Alain Gibaud

Programmer en C++

Des premiers pas à la maîtrise de C++20

ellipses

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.

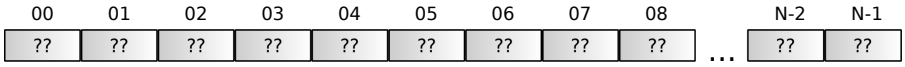

Figure 3.1 – Représentation simplifiée de la mémoire d'un ordinateur

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.

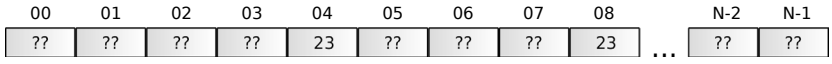

Figure 3.2 – Nouveau contenu de la mémoire

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