.
- **Chapitre 3** : **Les instructions**. Ce chapitre porte sur les instructions simples et surtout sur les instructions composées comme les boucles qui représentent un des grands mécanismes de la programmation.
- **Chapitre 4** : **Les types structurés**. Les types étudiés sont : les tableaux, les chaînes de caractères, les structures, les unions, les fichiers et les complexes. Ils permettent d'organiser les données plus complexes.
- **Chapitre 5** : **La programmation structurée**. Ce chapitre présente les sous-programmes qui structurent le programme en modules autonomes, ainsi que la récursivité, la compilation séparée et les librairies, et enfin le préprocesseur C.

- **Chapitre 6** : **La gestion dynamique de la mémoire**. Ce chapitre porte sur les pointeurs, l'allocation dynamique de la mémoire, les tableaux dynamiques et les listes chaînées.
- **Chapitre 7** : **La programmation système**. Ce chapitre présente comment interagir avec le système d'exploitation. Il présente des traitements comme la récupération des arguments de la ligne de commande, la gestion des signaux et des processus, la récupération de la date et de l'heure système ainsi que des informations sur les fichiers et les répertoires. Il présente également la génération d'un nombre aléatoire basé sur l'horloge de l'ordinateur.

Les suppléments

Le site **http://jeanmichel.lery.free.fr** propose le téléchargement des fichiers sources de cet ouvrage au format « .c », ainsi que des chapitres complémentaires.

Le mot de passe est : « **LangageC** ».

Remerciements

Je tiens à remercier Sylvie, Guillaume et Julien.

1

Les concepts fondamentaux

PLAN

1. L'environnement de programmation
2. Les outils de développement
3. La structure et les éléments du langage

Un langage de programmation est un support syntaxique qui permet d'indiquer à l'ordinateur une suite d'instructions à exécuter. Cette série d'instructions est écrite dans un programme qu'il faut ensuite traduire dans le langage de l'ordinateur. Dans ce chapitre, nous présentons les étapes et les outils de développement qui aboutissent à l'écriture du programme. Nous y étudions également les spécificités du langage C.

1. L'environnement de programmation

1.1. Le langage binaire et les logiciels

Les éléments de l'ordinateur (processeur, mémoire centrale, etc.) sont constitués de circuits logiques basés sur des *transistors* qui gèrent des valeurs numériques 0 ou 1. Ces deux valeurs constituent la base 2, ou *base binaire*. Le langage de l'ordinateur est composé d'une suite de 0 et de 1, c'est le *langage binaire*.

Chaque processeur a été créé pour effectuer une série d'actions élémentaires comme les additions, les soustractions, l'affichage d'un caractère, etc. Chacune de ces actions est une *instruction* écrite dans son langage binaire. Cet ensemble d'instructions se nomme le *jeu d'instructions*.

L'évolution des processeurs produit de nouvelles puces capables d'effectuer des calculs plus complexes. Leur *jeu d'instructions* s'étoffe, grandit, et diffère par conséquent de celui de ses prédécesseurs. Chaque processeur possède un langage binaire qui lui est propre.

Il existe une certaine compatibilité entre les processeurs d'un même constructeur. Le dernier de la gamme comprend le langage binaire de son prédécesseur, mais

l'inverse n'est pas vrai. La *compatibilité* est dite « ascendante ». Ainsi, un logiciel écrit dans le langage d'un processeur ancien sera compris par les processeurs récents du même constructeur, mais non par ceux d'un autre constructeur.

Le logiciel *exécutable* (binaire compris par le processeur) n'est pas *portable*. Il peut être transposé d'un ordinateur à un autre seulement si ces deux machines possèdent des processeurs et des systèmes d'exploitation compatibles. C'est notamment le cas des processeurs Intel et AMD, qui équipent les PC.

Afin de vendre leurs logiciels au plus grand nombre, les éditeurs doivent choisir un langage binaire compréhensible aussi bien par les ordinateurs récents que par les machines anciennes qui sont toujours en service. L'utilisation du langage binaire d'un processeur relativement ancien, compatible avec les plus récents, est donc un gage de bon fonctionnement, et de vente sur l'ensemble du parc informatique installé. Seuls les logiciels dont la performance peut être nettement améliorée par l'emploi des nouvelles instructions (celles des processeurs récents) seront réécrits dans les nouveaux langages binaires. C'est le cas de certains logiciels de traitement d'images ou multimédias.

À NOTER

Un *logiciel* est une suite logique d'instructions écrites dans le langage du processeur.

Le logiciel *exécutable* (binaire compris par le processeur) n'est pas *portable*. Il est transportable d'un ordinateur à un autre si ces deux machines possèdent des processeurs, des architectures matérielles et des systèmes d'exploitation identiques ou compatibles.

1.2. L'évolution des langages

Ce problème de non-portabilité des logiciels écrits en binaire est le point de départ de la création des *langages de programmation*. Le principe retenu consiste à utiliser un langage plus proche du langage humain que du langage binaire, et indépendant de l'ordinateur.

Le langage assembleur

La première évolution fut de nommer les instructions binaires avec des mnémoniques (mots courts représentant un codage) plus explicites pour le programmeur, ce qui a donné naissance au *langage assembleur*. Le programme écrit en assembleur n'est pas interprétable par le processeur, c'est du texte ; il faut le traduire en binaire *exécutable*. Un logiciel particulier effectue la traduction : le *compilateur*. Il existe autant de compilateurs qu'il y a de couples langage de programmation/langage binaire. Cette première évolution était insuffisante, car il existe autant d'assembleurs que de processeurs. Voici un exemple de la somme de 2 et 3 écrite en assembleur.

« Assembleur »

```
mov ax,2
add ax,3
```

Les langages évolués

L'étape suivante a consisté à définir des langages de programmation *évolués* totalement indépendants de l'ordinateur. Le programmeur écrit son application dans ce langage, puis se sert d'un logiciel particulier, un *compilateur*, pour traduire son programme en un fichier binaire exécutable. Ce langage étant indépendant du processeur, le *fichier source* écrit dans le langage évolué est totalement portable d'un ordinateur à l'autre. Une fois installé sur le nouvel ordinateur, il suffit de le traduire dans le langage binaire de cette machine grâce au compilateur (figure 1.1).

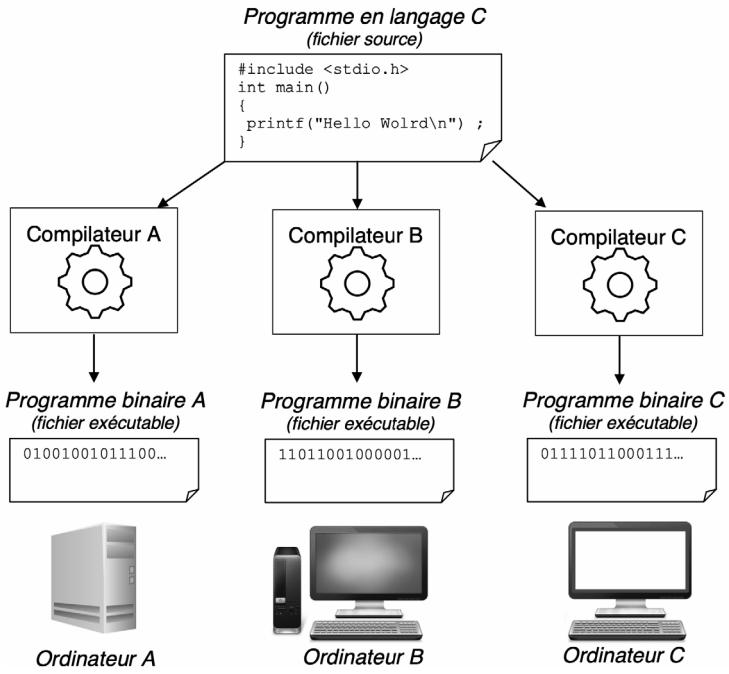


Figure 1.1

Portabilité d'un programme source.

Différents langages de programmation évolués ont vu le jour selon les besoins. Certains sont plus adaptés au développement scientifique, comme le FORTRAN ; d'autres permettent des développements pointus dans les domaines du système ou du réseau, comme le C ; enfin, certains ont été conçus pour faciliter l'écriture de programmes dans l'environnement Internet, comme les langages JAVA ou PHP.

1.3. Les étapes du développement

L'écriture d'un programme n'est que l'étape finale d'un développement qui se déroule en trois phases :

- **L'analyse**. C'est une étape de réflexion sur le ou les problèmes à résoudre. Elle conditionne les traitements futurs.

- **L'algorithme**. C'est l'étape organisationnelle. Les solutions préconisées à l'étape précédente doivent être organisées logiquement.
- **La programmation**. C'est l'étape technique d'écriture dans un langage de programmation.

Cet enchaînement est présenté dans la figure 1.2.

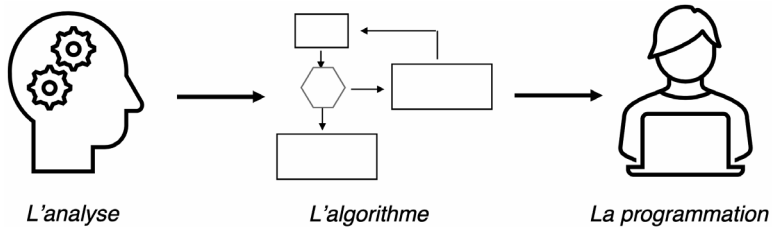


Figure 1.2
Les étapes du développement.

L'analyse

Le but de tout développement est de fournir une solution informatique à un problème donné. Il peut s'agir de la gestion d'un stock de magasin, ou du traitement d'une image satellite. Pour tous ces problèmes, il est nécessaire de cerner le sujet avant d'y apporter une solution. C'est *l'analyse du problème*.

Il existe des méthodes d'analyse, comme MERISE. Elle analyse complètement un problème donné, mais nécessite une mise en œuvre assez lourde. Il est possible d'employer des méthodes plus rapides et adaptées à des problèmes plus simples.

À NOTER

L'analyse propose des solutions qui seront programmées. Une mauvaise analyse produira un programme médiocre ou faux, quelle que soit la qualité de la programmation.

Voici un exemple d'analyse qui évolue au fur et à mesure des questions soulevées, et qui produit à la fin un programme très différent dans sa complexité.

Prenons le problème simple de l'interprétation du numéro de sécurité sociale (sans la clef finale). Le découpage de ce numéro est présenté à la figure 1.3.

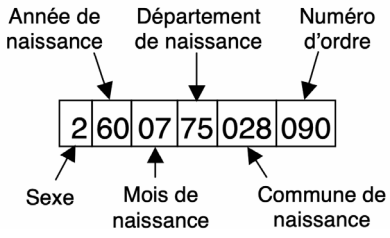


Figure 1.3
Interprétation d'un numéro de sécurité sociale.

Une analyse sommaire permet de faire les constats suivants :

- Il s'agit d'un nombre entier de 13 chiffres (sans la clef finale).
- Le premier chiffre indique le sexe (1=Masculin, 2=Féminin).
- Les deux chiffres suivants représentent l'année de naissance.
- Etc.

À partir de l'exemple de la figure 1.3 on en déduit qu'il s'agit :

- d'une femme ;
- née en 1960 ;
- au mois de juillet ;
- à Paris.

Or, l'interprétation du quatrième élément de cet exemple est faux ! Cette personne est née à Créteil dans le département de la Seine qui en 1960 regroupait Paris et les communes limitrophes (Val de Marne, Val d'Oise, …). Il faut donc pousser plus loin l'analyse en intégrant un volet historique et se poser de nouvelles questions :

- Comment a évolué le découpage des départements (Île-de-France, Corse…) ?
- Qu'en est-il des personnes nées dans les anciennes colonies françaises ?
- Comment est réglé le problème des centenaires ?

D'autre part, nous avons supposé que les valeurs du champ sexe sont : 1 (homme) et 2 (femme). Il faut se poser la question de l'existence d'autres valeurs numériques pour homme et femme car il s'agit d'une simple convention de codage.

Enfin l'analyse des numéros de départements montre qu'ils ne sont pas tous constitués de chiffres : les départements corses sont numérotés 2A et 2B. Cette dernière remarque peut changer tout le problème. L'information à traiter qui semblait numérique est peut-être une chaîne de caractères ? Or le traitement d'une chaîne diffère de manière importante du traitement d'un numéro. De même certains numéros de département sont sur 3 chiffres !

L'analyse simpliste du début aurait conduit à écrire un programme ne traitant qu'une partie de la problématique. Le problème est en réalité plus complexe qu'il n'y paraît. L'erreur qu'il ne faut pas commettre lors de l'analyse est d'apporter des solutions basées sur ses certitudes, avant de se poser toutes les questions.

L'algorithme

L'analyse met « à plat » le problème posé et propose des solutions. L'étape suivante consiste à organiser ces solutions, c'est *l'algorithme*.

Il peut être présenté sous la forme de diagrammes ou bien écrit dans un langage de programmation, car c'est déjà un programme ! Sa présentation au travers d'un langage de programmation a l'inconvénient de spécialiser l'écriture des solutions par l'utilisation d'une syntaxe particulière. L'utilisation d'un pseudo-langage est mieux adaptée car la syntaxe y est générique et simplifiée.

À NOTER

Le rôle de l'algorithme est avant tout d'organiser et de structurer l'ensemble des solutions pour aboutir à la solution globale. Il doit présenter la logique aboutissant au résultat final et être dissocié de l'aspect technique des langages informatiques. Une fois écrit, l'algorithme peut être traduit dans n'importe quel langage informatique.

Voici un exemple d'algorithme basé sur l'analyse précédente écrit en pseudo-langage. Seule une petite partie du problème analysé est présentée.

« Pseudo-Code »

```
Programme  Num_Sec_Soc
Déclarations
   Variables  Num_Sec,Libellé,Nom_Mois  en Chaînes_de_Caractères
   Variables  Sexe,Année,Mois,Département,Commune,Num_Ordre en Entier
Début
  Écrire ("Entrez votre numéro de Sécurité sociale :")
  Lire(Num_Sec)
  Sexe ← Conversion_En_Entier(sous_chaîne(Num_Sec,1,1)
  Si (Sexe = 1) alors
     Libellé ← "Monsieur"
  Sinon
     Libellé ← "Madame"
  Finsi
  Année ← Conversion_En_Entier(sous_chaîne(Num_Sec,2,2))
  Année ← Année + 1900
  Mois  ← Conversion_En_Entier(sous_chaîne(Num_Sec,4,2))
  Selon  Mois  Faire
    Cas 1  : Nom_Mois ← "Janvier"
    Cas 2  : Nom_Mois ← "Février"
    Cas 3  : Nom_Mois ← "Mars"
    Cas 4  : Nom_Mois ← "Avril"
    Cas 5  : Nom_Mois ← "Mai"
    Cas 6  : Nom_Mois ← "Juin"
    Cas 7  : Nom_Mois ← "Juillet"
    Cas 8  : Nom_Mois ← "Août"
    Cas 9  : Nom_Mois ← "Septembre"
    Cas 10 : Nom_Mois ← "Octobre"
    Cas 11 : Nom_Mois ← "Novembre"
    Cas 12 : Nom_Mois ← "Décembre"
  FinSelon
  Département ← Conversion_En_Entier(sous_chaîne(Num_Sec,6,2))
  Commune     ← Conversion_En_Entier(sous_chaîne(Num_Sec,6,5))
  Num-Ordre   ← Conversion_En_Entier(sous_chaîne(Num_Sec,11,3))
  Écrire ("Bonjour : ", Libellé)
  Écrire ("Vous êtes né en : ", Année)
  Écrire ("Au mois de : ", Nom_Mois)
  Écrire ("Dans le département : ", Département)
  Écrire ("Dans la commune : ", Commune)
  Écrire ("Avec le numéro d'ordre : ", Num_Ordre)
Fin
```

L'écriture et la compilation d'un programme

L'algorithme est déjà le programme. Pour produire le logiciel exécutable il faut :

1. Écrire le *programme source*.

   Il s'agit d'écrire un fichier texte contenant la traduction en C (ou un autre langage) de l'algorithme. Cette phase se nomme *l'édition de texte*.

   Le *fichier source* se nommera par exemple `prog.c`.

2. Produire le *programme objet*.

   Cette étape traduit en binaire le texte précédent grâce à un compilateur.

   C'est l'étape de *compilation*.

   Le *fichier objet* se nommera `prog.o` (sous UNIX/Linux ou MacOS) ou `prog.obj` (sous Windows).

3. Produire le *programme exécutable*.

   Le programme compilé précédent n'est pas encore exécutable.

   En effet, il est possible d'utiliser des sous-programmes déjà traduits en binaire, ce qui évite de tout réécrire. La phase de compilation précédente ne traduit pas ces sous-programmes qui sont stockées sous leur forme binaire dans des fichiers « à tiroirs » qui se nomment des *librairies*. Le fichier résultant de la compilation est incomplet, le code binaire correspondant aux sous-programmes est absent.

   La complétion du fichier objet avec les codes binaires des sous-programmes produit le fichier exécutable, qui est le *logiciel*.

   Cette étape de complétion se nomme *l'édition des liens*.

   Le *fichier exécutable* se nommera alors `prog` (sous UNIX/Linux ou MacOS) ou `prog.exe` (sous Windows).

La figure 1.4 présente ces différentes étapes.

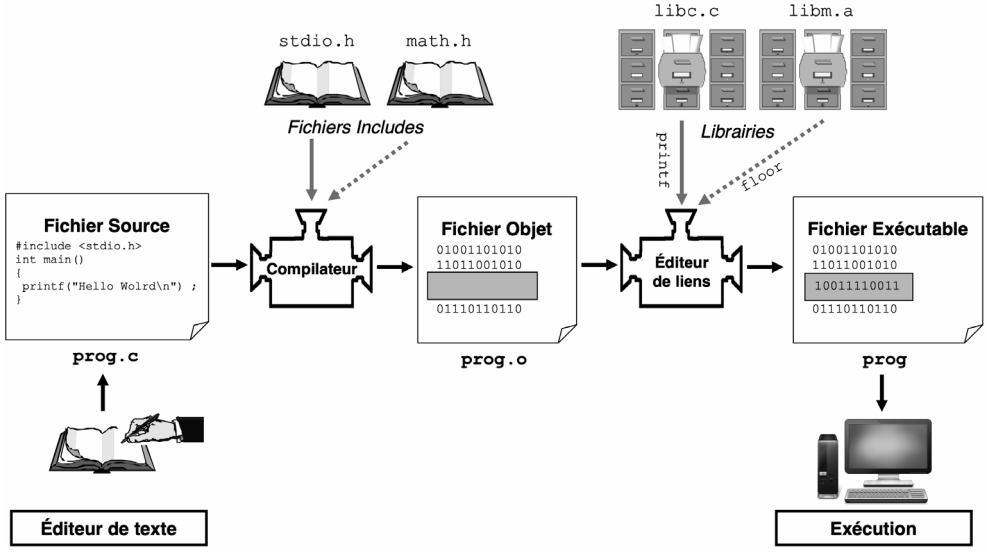


Figure 1.4

L'écriture et la compilation d'un programme.

Nous présentons dans la section 2, les outils de développement dans les trois environnements systèmes : Windows, Linux et MacOS.

L'écriture du programme

Cette phase consiste à écrire un fichier de texte, le *fichier source*, contenant le programme C, via un éditeur de texte (voir section 2).

Le fichier source C `numsecsoc.c` correspond à l'algorithme précédemment.

numsecsoc.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{/* --- Déclarations des variables --- */
 char Num_Sec[14], Libelle[9], Nom_Mois[10], Dept[3], Com[6] ;
 int Sexe, Annee, Mois, Num_Ordre, Departement, Commune ;

 /* --- Instructions --- */
 /* Saisie du numéro sous la forme d'une chaîne */
 printf("Entrez votre numéro de Sécurité sociale : ") ;
 scanf("%s",Num_Sec) ;
 /* Récupération du numéro 1 ou 2 du Sexe */
 Sexe = Num_Sec[0]-48 ;
 /* Affectation du Libellé */
 if (Sexe == 1)
   strcpy(Libelle,"Monsieur") ;
 else
   strcpy(Libelle,"Madame") ;
 /* Récupération de l'année de naissance */
 Annee = ((Num_Sec[1]-48)*10) + (Num_Sec[2]-48) ;
 Annee = Annee + 1900 ;
 /* Récupération du mois de naissance */
 Mois = ((Num_Sec[3]-48)*10) + (Num_Sec[4]-48) ;
 switch (Mois)
 {
   case 1  : strcpy(Nom_Mois,"Janvier")  ;
             break ;
   case 2  : strcpy(Nom_Mois,"Février")  ;
             break ;
   case 3  : strcpy(Nom_Mois,"Mars")     ;
             break ;
   case 4  : strcpy(Nom_Mois,"Avril")    ;
             break ;
   case 5  : strcpy(Nom_Mois,"Mai")      ;
             break ;
   case 6  : strcpy(Nom_Mois,"Juin")     ;
             break ;
   case 7  : strcpy(Nom_Mois,"Juillet")  ;
             break ;
   case 8  : strcpy(Nom_Mois,"Août")     ;
             break ;
   case 9  : strcpy(Nom_Mois,"Septembre");
             break ;
   case 10 : strcpy(Nom_Mois,"Octobre")  ;
             break ;
   case 11 : strcpy(Nom_Mois,"Novembre") ;
             break ;
   case 12 : strcpy(Nom_Mois,"Décembre") ;
             break ;
 }
 /* Récupération du département  */
 /* qui reste une chaîne         */
 Dept[0]=Num_Sec[5] ;
 Dept[1]=Num_Sec[6] ;
 Dept[2]='\0'       ;
 Departement=atoi(Dept) ;
```

```
 /* Récupération de la Commune */
 Com[0]=Num_Sec[5] ;
 Com[1]=Num_Sec[6] ;
 Com[2]=Num_Sec[7] ;
 Com[3]=Num_Sec[8] ;
 Com[4]=Num_Sec[9] ;
 Com[5]='\0'       ;
 Commune=atoi(Com) ;

 /* Récupération du numéro d'ordre */
 Num_Ordre=((Num_Sec[10]-48)*100)+((Num_Sec[11]-48)*10)+(Num_Sec[12]-
48);

 /* Affichage des résultats */
 printf("Bonjour                %s\n",Libelle)      ;
 printf("Vous êtes né en        : %d\n",Annee)      ;
 printf("Au mois de             : %s\n",Nom_Mois)   ;
 printf("Dans le département    : %d\n",Departement);
 printf("Dans la commune        : %d\n",Commune)    ;
 printf("Avec le numéro d'ordre : %d\n",Num_Ordre)  ;
}
```

ATTENTION

Aucun caractère accentué n'est autorisé dans le nom des variables. La variable algorithmique `Libellé` doit s'écrire `Libelle` en C. Les caractères accentués sont autorisés dans les messages affichés à l'écran (`printf`).

La compilation

Pour produire le fichier objet `numsecsoc.o` la commande à taper sous Linux ou MacOS (voir section 2 pour Windows) est :

```
$ cc -c numsecsoc
```

La compilation s'effectue en deux phases :

- l'analyse syntaxique ;
- la traduction en binaire.

La première phase analyse la syntaxe du programme. Elle modifie également le texte selon les *directives de compilation* éventuelles (`#include, #define`…).

Par exemple, la directive `#include <stdio.h>` a pour effet d'inclure à cet endroit du programme le contenu du fichier texte `stdio.h` situé dans le répertoire `/usr/include` (voir figure 1.4). La ligne `#include <stdio.h>` est remplacée par le texte du fichier `stdio.h` comme pourrait le faire un simple copier/coller.

Ces fichiers `include` sont les déclarations des procédures, fonctions ou constantes prédéfinies dans le langage C. `stdio.h` contient entre autres les déclarations des procédures `printf` et `scanf`.

Aucune traduction en binaire n'est effectuée tant qu'il y a des erreurs de syntaxe. Le fichier objet est généré quand le programme est syntaxiquement correct.

Les messages d'erreur qui apparaissent lors de la compilation sont des *erreurs de syntaxe* ou *compile-time errors* en anglais. Ils indiquent le numéro de ligne de l'erreur, mais l'erreur peut se trouver sur une des lignes précédentes.

Prenons l'exemple d'une erreur de syntaxe sur la 11 du programme précédent, en omettant le point-virgule à la fin de l'instruction `printf`.

```
printf("Entrez votre numéro de Sécurité sociale : ")
scanf("%s",Num_Sec) ;
```

Lors de la compilation, un message d'erreur indique la ligne numéro 12.

```
numsecsoc.c:12: parse error before 'scanf'
```

C'est en traitant le premier mot de la ligne 12 (`scanf`) que le compilateur s'aperçoit qu'il manque un caractère de fin d'instruction (;).

À NOTER

Cette phase se résume souvent à un aller-retour entre l'édition du texte pour corriger une erreur de syntaxe et la compilation qui en trouve une autre !

L'édition des liens

La commande d'édition des liens qui produit le fichier exécutable `numsecsoc` à partir de l'objet `numsecsoc.o`, est la suivante sous Linux ou MacOS (voir section 2 pour Windows) :

```
$ cc -o numsecsoc numsecsoc.o
```

Durant cette phase, l'éditeur de liens recherche dans le fichier `libc.a` qui est la librairie standard du C, les codes binaires des sous-programmes `printf` et `scanf` qui sont utilisés dans l'exemple.

Une erreur se produit si le code binaire n'est pas trouvé dans la librairie `libc.a`. Deux raisons peuvent en être la cause :

- Une erreur de syntaxe sur le nom de la procédure. Par exemple l'écriture de `print` à la place de `printf`.
- L'utilisation d'une procédure qui ne se trouve pas dans la librairie standard, mais dans une autre.

Dans le premier cas, le message d'erreur est :

```
numsecsoc.o: In function 'main':
numsecsoc.o(.text+0x24): undefined reference to 'print'
collect2: ld returned 1 exit status
```

La solution consiste à corriger l'erreur de syntaxe en remplaçant `print` par `printf`. Le numéro de la ligne de l'erreur n'apparaît plus car l'édition des liens porte sur le fichier objet (`numsecsoc.o`) et non sur le source (`numsecsoc.c`).

ATTENTION

Dès que le programme source `numsecsoc.c` est modifié, il faut à nouveau le compiler, puis faire une nouvelle édition des liens. Si vous oubliez la compilation, vous éditez les liens sur un objet inchangé, toujours erroné malgré la correction du programme source.

```
cc –c numsecsoc.c
cc –o numsecsoc numsec.o
```

Dans le second cas, le message est identique, mais la solution est différente.

Le programme `conv_francs_euros.c` convertit les euros en francs à deux décimales près en utilisant une fonction mathématique du C, `floor()`. Le code binaire de cette fonction n'est pas rangé dans la librairie `libc.a` mais dans la librairie `libm.c` (« m » pour mathématique).

conv_francs_euros.c

```
#include <stdio.h>
#include <math.h>
#define TAUX 6.55957
int main()
{
   float Euros ;
   float Francs;
   /* Saisie des euros */
   printf("Entrez la valeur en Francs (ex 97.5) : ") ;
   scanf("%f",&Francs);
   /* Conversion en euros avec la fonction floor() */
   Euros = (floor((Francs/TAUX)*100))/100 ;
   printf("Valeur en Euros : %f\n",Euros);
}
```

Les commandes de compilation et d'édition des liens sont :

```
$ cc -c conv_francs_euros.c
$ cc -o conv_francs_euros conv_francs_euros.o
```

Le message d'erreur lors de l'édition des liens est :

```
conv_francs_euros.o: In function 'main':
conv_francs_euros.o(.text+0x62): undefined reference to 'floor'
collect2: ld returned 1 exit status
```

La solution consiste simplement à fournir le nom de la libraire contenant cette fonction, la librairie mathématique `libm.a` pour `floor`, lors de la commande d'édition des liens (voir figure 1.4).

Pour cela on ajoute l'option `-lm`. La lettre « m » provient du nom de la librairie sans « lib » ni « .a ». Les librairies se trouvent dans le répertoire `/usr/lib`.

Voici la nouvelle commande d'édition des liens qui résout le problème :

```
$ cc -o conv_francs_euros conv_francs_euros.o -lm
```

À NOTER

Dans l'environnement UNIX, il existe un nombre considérable de librairies disponibles. Les librairies se trouvent dans le répertoire `/usr/lib`.

L'utilisation d'une procédure ou d'une fonction prédéfinie implique souvent la déclaration de la directive `include` adéquate, par exemple `#include <math.h>`, et parfois l'édition de lien avec l'option `-l` qui indique dans quelle librairie trouver le code binaire de cette procédure ou fonction, par exemple `-lm` pour la librairie mathématique `libm.a`.

La commande UNIX `ar` affiche la liste des modules contenus dans une librairie particulière. Voici la commande qui affiche la liste des modules de la librairie `libm.a` :

```
ar t /usr/lib/libm.a
```

La compilation et l'édition des liens

Les deux traitements peuvent s'effectuer en une seule commande. L'exécutable est généré à partir du fichier source. Le fichier objet n'est plus généré. La syntaxe est la suivante pour le premier exemple :

```
$ cc -o numsecsoc numsecsoc.c
```

Voici la syntaxe pour le second exemple avec la librairie mathématique :

```
$ cc -o conv_francs_euros conv_francs_euros.c –lm
```

Dans le premier cas, il est possible d'utiliser une syntaxe plus courte :

```
$ make numsecsoc
```

La commande `make` cherche le fichier source `numsecsoc.c` pour « construire » l'exécutable `numsecsoc`. Cette syntaxe, plus simple, n'est plus possible si une librairie autre que `libc.a` est nécessaire. Il faut alors écrire un fichier `Makefile`.

À NOTER

Si lors d'une commande de compilation et d'édition des liens, le nom de l'exécutable est omis, il s'appellera par défaut `a.out`. Par exemple :

```
cc conv_francs_euros.c -lm
```

L'exécution du programme

Le logiciel créé est exécuté en indiquant simplement son nom au niveau du prompt UNIX (`$`). Voici l'exécution des deux programmes précédents :

```
$ numsecsoc
Entrez votre numéro de Sécurité sociale : 1650475110019
Bonjour                Monsieur
Vous êtes né en        : 1965
Au mois de             : Avril
Dans le département    : 75
Dans la commune        : 75110
Avec le numéro d'ordre : 19

$ conv_francs_euros
Entrez la valeur en Francs (ex 97.5) : 100
Valeur en Euros : 15.240000
```

À NOTER

Sous UNIX, si la variable système `PATH` ne contient pas dans sa liste le répertoire courant « `.` », alors la syntaxe pour exécuter le programme `numsecsoc` se note (sans espace) :

```
./numsecsoc
```

À la place de :

```
numsecsoc
```

Pour y remédier il faut saisir la syntaxe suivante, sans espace, à chaque début de session.

```
PATH=$PATH:.
```

Pour rendre permanente cette modification il faut ajouter cette ligne de commande à la fin du fichier « `.profile` ». Pour plus de détails sur ces notions, se reporter au livre « *Unix & Linux – Utilisation et administration* » du même auteur.

Les erreurs qui apparaissent lors de l'exécution sont plus délicates à trouver, car il s'agit de problèmes de conception du programme. Le programme peut très bien fonctionner pour certaines valeurs et dysfonctionner pour d'autres, comme dans le cas d'une division par une valeur qui parfois devient 0.

Ce type d'erreur se nomme *erreur d'exécution* ou *runtime error* en anglais. Ces erreurs de conception se nomment *bogues* ou « *bugs* » en anglais.

La solution est d'essayer de localiser la ligne qui pose le problème, et ensuite de trouver pourquoi avec telle ou telle donnée cette ligne provoque l'erreur.

ATTENTION

La correction se fera au niveau du programme source, ce qui suppose de refaire la compilation et l'édition des liens.

1.4. Les recommandations

Bien que cet ouvrage traite uniquement de la programmation C, il convient de rappeler les règles qui régissent les développements d'applications.

La plupart d'entre elles sont des règles de bon sens. D'autres rappellent que le développement d'applications est souvent un travail d'équipe, et que l'apprentissage du développement n'est pas seulement celui d'un langage mais aussi celui de la rigueur dans l'écriture du programme source.

- **Faire l'analyse, puis l'algorithme avant d'écrire le programme**. Les solutions préconisées lors de l'analyse sont celles qui sont programmées. Faire un développement en négligeant certaines de ces étapes conduit à un programme mal conçu ou erroné.
- **Commenter le programme source**. Il est important d'inclure dans le fichier source des commentaires pour rendre lisible le programme. La lisibilité est indispensable au développeur qui, deux semaines plus tard, s'interroge sur la signification d'un calcul, ou de rendre compréhensible le programme à d'autres développeurs. Il est difficile de comprendre et de maintenir le programme d'un autre sans aucun commentaire.
- **Indenter le programme**. La présentation en « escalier » des niveaux d'instructions par le décalage vers la droite du début des lignes internes permet, par exemple, de visualiser rapidement les instructions d'une boucle qui dysfonctionne et ainsi de trouver rapidement l'erreur.
- **Choisir des noms de variables selon leur rôle**. Un choix judicieux du nom des variables rend plus lisible le programme. Par exemple, le choix du nom `Prix_TTC` indique mieux le rôle de cette variable qu'un nom comme `valeur`.
- **Concevoir le programme en vue de sa portabilité**. Le langage C est indépendant de l'ordinateur. C'est sa force mais aussi sa limite. Il n'est pas possible en respectant la norme du langage C de développer une interface graphique universelle. L'interface est dépendante du système d'exploitation

(Windows, UNIX, MacOS). Concevoir une application portable consiste à « encapsuler » les éléments dépendant du système, comme les fonctions graphiques, dans des procédures spécifiques, ce qui évite de disséminer ces appels dans le code source. L'adaptation du programme à un autre environnement se limitera aux seules procédures contenant ces fonctions.

2. Les outils de développement

Cette section présente les outils de développement sous trois systèmes différents. Nous montrons comment écrire un programme source, le compiler et l'exécuter.

2.1. Sous Windows

Windows propose des logiciels de développement « intégrés » gratuits comme DEV-C++ que nous présentons. Ils regroupent tous les outils nécessaires au développement. Ce sont des *IDE* « *Integrated Development Environment* ».

L'écriture du programme source

Après avoir téléchargé et installé DEV-C++ (version 5 et plus), il faut ouvrir l'application, et sélectionner : Fichier → Nouveau → Fichier Source (figure 1.5).

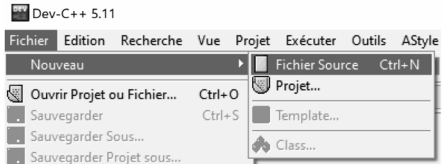


Figure 1.5

DEV-C++ : Création d'un nouveau fichier source.

Dans la nouvelle fenêtre d'édition on écrit le programme source C. Une fois le programme rédigé, il faut le sauvegarder dans un fichier source : Fichier → Sauvegarder Sous (figure 1.6).

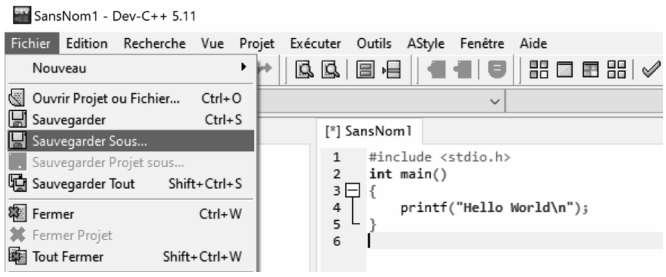


Figure 1.6

DEV-C++ : Édition et sauvegarde du fichier source.

Dans la nouvelle fenêtre il faut choisir le répertoire de sauvegarde (figure 1.7). Dans « Nom du fichier » on saisit le nom avec l'extension, par exemple `hello.c`. Dans « Type », on sélectionne « C source files (*.c) ».

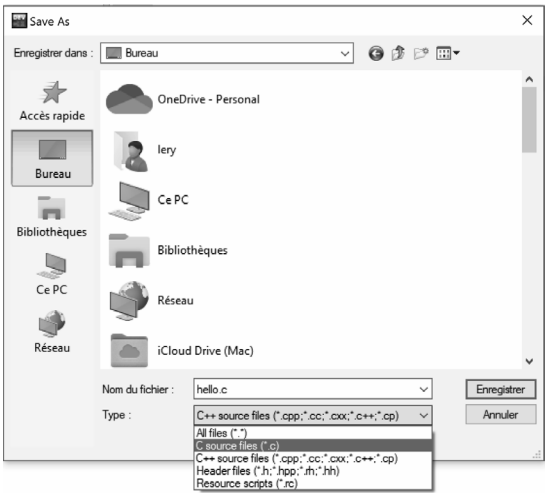


Figure 1.7

DEV-C++ : Fenêtre de sauvegarde.

À NOTER

Aucun espace n'est autorisé dans le nom du fichier. Cela provoque des erreurs d'exécution.

La compilation

Sélectionnez : Exécuter → Compiler (figure 1.8).

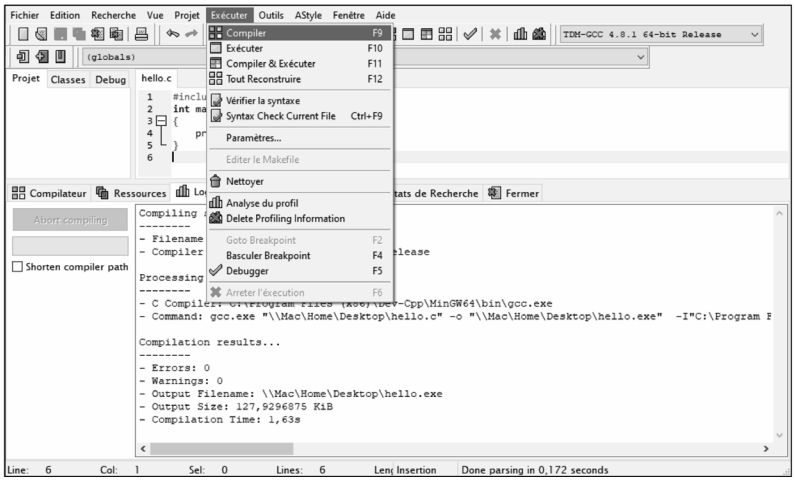


Figure 1.8

DEV-C++ : Fenêtre de compilation.

Le résultat de la compilation apparaît en partie basse. Si des erreurs apparaissent, il faut les corriger et compiler à nouveau jusqu'à la résolution de toutes les erreurs.

À NOTER

Lors de la compilation, les messages d'erreurs indiquent le numéro de ligne. L'affichage des numéros de ligne dans la fenêtre d'édition s'obtient par :

Menu Outils → Options de l'éditeur → Onglet Affichage → Case Numéro de ligne

L'exécution

Sélectionnez : Exécuter → Exécuter (figure 1.9). L'exécution se déroule dans une fenêtre sur fond noir. À la fin, la saisie d'une touche retourne à la fenêtre d'édition.

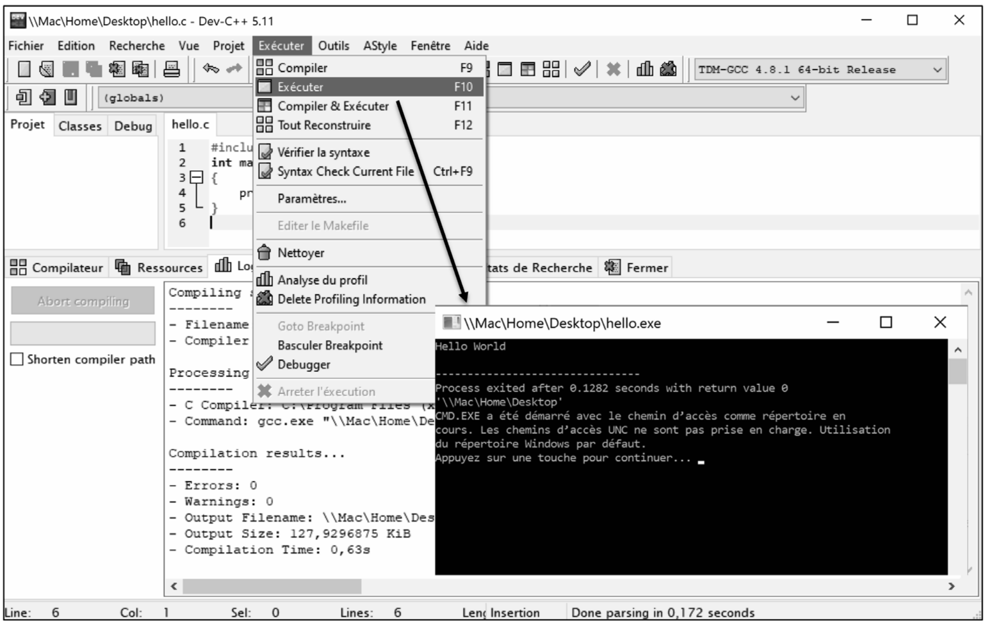


Figure 1.9

DEV-C++ : Fenêtre d'exécution.

2.2. Sous Linux

Nous présentons les outils sous la distribution Linux Ubuntu. L'édition de texte utilise l'éditeur *gedit*. La compilation utilise la commande `cc` ou `gcc`.

L'écriture du programme source

Nous présentons l'éditeur de texte *gedit* disponible en standard dans cette distribution.

Pour lancer *gedit*, cliquez sur l'icône « Afficher les applications » (étape 1).

Dans la fenêtre de recherche saisissez « edit » (étape 2), puis cliquez sur « Éditeur de texte » (étape 3) comme cela est présenté sur la figure 1.10.



Figure 1.10

Linux Ubuntu : Lancement du logiciel gedit.

Une fenêtre d'édition apparaît dans laquelle on écrit le programme source C.

Une fois le programme rédigé, il faut le sauvegarder en cliquant sur le bouton « Enregistrer » (étape 1).

Sélectionnez le répertoire (par exemple « Documents ») et indiquez le nom dans le champ « Nom » (étape 2), comme `hello.c` (figure 1.11).

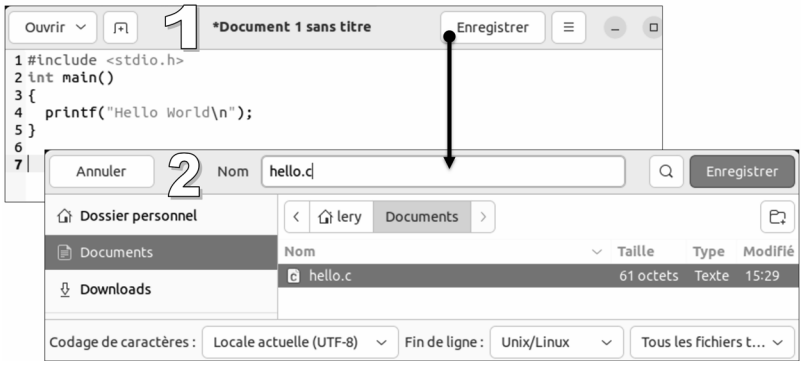


Figure 1.11

Linux Ubuntu : Édition et sauvegarde du fichier source.

À NOTER

Aucun espace n'est autorisé dans le nom du fichier. Cela provoque des erreurs d'exécution. Pour afficher les numéros de lignes, il faut cliquer sur l'icône avec trois barres horizontales. Dans les préférences de l'éditeur on coche « Afficher les numéros de ligne » (figure 1.12).

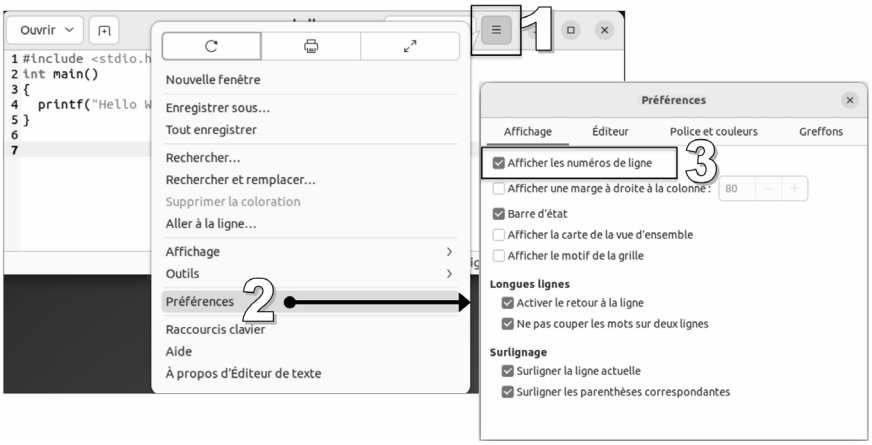


Figure 1.12

Linux Ubuntu : Affichage des numéros de ligne sous *gedit*.

La compilation

Pour compiler il faut ouvrir une *fenêtre Terminal* pour y saisir des commandes.

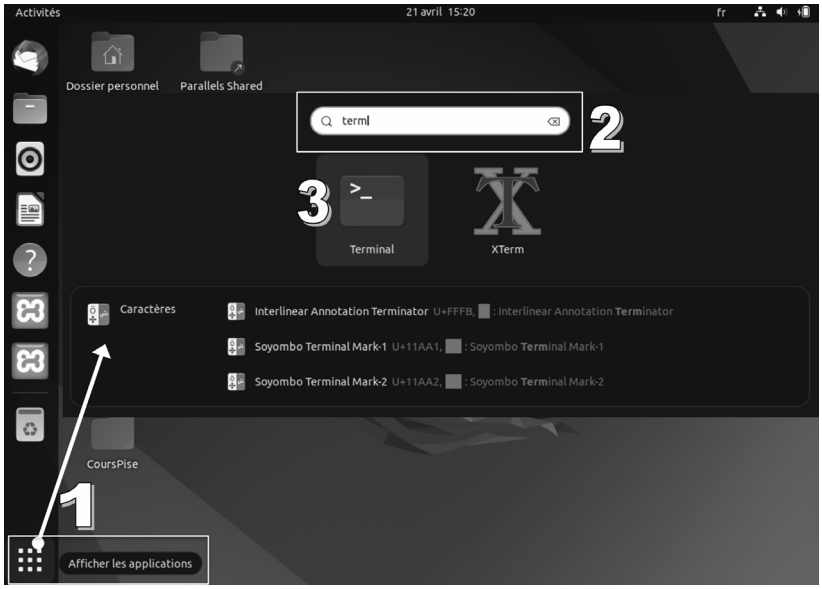


Figure 1.13

Linux Ubuntu : Ouverture d'une fenêtre Terminal.

Pour cela, cliquez sur l'icône « Afficher les applications » (étape 1). Dans la fenêtre de recherche saisissez « term » (étape 2), puis cliquez sur « Terminal » (étape 3) comme cela est présenté sur la figure 1.13.

La figure 1.14 présente les commandes UNIX pour : afficher les répertoires, et les fichiers ; se déplacer dans la hiérarchie ; compiler le fichier source.

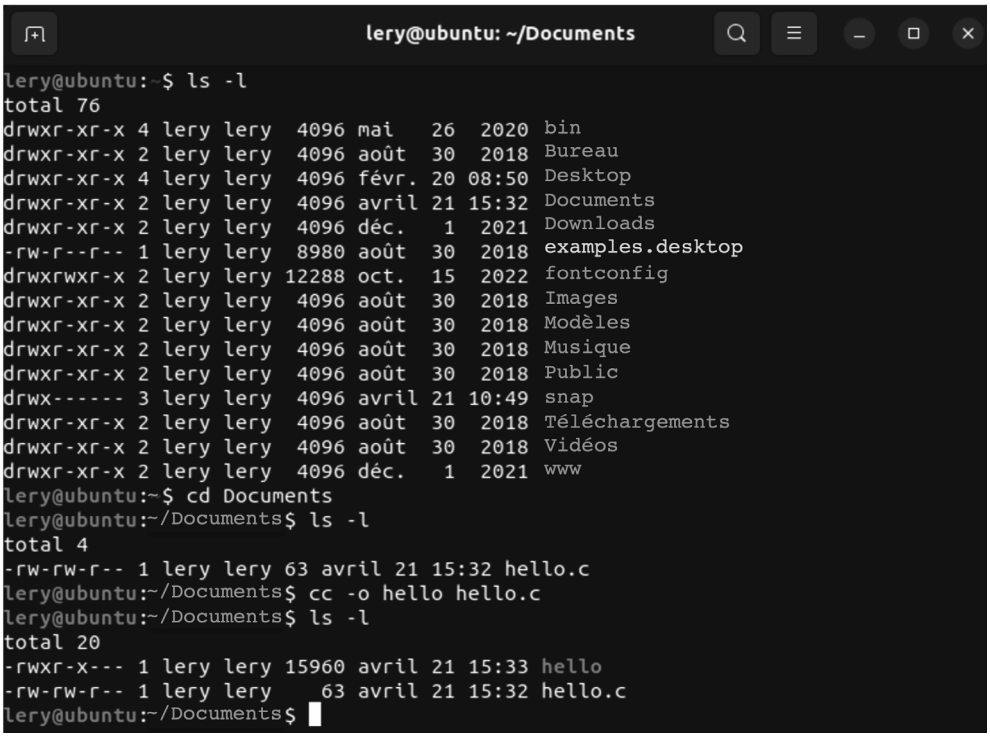


Figure 1.14

Linux Ubuntu : Affichage, déplacement et compilation.

À l'ouverture de la fenêtre Terminal, vous vous trouvez dans votre répertoire personnel. La première commande « `ls -l` » montre son contenu :

```
lery@ubuntu:~$ ls -l
total 76
drwxr-xr-x 4 lery lery  4096 mai   26  2020 bin
drwxr-xr-x 2 lery lery  4096 août  30  2018 Bureau
drwxr-xr-x 4 lery lery  4096 févr. 20 08:50 Desktop
drwxr-xr-x 2 lery lery  4096 avril 21 15:33 Documents
drwxr-xr-x 2 lery lery  4096 déc.   1  2021 Downloads
...
```

La liste ci-dessus présente le répertoire « `Documents` » dans lequel le fichier source « `hello.c` » a été sauvegardé à l'étape précédente.

Il faut s'y déplacer par la commande « `cd Documents` », puis afficher son contenu par la commande « `ls -l` ». Le fichier source apparaît.

```
lery@ubuntu:~$ cd Documents
```

```
lery@ubuntu:~/Documents$ ls -l
total 4
-rw-rw-r-- 1 lery lery 63 avril 21 15:32 hello.c
```

Il est maintenant possible d'effectuer la compilation.

```
lery@ubuntu:~/Documents$ cc -o hello hello.c
```

La commande suivante confirme la création du fichier exécutable « `hello` ».

```
lery@ubuntu:~/Documents$ ls -l
total 20
-rwxr-x--- 1 lery lery 15960 avril 21 17:02 hello
-rw-rw-r-- 1 lery lery    63 avril 21 15:32 hello.c
```

Si des erreurs apparaissent, il faut les corriger dans la fenêtre d'édition, sauvegarder le fichier source, le compiler à nouveau jusqu'à la résolution de toutes les erreurs. Dans cet environnement on conserve ouvertes la fenêtre d'édition pour éditer le programme, et la fenêtre Terminal pour le compiler et l'exécuter.

L'exécution

L'exécution se déroule dans la fenêtre Terminal. Voici un exemple d'exécution du programme précédent.

```
lery@ubuntu:~/Documents$ hello
Hello World
```

À NOTER

Selon l'environnement de travail sous Linux, et le contenu de la variable système `PATH`, l'exécution nécessite parfois d'ajouter « `./` » devant, sans aucun espace :

```
./hello
```

2.3. Sous MacOS

Le système MacOS est un système UNIX. Les outils sont très similaires à ceux qui ont été présentés sous Linux, en particulier la fenêtre Terminal et les commandes.

L'écriture du programme source

L'éditeur standard sous UNIX et donc sous MacOS est *vi* (pour *visual*). Cependant, cet éditeur très puissant et très prisé des administrateurs systèmes n'est pas simple d'utilisation. Aussi nous présentons dans cette section un éditeur de texte gratuit téléchargeable sur Internet « *Sublime Text* », mais n'importe quel autre éditeur de texte convient.

Après avoir téléchargé et lancé « *Sublime Text* » on écrit le programme source C dans la fenêtre d'édition.

Une fois le programme rédigé, il faut le sauvegarder en sélectionnant le menu :

File → Save.

Dans la nouvelle fenêtre, sélectionnez le répertoire (par exemple « Documents ») et indiquez le nom dans le champ « Save As », comme `hello.c` (figure 1.15).

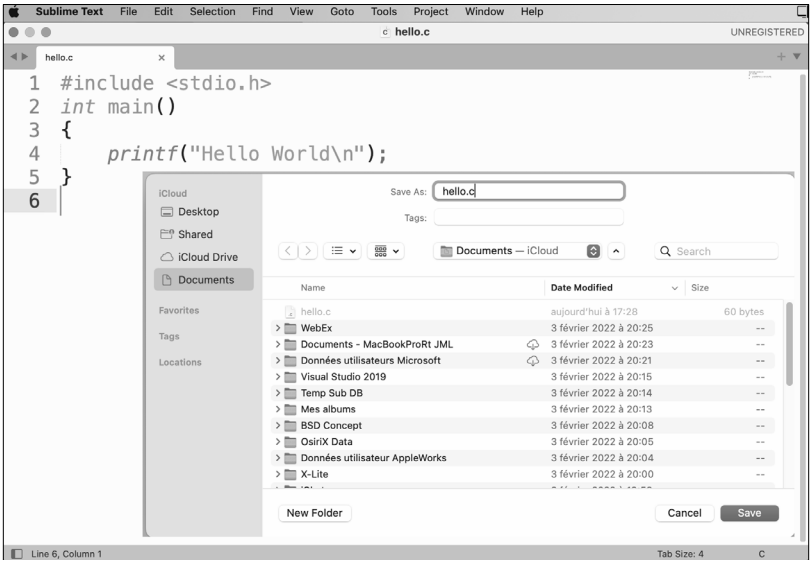


Figure 1.15

MacOS Sublime Text : Édition et sauvegarde du fichier source.

À NOTER

Afin d'afficher les numéros de lignes sous Sublime Text, il faut accéder à Sublime Text → Preferences → Settings, et dans la fenêtre des paramètres passer « line_numbers » à true.

La compilation

Comme pour Linux, pour compiler il faut ouvrir une fenêtre Terminal pour y saisir des commandes. Ce logiciel se trouve dans : Applications → Utilitaires → Terminal.

À l'ouverture de la fenêtre Terminal, vous vous trouvez dans votre répertoire personnel. La première commande « `ls -l` » montre son contenu :

```
~ lery$ ls -l
total 120
drwx------   29 lery  staff    928 26 fév  2022 Applications/
drwx------  329 lery  staff  10528 21 avr 17:38 Desktop/
drwx------   44 lery  staff   1408 21 avr 17:28 Documents/
drwx------   11 lery  staff    352 21 avr 10:58 Downloads/
...
```

La liste ci-dessus présente le répertoire « `Documents` » dans lequel le fichier source « `hello.c` » a été sauvegardé à l'étape précédente.

Il faut s'y déplacer par la commande « `cd Documents` », puis afficher tous les programmes sources C par la commande « `ls -l *.c` ». Le fichier apparaît.

```
~ lery$ cd Documents
Documents lery$ ls -l *.c
-rw-r--r--  1 lery  staff  60 21 avr 17:36 hello.c
```

Il est maintenant possible d'effectuer la compilation.

```
Documents lery$ cc -o hello hello.c
```

La commande suivante confirme la création du fichier exécutable « `hello` ».

```
Documents lery$ ls -l hello*
-rwxr-xr-x  1 lery  staff  49424 21 avr 17:38 hello
-rw-r--r--  1 lery  staff     60 21 avr 17:36 hello.c
```

Si des erreurs apparaissent, il faut les corriger dans la fenêtre d'édition, sauvegarder le fichier source, le compiler à nouveau jusqu'à la résolution de toutes les erreurs. Dans cet environnement on conserve ouvertes la fenêtre d'édition pour éditer le programme, et la fenêtre Terminal pour le compiler et l'exécuter.

À NOTER

Si le compilateur C n'est pas installé, il suffit d'installer les applications gratuites « XCode » et « Xcode Command Line Tools » à partir de l'App Store d'Apple.

L'exécution

L'exécution se déroule dans la fenêtre Terminal. Voici l'exécution du programme précédent.

```
Documents lery$ hello
Hello World
```

RAPPEL

Selon l'environnement de travail sous MacOS, et le contenu de la variable système `PATH`, l'exécution nécessite parfois d'ajouter « `./` » devant, sans aucun espace :

```
./hello
```

3. La structure et les éléments du langage

Le langage C est un langage informatique. Comme toute langue, il possède son alphabet, son orthographe, sa grammaire.

Ce chapitre décrit les différents éléments de ce langage comme par exemple, la structure de la langue, ses mots réservés et le mode de gestion des données. La syntaxe détaillée est abordée dans les chapitres suivants.

3.1. Structure d'un programme C

La fonction principale d'un programme informatique est de traiter des données. Ce concept se retrouve directement dans la structure même du programme où apparaît la déclaration des données et leur manipulation.

À NOTER

Un programme C est structuré en blocs. Un bloc contient une ligne d'en-tête, une partie déclaration et une partie instruction. La déclaration des variables réserve de la place en mémoire. Les instructions modifient les données contenues dans ces variables.

Dans le langage C toute variable doit être déclarée avant son usage. Cela peut se faire juste avant sa première utilisation n'importe où dans le programme depuis la norme C99. Cependant les bonnes pratiques préconisent de déclarer toutes les variables au début avant la première instruction, c'est la partie « déclaration des variables », ce qui était imposé en C90. Cela rend le programme plus lisible et plus facile à maintenir. Dans la structure de bloc présentée à la figure 1.16, la partie déclaration se termine lorsque la première instruction `printf()` est trouvée.

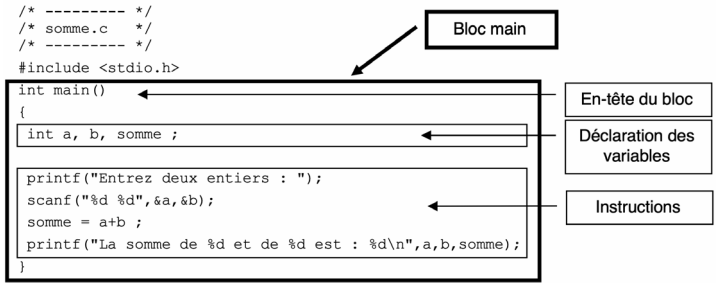


Figure 1.16

Structure en bloc.

Quand le programme contient des sous-programmes, chacun d'eux est un bloc.

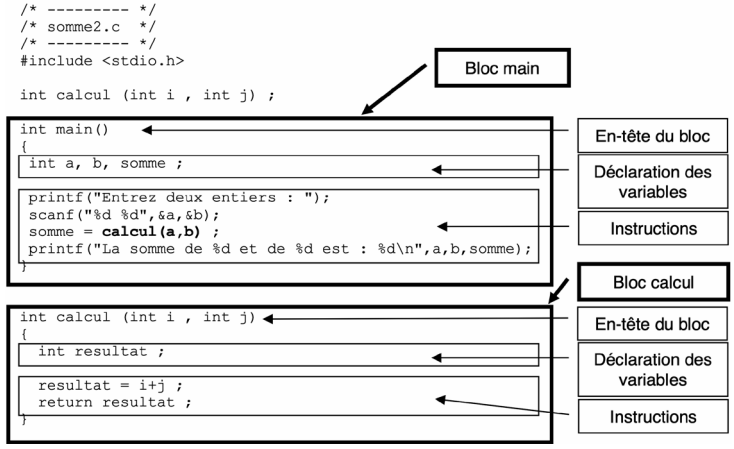


Figure 1.17

Blocs multiples.

Dans `somme2.c` (figure 1.17) le programme principal `main` appelle une fonction `calcul()` qui effectue la somme des deux entiers.

Les deux blocs sont écrits l'un après l'autre dans le même fichier.

Les programmes `somme.c` et `somme2.c` effectuent le même traitement. Les syntaxes de compilation sont similaires, voici celle du premier :

```
$ cc -o somme somme.c
```

Voici un exemple d'exécution de ce programme :

```
$ somme
Entrez deux entiers : 12 24
La somme de 12 et de 24 est : 36
```

À NOTER

La ligne d'en-tête du programme principal est : `int main()`. Cette syntaxe, imposée par la norme C99, pouvait s'écrire `main()` auparavant. Si le programme retourne un entier (`int`) il faut ajouter une dernière ligne dans son bloc : `return 0`. La norme C99 autorise cette absence pour `main()`. Ces détails syntaxiques sont abordés au chapitre 5.

3.2. Notions sur les variables

Qu'est-ce qu'une variable ?

Une *variable* est un *espace mémoire* que le programme réserve lors de son exécution. La réservation est effectuée à la déclaration de la variable. Les données utilisées par le programme sont rangées dans des variables. Leur déclaration doit être effectuée avant les instructions les traitant.

Une variable est caractérisée par :

- Un *nom.* L'accès à l'espace mémoire réservé se fait grâce au nom qui lui est affecté. C'est le nom de la variable.
- Un *type.* La taille de cet espace est fournie par le type de la variable. Il correspond à un certain nombre d'octets. Par exemple, un caractère prendra un octet (8 bits), alors qu'un entier occupera généralement quatre octets (32 bits).

  Le type indiquera aussi la méthode utilisée pour coder et décoder les données en binaire. Un caractère ne sera pas codé de la même manière qu'un entier.

ATTENTION

La déclaration d'une variable réserve un espace mémoire qui n'est pas « nettoyé ». La valeur de la variable après sa déclaration est indéfinie ! Chaque logiciel réserve, au démarrage, un espace mémoire pour ses données, puis libère celui-ci quand il se termine, sans le nettoyer ! Une variable dont l'espace est réservé dans la mémoire précédemment libérée, récupère de facto une valeur correspondant aux informations binaires laissées par le logiciel précédent. La valeur de la variable correspond à ce qui « traîne en mémoire » à cet endroit. Elle est indéfinie. Mais la variable possède bien une valeur ! Si la réservation est effectuée dans une zone mémoire jamais utilisée, la valeur de la variable est nulle car l'information binaire ne contient que des 0.

La première utilisation d'une variable doit consister à lui affecter une bonne valeur, donc à faire une sorte de « ménage ». Cette notion se nomme *l'initialisation des variables*.

La figure 1.18 présente ce que provoquent la déclaration et l'initialisation d'une variable de type caractère et d'une variable de type entier.

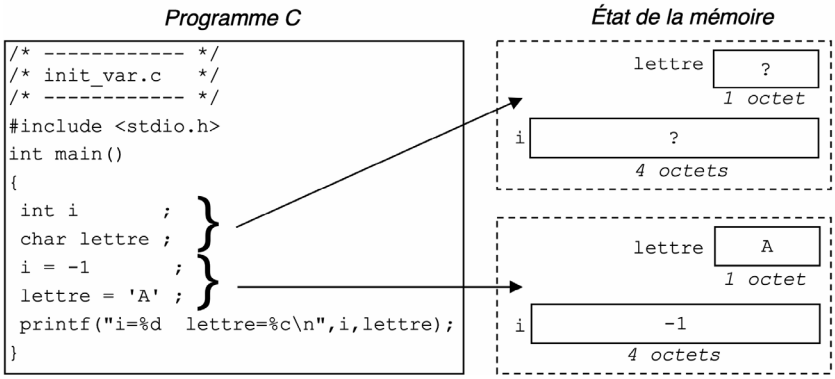


Figure 1.18

Déclaration et initialisation des variables.

Nom des variables

Les caractères valides pour nommer une variable sont :

- les lettres majuscules : A à Z ;
- les lettres minuscules : a à z ;
- les chiffres : 0 à 9 ;
- le caractère souligné (_).

Les caractères accentués et de ponctuation ne sont pas autorisés.

Voici quelques noms autorisés : compteur, resultat, Nombre_Annees, calcul2.

Voici des exemples de noms interdits : an'02, valeur-actuelle, +value, année.

ATTENTION

Les lettres majuscules et minuscules diffèrent en C. Une variable nommée `PlusValue` sera différente de `plusvalue`. Les mots réservés du langage sont en minuscules. L'écriture de `Main()` à la place de `main()` provoque une erreur d'édition des liens.

erreur_main.c

```
#include <stdio.h>
int Main()
{
 printf("Hello World\n");
}
```

La compilation provoque une erreur d'édition des liens sur `Main()`.

```
$ cc -o erreur_main erreur_main.c
/usr/bin/ld : /usr/lib/gcc/x86_64… : In function '_start':
(.text+0x1b) : undefined reference to 'main'
collect2: error: ld returned 1 exit status
```

Type des variables

Codage binaire d'une variable selon son type

Les variables contiennent des données modifiables par les instructions. Les données « humaines » sont par exemple des numéros, des caractères, des nombres avec des virgules. Or dans l'ordinateur il n'y a que des 0 et des 1. Il faut définir une méthode pour coder et décoder ces données en binaire. C'est le rôle du type de la variable qui indique comment traduire en binaire et comment retrouver l'information sous sa forme originale « humaine ».

Une donnée qui semble identique sera codée différemment en binaire selon son type. Ainsi la valeur numérique 1 diffère en binaire de la donnée de type caractère '1'. Dans ce dernier cas, 1 est un caractère au même titre que A ou B quand on tape A3B1 sur le digicode d'un immeuble. La figure 1.19 présente le codage binaire de de la variable de type entier (`int` en C) et de la variable caractère (`char` en C).

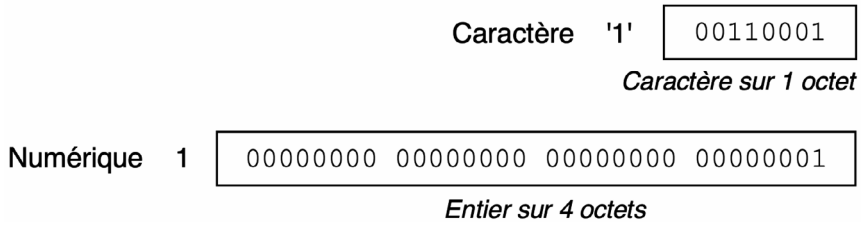


Figure 1.19

Codage binaire de l'entier 1 et du caractère '1'.

Les types simples

Les variables de types simples concernent les données « uniques » comme un entier, un caractère ou un nombre avec une virgule. Elles ont des opérateurs associés (addition, soustraction…), et des fonctions particulières (`toupper()`, `tolower()` pour les caractères par exemple).

Les types simples sont :

- le type entier (`int`) ;
- le type réel (`float` ou `double`) ;
- le type caractère (`char`) ;
- le type booléen (`int`) ;
- le type énuméré (`enum`) ;
- le type pointeur (`int *` ou `char *`).

Les types structurés

Les variables de types structurés correspondent à un ensemble de données, comme une suite de nombres (tableau d'entiers) ou de caractères (chaîne de caractères).

Chaque élément de cet ensemble peut contenir une donnée de type simple mais également une donnée structurée. Cet « emboîtement » de structures de données ressemble un peu aux poupées russes : L'ouverture d'une poupée dévoile d'autres poupées à l'intérieur. Au bout d'un moment on retombe sur une ou plusieurs données de type simple. Bien comprendre les données élémentaires permet d'assimiler plus facilement les données structurées.

Les types structurés sont :

- le type tableau (`int notes[10]`) ;
- le type chaîne de caractères (`char prenom[20]`) ;
- le type enregistrement (`struct`) ;
- le type union (`union`) ;
- le type fichier (`FILE`) ;
- le type complexe (`_Complex`) ;

Portée des variables

Les variables déclarées à l'intérieur d'un bloc ou à l'extérieur des blocs sont dites respectivement *locales* ou *globales*. Voici les définitions de ces notions. Leur utilisation est détaillée au chapitre 5 avec les procédures et les fonctions.

Les variables locales

Toute variable déclarée à l'intérieur d'un bloc est une variable *locale*.

Elle n'existe que dans ce bloc. Elle est invisible des autres blocs. Seules les instructions du bloc où elle est déclarée peuvent la manipuler.

Dans la figure 1.17, les variables `a, b` et `somme` sont locales au bloc `main()`. Les variables `i, j` et `resultat` sont locales au bloc `calcul()`. Toute instruction du bloc `calcul()` essayant de manipuler la variable `somme` provoquerait une erreur de compilation indiquant que la variable `somme` n'est pas déclarée !

Les variables globales

Toute variable déclarée à l'extérieur des différents blocs est une variable *globale*.

Elle existe pour tous les blocs et elle est manipulable par toutes les instructions. Elle ne doit pas être déclarée une autre fois dans un bloc.

Dans le programme `somme3.c` les variables `a` et `b` deviennent des variables globales. Elles sont directement utilisables par `main()` et `calcul()`.

somme3.c

```
#include <stdio.h>
int calcul();
int a, b ;

int main()
{
 int somme ;
 printf("Entrez deux entiers : ");
```

```
 scanf("%d %d",&a,&b);
 somme = calcul() ;
 printf("La somme de %d et de %d est : %d\n",a,b,somme);
}

int calcul()
{
  int resultat ;
  resultat = a+b ;
  return resultat ;
}
```

Re-déclaration d'une variable globale

Si une variable globale est redéclarée dans un bloc, cela provoque la création d'une nouvelle variable locale ayant le même nom. Les instructions du bloc où se trouve la nouvelle déclaration utiliseront la variable locale.

ATTENTION

La re-déclaration dans un bloc d'une variable globale rend celle-ci inaccessible pour les instructions de ce bloc. Elle est « cachée » par la variable locale créée par cette déclaration.

Les classes de stockage des variables

La notion de *classe de stockage* d'une variable est une notion avancée en programmation C. Elle correspond à la portée et à la durée de vie de la variable. Elle se note avant le type de la variable au moment de sa déclaration.

Cette section ne détaille pas chaque classe. Elles le sont au chapitre 5. Néanmoins pour comprendre leur rôle, nous présentons la classe `static`.

La classe static

Une variable locale est créée à chaque exécution du bloc où elle est déclarée. Elle est détruite quand l'exécution des instructions du bloc est terminée. Ainsi sa durée de vie est celle du sous-programme où elle est déclarée. À chaque nouvel appel du sous-programme, la variable locale est à nouveau créée puis détruite.

Avec l'indicateur de classe `static`, sa durée de vie est prolongée, elle devient permanente. La variable est bien locale, elle reste inconnue des autres sous-programmes, mais elle n'est plus détruite à la fin du premier appel du sous-programme. Sa valeur est conservée pour l'appel suivant.

Le programme `statique.c` présente un exemple d'utilisation.

statique.c

```
#include <stdio.h>

void traitement() ;

int main()
{
  int i ;
  for (i=0 ; i<3 ; i++)
    traitement();
}
```

```
void traitement()
{
  /* La variable k conserve sa valeur à chaque appel successif */
  static int k=0;
  int j=0       ;
  printf("j=%d k=%d\n",++j,++k);
}
```

La commande de compilation de ce programme est :

```
$ cc -o statique statique.c
```

L'exécution de ce programme montre que la variable `k` conserve sa valeur d'un appel à l'autre du sous-programme `traitement()` :

```
$ statique
j=1 k=1
j=1 k=2
j=1 k=3
```

Liste des classes de stockage

Voici la liste des classes de stockage :

- `auto` : classe automatique, valeur par défaut. Ce sont les variables locales. Elles sont créées au moment de l'exécution du bloc dans lequel elles sont déclarées et détruites à la fin de celui-ci. Leur durée de vie est celle du bloc.
- `static` : indique au compilateur de conserver la valeur de la variable locale, d'un appel à l'autre du sous-programme. La variable locale statique n'est pas détruite à la fin de chaque appel, sa durée de vie est celle du programme.
- `register` : est utilisé pour positionner la variable locale dans un registre du processeur plutôt qu'en mémoire vive, si possible, ce qui accélère le temps d'exécution. Cela est utile pour les entiers compteurs de boucle, très souvent accédés, car les registres sont plus rapides d'accès que la mémoire vive.
- `extern` : est utilisé dans le cas de la compilation séparée. Précise que la variable globale est déclarée dans un autre fichier.

Il existe aussi des qualificateurs de classe. Par exemple, `const` fige la valeur de la variable. Ce qui la transforme en une constante.

Les qualificateurs sont :

- `const` : indique au compilateur que la variable ne doit pas changer.
- `restrict` : introduit en C99, cela s'applique aux pointeurs exclusivement.

  Cela indique au compilateur que le pointeur ainsi déclaré sera le seul à pointer sur une zone mémoire spécifique.
- `volatile` : indique au compilateur que la variable peut être modifiée à son insu. Cela empêche toute optimisation de code de sa part et l'oblige à lire et à écrire la variable à chaque fois que cela est indiqué par le programme.

  Cela est utilisé par exemple avec la programmation des signaux.

Ces notions sont présentées ici en même temps que la notion de variable. Mais leur utilisation et surtout leur compréhension sont détaillées au chapitre 5.

3.3. Notions sur les constantes

Qu'est-ce qu'une constante ?

Une constante est l'inverse d'une variable. C'est une valeur fixe qui ne change jamais durant l'exécution du programme. Elle peut être entière, réelle, caractère, ou chaîne de caractères comme une variable.

Voici quelques exemples :

- constantes entières : `1, 2, -10` ;
- constantes caractères : `'A', 'B', 'Z'` ;
- constantes chaînes de caractères : `"Bonjour", "Dupont", "Avril"`.

Les constantes font partie du langage C. Elles ne n'ont pas besoin d'être déclarées pour être reconnues par le compilateur. Leur utilisation permet d'affecter une valeur aux variables, soit directement, soit au travers des expressions (calculs).

Voici des exemples d'instructions où apparaissent des constantes :

```
i = -10 ;
surface_sphere = 4 * 3.14159 * rayon * rayon ;
strcpy(nom,"Dupont") ;
initiale = 'D' ;
```

Après ces instructions, les variables modifiées contiennent les valeurs suivantes :

- La variable `i` contient la valeur entière `-10`.
- La variable `surface_sphere` contient la valeur réelle `12.56636` multipliée par le `rayon` au carré.
- La variable `nom` contient le texte (ou chaîne de caractères) `Dupont`.
- La variable `initiale` contient la lettre `D`.

Déclaration des constantes

Une constante n'a pas besoin d'être déclarée. Cependant dans certains cas, il peut être intéressant de lui donner un nom ou mnémonique afin de rendre plus lisible le programme. L'exemple précédent du calcul de la surface d'une sphère gagnerait un peu en lisibilité s'il s'écrivait :

```
surface_sphere = 4 * PI * rayon * rayon ;
```

Pour que `PI` représente la valeur `3.14159`, il suffit d'en faire la déclaration en début de programme. Il existe deux moyens pour déclarer une constante :

- avec le modificateur de classe `const` ;
- avec la directive de compilation `#define`.

Le programme `declar_const.c` présente la syntaxe des deux méthodes.

declar_const.c

```
#include <stdio.h>
#define PI 3.14159
int main()
{
```

```
  const int COEFF=4 ;
  float surface, rayon ;

  printf("Entrez un rayon d'une sphere : ");
  scanf("%f",&rayon);
  surface = COEFF * PI * rayon * rayon     ;
  printf("La surface est : %f\n",surface)  ;
}
```

La commande de compilation de ce programme est :

```
$ cc -o declar_const declar_const.c
```

Voici un exemple d'exécution de ce programme :

```
$ declar_const
Entrez le rayon d'une sphère : 2.5
La surface est : 78.539749
```

La méthode la plus utilisée dans le langage C est celle du `#define`.

Une autre utilisation de la déclaration des constantes est le *paramétrage* du programme. Cela consiste à écrire le programme de manière à pouvoir changer facilement une valeur constante qui pourrait quand même évoluer un jour.

Prenons le cas d'un programme de gestion du personnel d'une entreprise de 200 salariés maximum. Une des constantes du programme est cette limite du nombre de salariés. Rien ne dit que l'entreprise aura toujours cette taille maximale. Il sera peut-être nécessaire dans le futur d'adapter ce programme à la gestion d'au plus 2000 personnes. Si lors de l'écriture initiale du programme, une constante `NOMBRE_PERSONNES` avec la valeur 200 est déclarée dès le début, et que chaque écriture de ce nombre est remplacée par la constante `NOMBRE_PERSONNES`, alors l'adaptation à la nouvelle limite devient très simple. Il suffit de changer la ligne de déclaration, et d'y remplacer la valeur 200 par la valeur 2000, puis de recompiler le programme. Le reste du programme est changé.

À NOTER

La modification de la déclaration d'une constante modifie automatiquement, toutes les lignes du programme où elle est utilisée. Cette modification s'opère au moment de la compilation.

Des exemples de paramétrage sont présentés dans le chapitre 5 avec les tableaux.

3.4. Notions sur les instructions

Les expressions

Une expression est un calcul combinant des variables, des constantes et des opérateurs. Ces expressions peuvent être arithmétiques, ou logiques lorsque le calcul est booléen (VRAI ou FAUX).

Les expressions sont généralement utilisées à droite du signe égal. Le résultat du calcul est alors rangé dans la variable se trouvant à gauche du signe.

Le premier exemple est un calcul arithmétique donnant un résultat entier :

```
resultat = ((3*i)+(5-k))/18 ;
printf("%d", resultat) ;
```

La variable `resultat` s'écrit sans accent. Après le calcul, le contenu de la variable est affiché à l'écran avec l'instruction `printf`.

Le deuxième exemple est un calcul logique ou booléen :

```
Est_En_Retraite  = (age > 64) ;
```

La comparaison donnera VRAI si l'âge est strictement plus grand que 64, et FAUX sinon. `Est_En_Retraite` contient VRAI ou FAUX.

Il est possible de trouver des expressions en arguments d'une procédure. Les deux lignes du premier exemple pourraient être réécrites en une seule :

```
printf("%d", ((3*i)+(5-k))/18 ) ;
```

Les instructions

Une instruction peut être assimilée à une phrase du langage. Elle indique une action complète comme faire un calcul et ranger le résultat dans une variable (une affectation), afficher du texte à l'écran (un appel de procédure), faire plusieurs fois un calcul (une instruction répétitive ou boucle).

Il y a deux types d'instructions :

- les *instructions simples* ;
- les *instructions composées*.

Les instructions simples sont des « phrases simples » du langage. Il s'agit de :

- l'instruction d'affectation ;
- l'appel de procédure ;
- l'instruction `goto`.

Les instructions composées correspondent à des « phrases complexes ». Elles utilisent dans leur syntaxe d'autres instructions qui peuvent être simples ou composées. Il s'agit de :

- la séquence d'instructions ;
- les instructions conditionnelles ou tests ;
- les instructions répétitives ou boucles.

Voici l'exemple d'une instruction conditionnelle :

```
if (Est_En_Retraite)
{
   Nb_Annees_Retraite = (age - 64) ;
   printf("En retraite depuis %d années\n", Nb_Annees_Retraite) ;
}
```

ATTENTION

Une instruction du langage C est toujours terminée par un point-virgule (;). Ceci n'est cependant pas le cas des instructions composées quand elles utilisent des accolades `{}`. Il n'y a pas de point-virgule après une accolade.

3.5. Les commentaires

Commenter un programme fait partie des bonnes habitudes à prendre quand on s'initie à la programmation. Ce n'est pas une perte de temps ! Cela est indispensable pour la lisibilité d'un programme. Toutes les lignes encadrées par les caractères `/*` et `*/` sont des commentaires. On peut aussi utiliser les deux caractères `//` en début de ligne pour mettre en commentaire une seule ligne.

En voici deux exemples :

```
/* Calcul de la surface
   d'une sphère */
// Traitement de la date du jour
```

À NOTER

Utilisez les commentaires en début de programme pour informer sur l'auteur, la date de création et de mise à jour, et son rôle. Utilisez des lignes de tirets en commentaire pour séparer et visualiser facilement les sous-programmes : `/* ------------ */`

3.6. Les éléments du langage

Certains caractères et certains mots sont réservés, ce qui rend impossible leur usage dans le nom des variables. Appeler une variable `int` provoquerait une erreur de compilation. Voici la liste des caractères et des mots réservés.

Les Caractères spéciaux

Le nom des variables ne peut être constitué que de lettres alphabétiques (minuscules et majuscules), de chiffres et du caractère souligné (_). Les autres caractères sont interdits. Le tableau 1.1 présente quelques caractères spéciaux.

Tableau 1.1 : Quelques caractères spéciaux

Caractères spéciaux	Description
#	Préfixe de directive de compilation
;	Ponctuation : fin d'instruction
{ }	Ponctuation : regroupement d'instructions
( )	Liste d'arguments de sous-programme
,	Séparateur des arguments d'une liste
[ ]	Indice de tableau
/* */ ou //	Commentaires
'	Délimiteur d'une constante de type caractère
"	Délimiteur d'une constante chaîne de caractères
.	Champ d'un enregistrement (structure)
->	Pointeur vers un champ de structure
&&	ET logique
||	OU logique
!	NON logique
&	Opérateur binaire ET, ou adresse d'une variable
|	Opérateur binaire OU
^	Opérateur binaire OU Exclusif
<<	Opérateur binaire, décalage à gauche
>>	Opérateur binaire, décalage à droite