



Les bases de la programmation

21 janvier 2019

Table des matières

1.	Introduction	1
2.	Pourquoi la programmation ?	2
3.	Créer des programmes	4
3.1.	On programme généralement dans un contexte donné	4
3.2.	Un type de fichiers particulier: les programmes	5
4.	Au commencement était le code source	6
5.	Vers l'exécutable, et au-delà !	9
5.1.	Langages compilés	9
5.2.	Langages interprétés	11
5.3.	Langages à machine virtuelle	13
5.4.	Les limites du classement	14
6.	Il y a programmer, et bien programmer	15
6.1.	Les outils indispensables	15
6.2.	Les IDE	18
6.3.	Quelques outils supplémentaires	19
6.4.	Les bonnes pratiques	21
7.	Et maintenant ?	22
8.	Conclusion	26

1. Introduction

Dans ce cours, nous allons aborder les bases de la programmation d'un ordinateur, afin de constituer une introduction en douceur aux autres cours de programmation que compte le site, destinée à ceux qui n'ont jamais programmé.

On y verra ce qu'est la programmation informatique, et pourquoi elle est nécessaire. On abordera un certain nombre de notions essentielles à la compréhension d'un cours de programmation, comme les notions de langage informatique, d'exécutable, d'interpréteur ou encore de débogueur, et l'on en profitera pour enseigner le vocabulaire usuel de la programmation.

On abordera également quelques notions un peu plus avancées, comme les patrons de conception ou les bonnes pratiques en programmation, car elles sont transversales à tous les langages de programmation, et s'avéreront utiles un peu plus tard dans l'apprentissage. Enfin, on présentera brièvement les principaux langages de programmation, afin de vous orienter vers un choix qui vous correspond.

Certains concepts de ce cours vous paraîtront sans doute un peu abstraits : c'est normal, vous comprendrez mieux une fois qu'ils seront mis en application dans le premier langage que vous apprendrez. N'hésitez pas alors à revenir lire le passage correspondant de ce cours, pour mieux fixer les idées.



Prérequis

Connaissance suffisante de son ordinateur pour être autonome.

Objectifs

Faire comprendre en quoi consiste la programmation.

Introduire l'essentiel des notions et du vocabulaire usuels de la programmation.

Présenter les principaux langages de programmation, en particulier ceux qui sont enseignés sur Zeste de Savoir.


2. Pourquoi la programmation ?

Pour bien comprendre la raison d'être de la programmation, il vous faut déjà comprendre ce qu'est *réellement* un ordinateur en fin de compte. Beaucoup de gens pensent qu'un ordinateur est une grosse boîte noire, quelque peu mystique et particulièrement intelligente. Comme nous allons le voir, la réalité est tout autre.

D'abord, un ordinateur, ça regroupe énormément de choses, depuis l'ordinateur de bord de votre voiture jusqu'aux supercalculateurs de Météo France, en passant par les téléphones dits intelligents (*smartphone*, en anglais), les ordinateurs de bureau (fussent-ils Mac ou PC), les consoles de jeu et les caulettes programmables. Mais derrière cette variabilité se cache une réalité beaucoup plus simple : un ordinateur est composé des trois éléments suivants.

- Un **processeur**, dont nous allons reparler dans un instant.
- De la **mémoire**, qui peut prendre des formes extrêmement variées, mais dont l'utilité est toujours la même : garder un certain nombre d'informations (que l'on appelle des **données**), à disposition du processeur jusqu'au moment où il en aura besoin.
- Des **périphériques**, qui peuvent en théorie être absents. Un périphérique est un circuit électronique dédié à une tâche précise, et qui sert au processeur à communiquer avec le monde extérieur : par exemple, une carte graphique¹, pour afficher des choses sur un support visuel, une carte son, pour... produire du son, un clavier ou une souris, pour que l'utilisateur puisse lui communiquer des informations, ou encore un capteur de température, pour éviter la surchauffe.

L'élément central d'un ordinateur est donc son processeur. De quoi s'agit-il ? Simplement et uniquement d'une machine à calculer extrêmement rapide. En effet, bien qu'il fonctionne à l'électricité, un processeur n'est rien de plus qu'un automate : il est possible de lui donner des ordres, que l'on appelle des **instructions**, et une suite d'instructions strictement identiques produira systématiquement le même résultat, sans la moindre originalité. Les instructions qu'un processeur peut recevoir diffèrent d'un modèle à l'autre², et peuvent être très variées au sein d'un même modèle, mais elles se ramènent toutes à quatre types.

1. En fait, les cartes graphiques sont des ordinateurs à elles seules, puisqu'elles ont un processeur et de la mémoire qui leur sont propres... et qu'on peut leur brancher des périphériques, par exemple un écran ! Pour en savoir plus sur celles-ci, n'hésitez pas à aller lire [le cours de Mewtow](#)  sur le sujet.

2. La plupart des ordinateurs que vous rencontrerez utilisent des processeurs de la famille x86 du fabricant Intel, et son extension x86_64 initialement lancée par le fabricant AMD (les plus courants dans les ordinateurs de bureau), ou des processeurs de la famille ARM conçus par l'entreprise... ARM (les plus courants dans les téléphones intelligents).

2. Pourquoi la programmation ?

- Copier le contenu d'un emplacement en mémoire vers un autre emplacement en mémoire.
- Comparer le contenu d'un emplacement en mémoire avec une valeur prédéfinie.
- Effectuer des opérations mathématiques de base. Quand je dis de base, je parle vraiment de base : addition, multiplication, quelques fonctions trigonométriques, etc. Peut-être certains processeurs spécialisés peuvent-ils réaliser des opérations un peu plus sioux, mais l'immense majorité des processeurs ne connaît pas plus d'opérations mathématiques qu'un élève de Terminale S.
- Envoyer ou recevoir des données vers ou depuis un périphérique.

Et c'est tout. Tout de suite, ça casse le mythe, hein ? Et pourtant, c'est en profitant du fait que le processeur est capable d'obéir à des millions de telles instructions basiques chaque seconde que l'on parvient à écrire des jeux vidéos. **Programmer un ordinateur**, cela consiste donc à placer en mémoire les données adéquates pour que votre processeur et ses périphériques, contrôlés par lui, se comportent comme vous le voulez.



FIGURE 2. – Voilà à quoi ressemble un processeur, ici, le modèle Core i5 de chez Intel. Image tirée de [Wikimedia Commons](#) [↗](#), voyez là-bas pour les informations de droit d'auteur.

?

Mais c'est super, ça ! Et comment je fais pour savoir quelles données lui donner à manger ?

Les principaux processeurs du marché disposent de manuels d'utilisation, que l'on peut trouver assez facilement sur Internet. Le seul hic... c'est qu'ils font généralement autour de 3000 pages A4. Et attention ! Pas question de sauter des lignes : il faut maîtriser pleinement un chapitre pour ne serait-ce que comprendre le suivant. Et cela fait, vous ne saurez vous servir que de votre processeur, et même pas comment lui faire contrôler ses périphériques. En clair, si vous avez déjà du mal à comprendre les manuels de montage de chez [insérez ici le nom d'un vendeur de meubles en kit, éventuellement suédois], vous n'en avez pas fini.

3. Créer des programmes

Et je ne dis pas cela pour vous décourager : cela est vrai pour tout le monde. Fournir exactement les bonnes données au processeur pour qu'il fasse quelque chose d'un peu plus complexe qu'additionner deux nombres est *horriblement* difficile. C'est pourquoi, dans la vraie vie, on procède différemment... et c'est ce que vous allez découvrir dans ce cours !

3. Créer des programmes

Programmer, c'est avant tout créer des **programmes**. Mais là, on se mord un peu la queue : dans ce cas, ça veut dire quoi « programme », et comment on en crée un ? *A fortiori* plusieurs ?

À moins que vous ne lisiez ce cours depuis le futur, à l'aide d'une technologie qui m'est encore inconnue³, vous le consultez depuis votre navigateur Web (Firefox, Chrome, Internet Explorer, Edge, Safari, Opera...), qui est un programme. Tous et chacun de vos jeux vidéo sont des programmes. De même que votre traitement de texte, votre lecteur vidéo, la fenêtre qui s'affiche lorsque vous consultez une image, ou encore l'application « réveil » de votre éventuel téléphone.

3.1. On programme généralement dans un contexte donné

Tous ces programmes tournent sur un **système d'exploitation**⁴ : ce peut être Windows, Linux⁵, OS X, le plus couramment, Android ou iOS (ou éventuellement Firefox OS) si vous êtes sur téléphone ou tablette, ou encore OpenBSD, FreeBSD ou Solaris pour les plus explorateurs d'entre vous.

Celui-ci est une sorte de super-programme, qui fait le pont entre le matériel et les autres programmes. Le qualifier de dictateur bienveillant serait plus exact : il a la mainmise totale sur l'accès au matériel, c'est lui qui décide sur quoi le processeur va travailler, et il fournit un certain nombre de services auxquels les programmes devront avoir recours pour toutes leurs tâches qui concernent les ressources communes (en particulier, la mémoire et l'utilisation des périphériques).

Son utilité première (outre la gestion des conflits entre programmes pour l'accès aux ressources), ce sont précisément ces services. Il fournit en réalité des abstractions, que les programmes peuvent utiliser pour faire fonctionner le matériel, quel que soit le matériel effectivement utilisé.

Par exemple, malgré leur grande diversité, il est un service que tous les systèmes d'exploitation offrent, sans exception : un **système de fichiers**. Il s'agit d'une manière de structurer les données conservées en mémoire, en particulier sur les mémoires « longue durée » comme les disques durs, clés USB ou autres CD, de manière à pouvoir retrouver l'intégralité d'un ensemble cohérent de données. En effet, il serait dommage de mettre de côté une image et que votre ordinateur n'en retrouve que la moitié !

Le système définit ainsi des **fichiers**, qui sont des ensembles cohérents de données. Il s'agit là d'une abstraction destinée à faciliter le travail du programmeur : le disque dur ne sait pas ce

3. Ou plus prosaïquement, que vous l'ayez imprimé sur papier, au mépris des arbres et des petits oiseaux, bande de rabouins, mais on vous aime quand même, alors faites des copies pour vos copains.

4. Enfin *presque* tous les programmes.

5. Alias GNU/Linux pour les plus fervents partisans de Richard Stallman.

3. Créer des programmes

qu'est un fichier, il ne connaît que les secteurs, des blocs de mémoire d'une taille déterminée. Un programme va se contenter de demander à lire tel fichier, et c'est le système d'exploitation qui convertira cela en instructions destinées au modèle précis de disque dur présent dans la machine : à partir d'un nom de fichier, il demande au disque le contenu de certains secteurs afin de reconstituer ledit fichier. De cette manière, ce programme pourra fonctionner sur une autre machine, et avec un autre modèle de disque dur, pour peu que le même système d'exploitation y soit installé : pas besoin de le réécrire.

Et on touche là à la limite des systèmes d'exploitation : ils sont largement incompatibles entre eux. Un programme destiné à fonctionner sous Windows ne pourra pas fonctionner sous un autre système d'exploitation, et si on veut le rendre disponible ailleurs que sous Windows (le terme technique est « **porter** un programme sur un système d'exploitation donné »), il faudra réécrire toutes les parties qui font appel aux services du système d'exploitation, ce qui peut s'avérer fastidieux.



Malgré cet inconvénient, il est presque toujours plus pratique de créer un programme pour un système d'exploitation donné que de faire en sorte qu'il fonctionne sans faire appel au système d'exploitation.

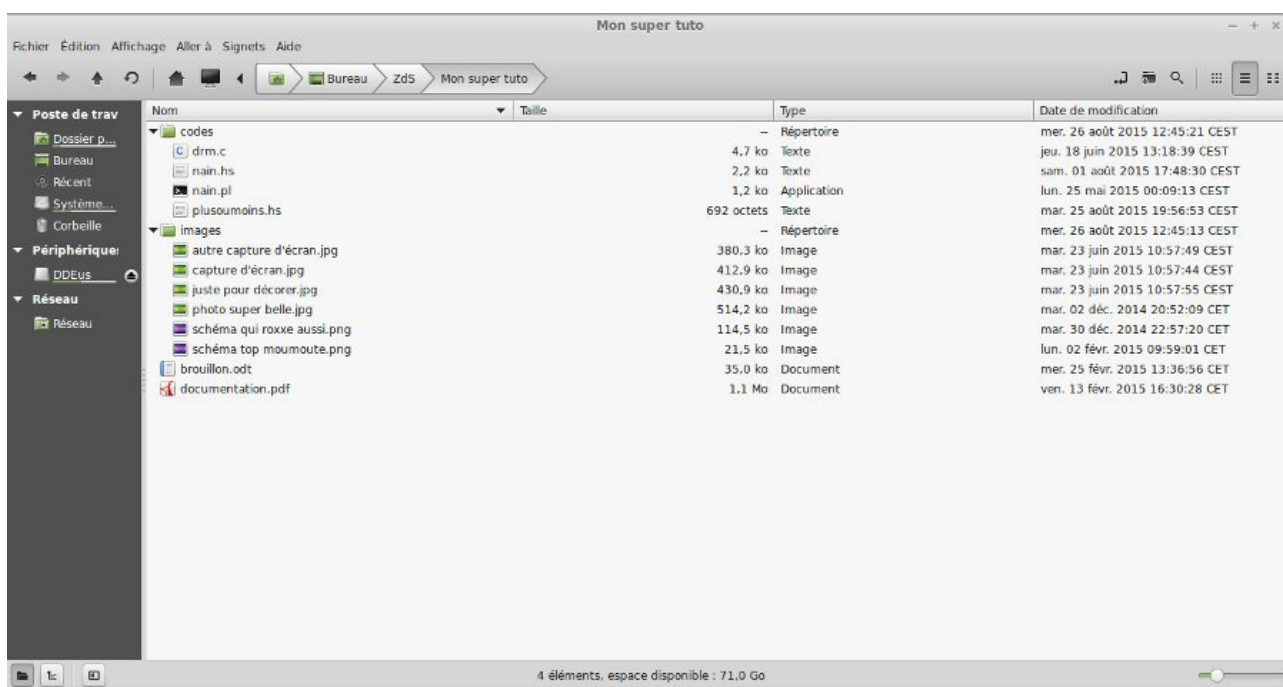



FIGURE 3. – On affiche généralement les fichiers sous une forme arborescente, au moyen d'un **gestionnaire de fichiers** (ici, [Nemo](#) ) , qui est lui-même un programme.

3.2. Un type de fichiers particulier : les programmes

Mais revenons sur ces fichiers, car ils se divisent en deux catégories. D'une part, les **fichiers de données**, qui stockent de l'information, et sont destinés à être utilisés comme tels : une image, une vidéo, un son, un document de bureautique, un livre électronique, etc. D'autre part,

4. Au commencement était le code source

les **fichiers exécutables**, qui stockent le moyen de faire réaliser un certain nombre d'actions à l'ordinateur.

Et c'est là que l'on retrouve nos programmes ! En effet, quand on fait faire à un programme ce pour quoi il est conçu, on dit qu'on l'exécute⁶. Mais n'en restons pas là, il existe deux sortes de fichiers exécutables. Les premiers s'appellent les **bibliothèques logicielles**⁷.

Une bibliothèque logicielle est une collection de petits outils destinés à une tâche bien précise qui sont partagés, car ils peuvent être utiles à de nombreux programmes. En effet, des centaines de programmes peuvent avoir besoin d'afficher une barre de menus en haut de leur fenêtre, ou de décompresser une archive ZIP. Il serait fastidieux et totalement contre-productif que chaque programme qui en a besoin réécrive et contienne les instructions permettant de réaliser cette tâche. On regroupe donc ces instructions dans une bibliothèque, et les programmes qui en ont besoin se contentent de l'appeler le moment venu.

Il n'est donc pas possible d'exécuter directement une bibliothèque. Il faut nécessairement l'appeler depuis un autre programme, conçu pour réaliser des tâches qui ne serviront qu'à lui. On l'appelle un **logiciel** et il constitue notre seconde catégorie de fichiers exécutables. Tous les programmes dont je vous ai parlé plus haut (navigateur, traitement de texte, etc.) sont des logiciels, qui s'appuient sur des bibliothèques mais ne servent d'appui à aucun autre programme.

Pour terminer sur le vocabulaire, on distingue les **bibliothèques système**, mises à disposition nativement par le système d'exploitation, et les **bibliothèques tierces**, qu'il faut installer en même temps que l'on installe un logiciel qui en a besoin.

i

Une des principales raisons pour lesquelles un programme ne peut pas fonctionner sous un autre système d'exploitation, c'est qu'il n'est pas constitué de code machine brut : il est organisé selon un format spécifique (appelé « format d'exécutable », sans aucune originalité) qui est propre à chaque système, quoique certains partagent un même format. Par exemple, Linux et les BSD utilisent le format ELF, tandis que Windows utilise le format PE.

Pour résumer, programmer, cela consiste à générer un ou des fichiers exécutables pour un système d'exploitation donné (ou des données exécutables sans système d'exploitation, si vous êtes un Rambo de l'informatique), fichiers qui seront aptes à réaliser un ensemble cohérent de tâches. Pour cela, vous aurez besoin de plusieurs choses, et c'est ce que nous allons voir dès maintenant.

4. Au commencement était le code source

Pour créer vos propres programmes, il vous faudra des outils, qui seront détaillés dans les deux sections suivantes. Mais il vous faudra également acquérir des connaissances brutes et, en premier lieu, apprendre à maîtriser un **langage de programmation**.

6. Eh oui, sans procès équitable ni dernières volontés ! C'est dire combien les informaticiens sont des sagouins...

7. La plupart du temps, elles ont l'extension **.dll** sous Windows, **.dylib** ou **.so** sous OS X et **.so** sous Linux.

4. Au commencement était le code source

?

Mais *qu'es acò* encore ?

Un langage de programmation, c'est un ensemble de conventions et d'abstractions, qui permettent d'écrire ce que l'on veut faire faire à son ordinateur sous une forme plus compréhensible par un humain. Par exemple, pour calculer la somme de tous les nombres de 1 à 100, si l'on se limite strictement à ce dont est capable un processeur, cela prendra la forme de la suite d'instructions suivante.

```
1 METS la valeur 1 dans l'emplacement mémoire MEM1
2 METS la valeur 0 dans l'emplacement mémoire MEM2
3 ADDITIONNE la valeur de l'emplacement MEM2 à celle de MEM1 et
  stocke le résultat dans MEM2
4 ADDITIONNE 1 à MEM1
5 COMPARE MEM1 à 101
6 SI les deux sont différents, RETOURNE à la troisième instruction
```

Vous avouerez que ce n'est pas très sexy. Il s'agit pourtant de la forme que prend le plus basique de tous les langages de programmation, l'**assembleur**. Celui-ci donne une maîtrise presque totale sur sa machine, car il n'est que la transcription des instructions brutes sous une forme humainement compréhensible. Mais en contrepartie, il est spécifique à chaque famille de processeurs, et la moindre action intelligente est excessivement difficile à mettre en œuvre. Un langage de programmation plus évolué (c'est-à-dire tous les autres) permettra d'écrire plutôt ce qui suit.

```
1 RÉSERVE deux emplacements mémoire, MEM1 et MEM2, valant
  respectivement 1 et 0
2 TANT QUE MEM1 est inférieur ou égal à 100
3   AJOUTE MEM2 et MEM1 et stocke le résultat dans MEM2
4   AJOUTE 1 à MEM1
```

Ou encore, dans d'autres langages, il est également possible d'écrire tout simplement la ligne suivante.

```
1 La somme de tous les nombres de 1 à N vaut N fois N+1 divisé par 2.
```

Cela facilite évidemment *énormément* la programmation d'un ordinateur. Au lieu d'apprendre les trois mille et plus pages du manuel d'utilisation du processeur, on apprend un langage de programmation. Pour programmer, il suffit alors d'écrire un texte⁸, qui décrit dans un formalisme donné ce que fait votre programme, et que l'on appelle un **code source** : un programme spécialement dédié, que l'on appelle selon les cas un **compilateur** ou un **interpréteur** (voir

4. Au commencement était le code source

section suivante), va ensuite se charger de transformer votre code source en les données adéquates pour faire fonctionner votre ordinateur comme vous l'entendez.

```
1  #![feature(collections)]
2
3  macro_rules! create_animal(
4      ($name:ident, $kind:expr) => {
5          pub struct $name {
6              name: String,
7              kind: String
8          }
9
10         impl $name {
11             pub fn new(name: &str) -> $name {
12                 $name {name: String::from_str(name), kind: String::from_str($kind)}
13             }
14         }
15
16         impl ::Animal for $name {
17             fn get_name(&self) -> &str { &self.name }
18             fn get_kind(&self) -> &str { &self.kind }
19             fn introduce(&self) { println!("I am {} of the proud {} race !", self.name, self.kind); }
20         }
21     );
22 }
23
24 trait Animal {
25     fn get_name(&self) -> &str;
26     fn get_kind(&self) -> &str;
27     fn introduce(&self);
28 }
29
30 create_animal!(Dog, "Canid");
31 create_animal!(Cat, "Feline");
32 create_animal!(Lion, "Feline");
33 create_animal!(Wolf, "Canid");
34
35 fn main() {
36     Dog::new("Plutot").introduce();
37     Cat::new("Felix").introduce();
38     Lion::new("Simba").introduce();
39     Wolf::new("Wolfi").introduce();
40 }
```

FIGURE 4. – Voici un exemple de code source, ici dans le langage Rust et sous l'éditeur [Sublime Text](#) [↗](#).

Comme je l'ai dit, ce langage est *formel*. En effet, l'ordinateur n'étant jamais qu'une machine à calculer, pas très intelligente de base, le compilateur/interpréteur ne peut pas faire de miracle : contrairement à un langage humain, où les sous-entendus, connotations et supplétifs permettent de se faire comprendre même sans dire exactement les choses, les langages informatiques n'ont qu'un nombre limité de moyens d'expression, qui doivent être utilisés d'une manière spécifique. On appelle cela la **syntaxe** d'un langage, et si on ne la respecte pas à la lettre, le compilateur/interpréteur refusera de fonctionner.

Le bon côté, en revanche, c'est qu'un même code source pourra être compilé ou interprété de manière à fonctionner sur plusieurs processeurs différents : là où chaque processeur doit être contrôlé au moyen de ses instructions spécifiques qui ne marcheront pas ailleurs, un programme écrit dans un langage de programmation autre que l'assembleur sera réutilisable sur d'autres types de machine, pour peu que le compilateur/interpréteur soit disponible sur cette plate-forme.

Il existe au total des milliers de langages différents. Certains sont proches du fonctionnement réel de la machine, d'autres très éloignés : on appelle les premiers des langages de **bas niveau** et les autres des langages de **haut niveau**⁹. Certains sont spécialisés pour être efficaces dans un type donné de tâche, d'autres sont polyvalents. Certains sont Turing-complets (en très résumé,

8. De très rares langages exotiques, comme le [Piet](#) [↗](#), s'écrivent sous une forme autre que le texte. Ils n'ont cependant pas d'utilité pratique et sont là pour pousser la logique des langages de programmation dans ses derniers retranchements.

5. Vers l'exécutable, et au-delà !

cela signifie qu'ils sont en théorie capables de servir à programmer n'importe quoi), d'autres non.

Mais tous partagent des abstractions, des conventions, des concepts qui permettent de les classer en grandes familles, qui regrouperont des langages ayant plus ou moins le même « patron de conception » : cela s'appelle des **paradigmes** de programmation. Apprendre un langage est beaucoup plus simple lorsque l'on connaît déjà un langage qui suit le même paradigme : en effet, il n'est plus nécessaire d'apprendre toute la théorie sur laquelle repose le langage, mais uniquement son implémentation effective.

i

Il n'appartient pas à ce cours de vous présenter les différents paradigmes de programmation. Afin que vous connaissiez le vocabulaire, les principaux paradigmes sont la programmation impérative (ou procédurale), la programmation orientée objet qui en est une extension, la programmation fonctionnelle, la programmation logique et la programmation concurrente, qui est transversale aux quatre autres.

Outre la syntaxe, l'apprentissage d'un langage inclut généralement la connaissance correcte de sa **bibliothèque standard**. En effet, il serait très pénible de réinventer la roue chaque fois que l'on veut créer un nouveau programme. Chaque langage est donc généralement livré avec un ensemble de morceaux de code qui permettent de réaliser des tâches courantes, comme déterminer si une liste d'éléments contient une valeur donnée, en particulier des tâches qui doivent être réalisées différemment d'un système d'exploitation à l'autre, comme ouvrir un fichier ou afficher du texte à l'écran.

Tous ces morceaux de code pris ensemble s'appellent la bibliothèque standard du langage, et il est possible d'appeler chacun d'eux facilement depuis votre programme, via une syntaxe particulière. En particulier, les tâches spécifiques à chaque système d'exploitation pourront généralement être appelées au moyen d'une commande unique, et c'est la bibliothèque standard qui se chargera de déterminer comment, précisément, elle doit les réaliser.

5. Vers l'exécutable, et au-delà !

Une fois que votre code source est rédigé, vous désirez légitimement pouvoir exécuter votre programme. Les langages de programmation se répartissent en trois catégories en fonction de la manière dont cette étape s'opère.

5.1. Langages compilés

La **compilation** consiste à transformer le code source en un fichier exécutable, généralement unique, qui est immédiatement utilisable. Cette transformation est effectuée par un **compilateur**, et si tout est généralement transparent pour vous, le processus est en réalité assez complexe et passe par plusieurs étapes¹⁰.

9. Notez bien que la distinction entre ces deux catégories est très floue, et n'est pas réellement utile dans la pratique. Il s'agit plus d'un moyen de situer rapidement le degré d'abstraction d'un langage.

10. Si vous voulez en savoir plus, demandez à votre programmeur C ou C++ préféré ce que sont les fichiers objet ou l'édition de liens.

5. Vers l'exécutable, et au-delà !

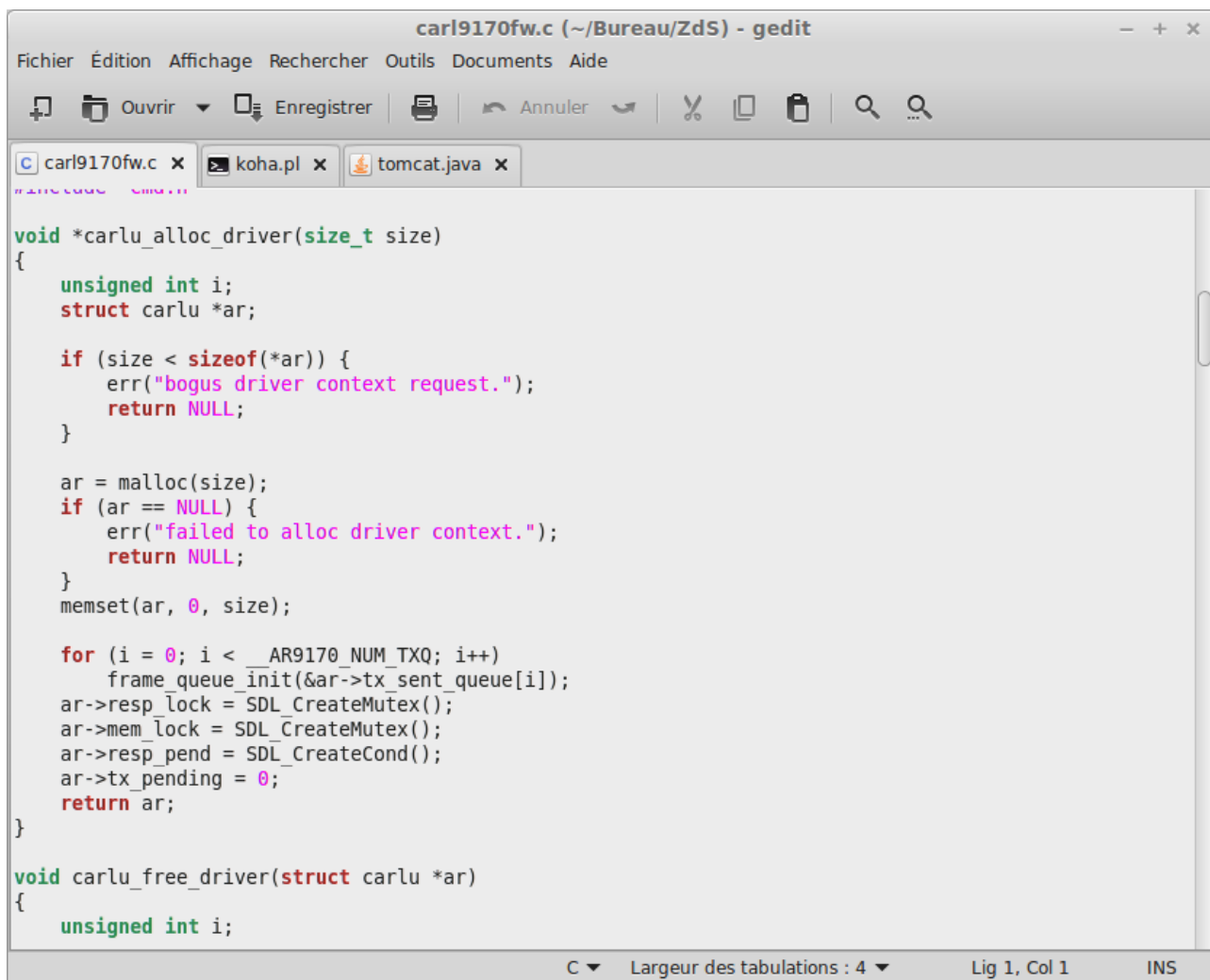
La compilation présente l'avantage que le fichier exécutable généré sera optimisé pour votre machine, ce qui signifie qu'il sera plus rapide que s'il était interprété, et qu'il prend généralement moins de place sur le disque dur que son code source. Cela peut s'avérer un inconvénient si votre programme est destiné à être diffusé auprès d'autres gens : il faudra soit leur fournir le code source et les laisser le compiler par leurs propres moyens¹¹, soit fournir un exécutable différent pour chaque système d'exploitation et pour chaque type de processeur, et donc répéter autant de fois la compilation à chaque mise à jour du programme, ce qui devient très vite fastidieux.

La différence dans la vitesse d'exécution est loin d'être négligeable : en général, un programme en langage compilé tournera environ dix fois plus vite qu'un programme en langage interprété (toutes autres choses égales), et cette différence peut monter jusqu'à un rapport de un à cent sur certaines tâches. Il faut toutefois noter que cette métrique n'est pas toujours pertinente. En effet, si une tâche s'effectue en une milliseconde dans un langage et en cent dans un autre, la différence n'est pas humainement perceptible, et ne devient critique que s'il faut la répéter des millions de fois (par exemple, il faut impérativement que chaque image d'un film soit décodée et affichée à l'écran en moins d'un 24^e de seconde).

Par ailleurs, certaines tâches sont tellement longues qu'elles rendent les différences de vitesse d'exécution des tâches qui les entourent négligeables : lire un fichier sur un disque dur est *très* long, à l'échelle d'un processeur moderne, aussi, il importe généralement peu que le programme choisisse en une ou en dix millisecondes quel fichier il va lire. Dans le même ordre d'idée, si un programme interagit avec un utilisateur humain, celui-ci sera toujours très très lent à réagir comparé à l'ordinateur, et qu'importe (dans une certaine mesure) la vitesse du programme, il passera l'essentiel de son temps à attendre que l'humain ait le temps de réagir.

11. Autant dire que s'il s'agit d'un programme pour aider votre vieille tata Germaine à faire ses comptes, vous oubliez...

5. Vers l'exécutable, et au-delà !



```
carlu9170fw.c (~/Bureau/ZdS) - gedit
Fichier Édition Affichage Rechercher Outils Documents Aide
Ouvrir Enregistrer Annuler
carlu9170fw.c x koha.pl x tomcat.java x

void *carlu_alloc_driver(size_t size)
{
    unsigned int i;
    struct carlu *ar;

    if (size < sizeof(*ar)) {
        err("bogus driver context request.");
        return NULL;
    }

    ar = malloc(size);
    if (ar == NULL) {
        err("failed to alloc driver context.");
        return NULL;
    }
    memset(ar, 0, size);

    for (i = 0; i < _AR9170_NUM_TXQ; i++)
        frame_queue_init(&ar->tx_sent_queue[i]);
    ar->resp_lock = SDL_CreateMutex();
    ar->mem_lock = SDL_CreateMutex();
    ar->resp_pend = SDL_CreateCond();
    ar->tx_pending = 0;
    return ar;
}

void carlu_free_driver(struct carlu *ar)
{
    unsigned int i;
```

FIGURE 5. – Le C est un langage compilé. Code tiré de [CARL9170](#) .

5.2. Langages interprétés

Dans le cas des langages interprétés, en revanche, on fournit le code source à l'**interpréteur**, qui exécute directement le programme : plus besoin de vous soucier du système d'exploitation ou du type de processeur des utilisateurs de votre programme, vous diffusez directement le code source, qui est prévu pour !

En pratique, ce n'est pas tout à fait vrai¹². L'interpréteur est indispensable au processus, et doit donc être installé sur l'ordinateur de votre utilisateur. Ce qui constitue un obstacle quand l'interpréteur n'existe pas pour son système d'exploitation et/ou son processeur. Voire, de manière plus fourbe, la perspective de devoir installer un (gros) logiciel annexe peut décourager les gens d'utiliser votre programme : par exemple, l'interpréteur Perl est installé par défaut sous Linux, mais absent sous Windows, ce qui fait que très peu de gens sous Windows utilisent des programmes en Perl, alors qu'ils sont assez courants sous Linux. Et par conséquent, quelqu'un qui voudrait diffuser son programme sous Windows *et* sous Linux choisira rarement de programmer en Perl.

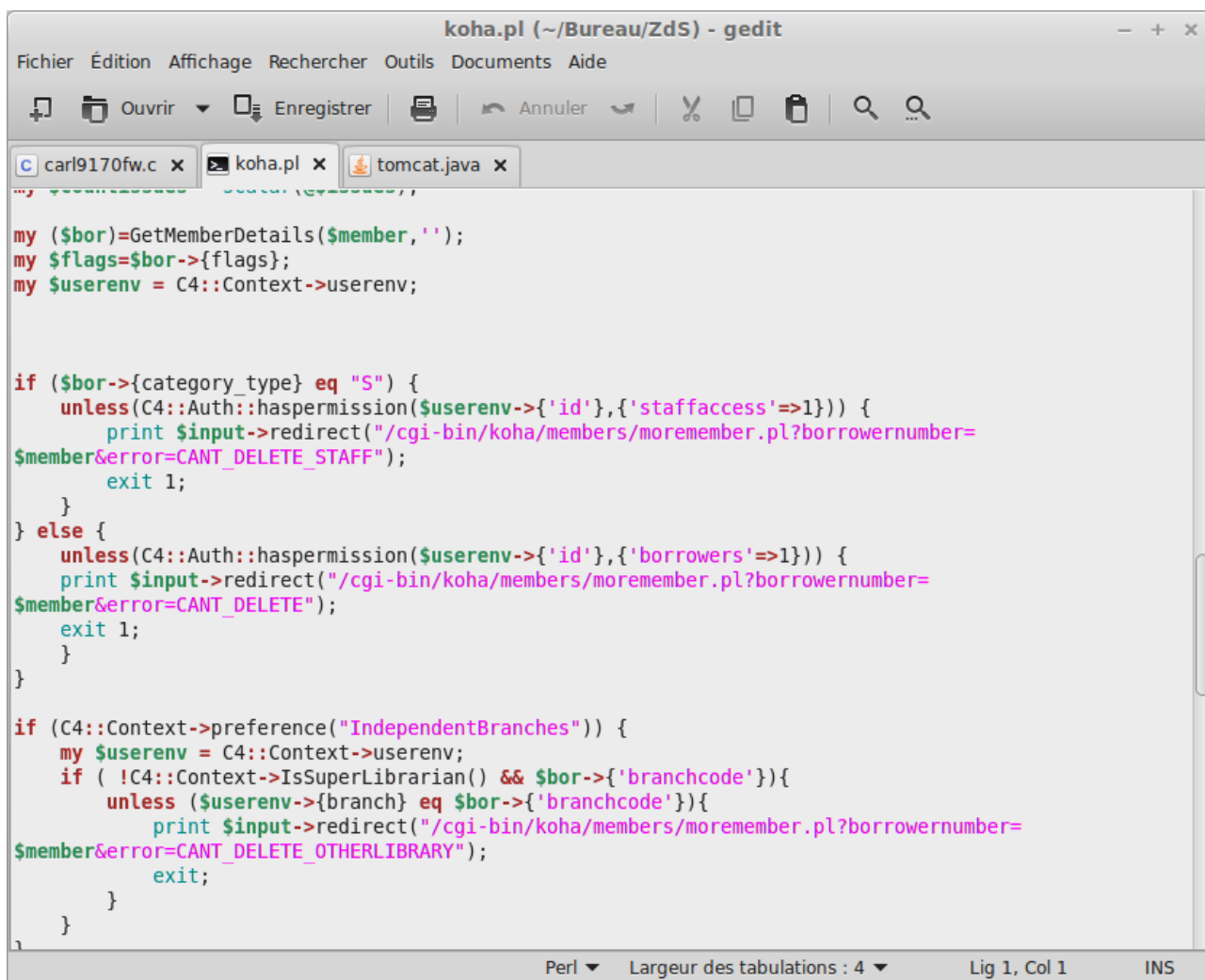
12. Je laisse de côté le fait que certains langages interprétés permettent d'envoyer directement une demande au système d'exploitation, et qu'un code source utilisant cette possibilité ne sera pas portable. En effet, l'usage

5. Vers l'exécutable, et au-delà !

Par ailleurs, comme le code source doit être « traduit » en code machine à chaque exécution, les langages interprétés sont souvent lents comparés à des langages compilés équivalents. Ce point est à nuancer légèrement. En effet, les interpréteurs optimisent moins le code machine qu'ils génèrent, ce qui les rend plus lents à l'exécution, mais le processus de génération du code machine est du coup plus rapide que celui des compilateurs. Cela a pour conséquence que, pendant le développement du programme, une période où il faut très souvent le tester car on est constamment en train de le modifier, on perd moins de temps à le rendre exécutable avec un langage interprété qu'avec un langage compilé.

En outre, même s'il existe des moyens de rendre un code source presque illisible par un humain (on parle d'**obscurcissement**, ou d'**obfuscation** en anglais), écrire un programme dans un langage interprété implique nécessairement de diffuser ce code source à ses utilisateurs, qui pourront alors le décortiquer et en faire ce que bon leur semble : selon votre conception de la propriété intellectuelle et des moyens de la défendre, cela peut s'avérer problématique.

Enfin, un projet un tant soit peu important nécessite un code source comportant de nombreux fichiers, qu'il faudra tous diffuser, alors qu'un programme compilé tient dans un seul fichier. Cela est rarement important en pratique, cependant.



```
my ($bor)=GetMemberDetails($member,'');
my $flags=$bor->{flags};
my $userenv = C4::Context->userenv;

if ($bor->{category_type} eq "S") {
    unless(C4::Auth::haspermission($userenv->{'id'},{'staffaccess'=>1})) {
        print $input->redirect("/cgi-bin/koha/members/moremember.pl?borrowernumber=
$member&error=CANT_DELETE_STAFF");
        exit 1;
    }
} else {
    unless(C4::Auth::haspermission($userenv->{'id'},{'borrowers'=>1})) {
        print $input->redirect("/cgi-bin/koha/members/moremember.pl?borrowernumber=
$member&error=CANT_DELETE");
        exit 1;
    }
}

if (C4::Context->preference("IndependentBranches")) {
    my $userenv = C4::Context->userenv;
    if ( !C4::Context->IsSuperLibrarian() && $bor->{'branchcode'}){
        unless ($userenv->{branch} eq $bor->{'branchcode'}){
            print $input->redirect("/cgi-bin/koha/members/moremember.pl?borrowernumber=
$member&error=CANT_DELETE_OTHERLIBRARY");
            exit;
        }
    }
}
```

FIGURE 5. – Le Perl est un langage interprété. Code tiré de [Koha](#) .

de cette fonctionnalité est généralement considérée comme une mauvaise chose.

5.3. Langages à machine virtuelle

Et entre les deux, on trouve les **langages à machine virtuelle**, souvent raccourcis en « langages à VM » d'après l'anglais *virtual machine*, qui ont un pied dans chaque monde. Le principe en est de compiler le code source non pas en code machine compréhensible par un processeur donné, mais en un code machine « fictif » (généralement appelé *bytecode*) qui sera lui-même interprété par la machine virtuelle associée au langage.

Un tel langage tire ses avantages et inconvénients des deux autres catégories. Comme avec les langages interprétés, le programme compilé en *bytecode* peut-être exécuté sur n'importe quel système d'exploitation et processeur... pourvu que la machine virtuelle soit disponible pour cette combinaison.

En revanche, comme il y a eu une compilation en amont, le programme est plus rapide que dans un langage interprété équivalent, et parvient souvent à atteindre des vitesses similaires à celle d'un langage compilé en code machine « réel ». Cela se paye cependant souvent par le fait que cette machine virtuelle peut être assez gourmande en ressources, notamment en mémoire.

Enfin, de manière plus anecdotique, il est possible de créer de nouveaux langages qui se compilent vers le même *bytecode* qu'un autre langage pré-existant, rendant ainsi très facile leur coopération. Par exemple, les langages Clojure et Frege compilent tous deux vers le *bytecode* de Java : il s'agit de langages fonctionnels, radicalement différents de Java dans leur conception ; il est alors envisageable d'écrire les différentes parties d'un programme avec celui des langages qui paraît le plus adapté, et de faire fonctionner tous ces morceaux ensemble sur la machine virtuelle.

Cette possibilité existe aussi dans une certaine mesure dans les langages compilés, puisqu'ils compilent tous vers un même code machine.

5. Vers l'exécutable, et au-delà !

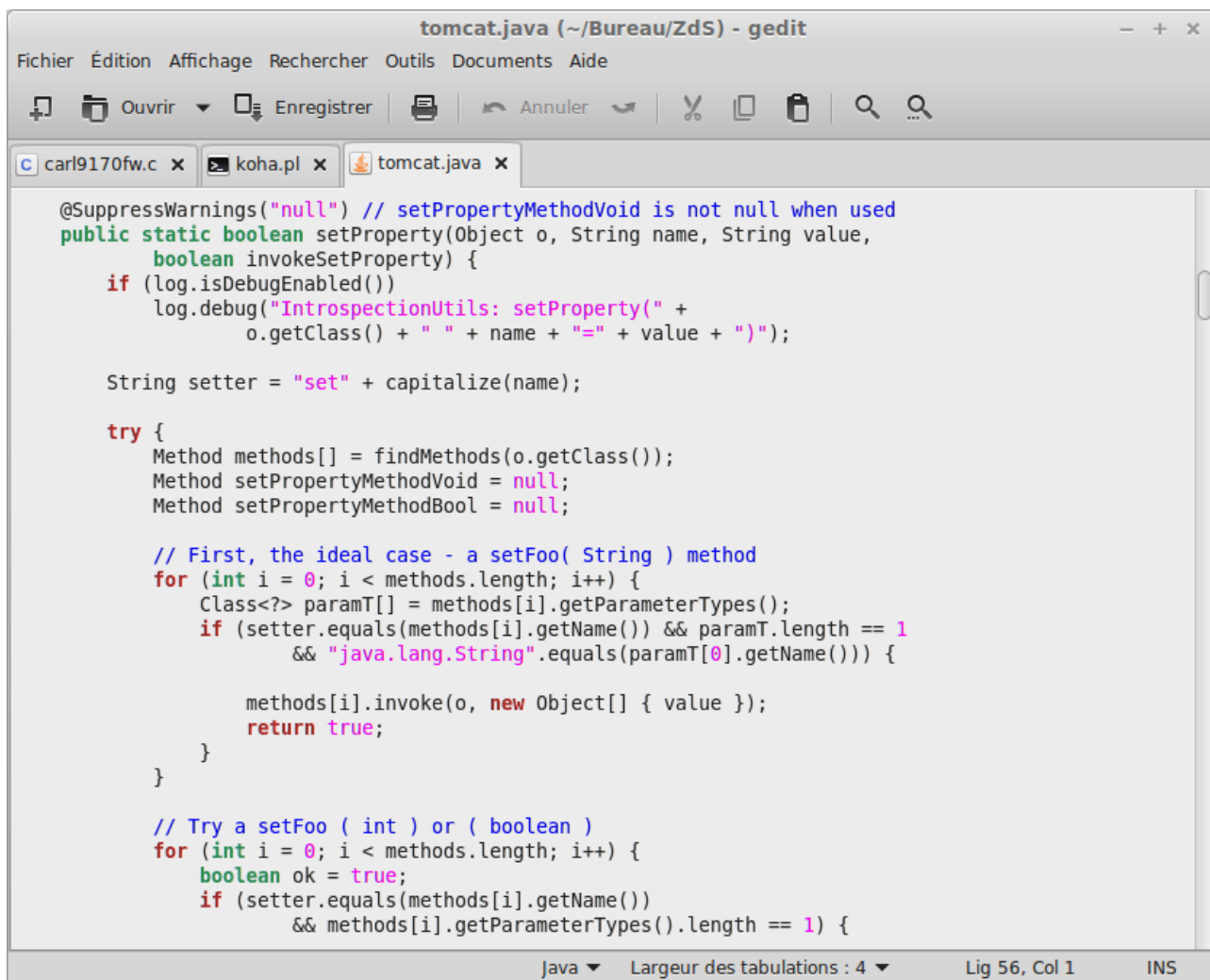


FIGURE 5. – Le Java est un langage qui compile vers une machine virtuelle. Code tiré de [Apache Tomcat](#). Comme vous pouvez le voir, ces trois langages (C, Perl et Java) ont comme un air de famille : c'est qu'ils appartiennent tous à la famille des langages impératifs.

5.4. Les limites du classement

Dans la pratique, cette séparation en trois est loin d'être aussi nette. Voici quelques exemples de situations où elle est mise en défaut.

Les langages interprétés utilisent la plupart du temps une compilation « à la volée ». Cela signifie que les langages compilés peuvent faire de même, et certains ne s'en privent pas. Par exemple, Haskell est un langage compilé, mais il possède un interpréteur qui permet de bénéficier de tous les avantages d'un langage interprété en termes de rapidité de développement.

Dans l'autre sens, certains langages sont interprétés la plupart du temps, mais offrent aussi la possibilité d'être compilés. C'est par exemple le cas de Python.

Stricto sensu, la compilation ne transforme pas nécessairement un code source dans un langage vers du code machine, réel ou virtuel : elle peut aussi le transformer vers un autre langage, qui sera lui-même compilé ou interprété. Par exemple, Elm est un langage fonctionnel, avec

6. Il y a programmer, et bien programmer

tout l'intérêt que cela peut avoir, qui est ensuite compilé en JavaScript, un langage impératif et interprété massivement utilisé dans le développement Web.

En particulier, un certain nombre de langages sont compilés vers un langage de plus bas niveau. À commencer par le C, qui est compilé en assembleur, lequel est ensuite compilé en code machine (en vérité, il est d'usage de dire qu'il est **assemblé** en code machine). Et d'autres langages, comme Haskell, compilent vers du C. Cela a pour conséquence de faciliter l'utilisation de bibliothèques écrites dans le langage de destination au sein de programmes écrits dans le langage source. Et, de manière plus anecdotique, de glisser des optimisations de dernière minute en écrivant certains bouts de code, critiques en terme de performance, directement dans le langage de destination.

Enfin, pour des raisons de portabilité de l'interpréteur, il n'est pas rare que les langages interprétés ne soient pas transformés directement en code machine de l'ordinateur sur lequel ils sont exécutés, mais passent par un *bytecode* intermédiaire, qui est exécuté par une machine virtuelle intégrée à l'interpréteur.

6. Il y a programmer, et bien programmer

6.1. Les outils indispensables

Comme vous vous en doutez peut-être, pour programmer, il nous faut plusieurs outils. En fait, malgré toute la multitude d'outils disponibles pour faire tout un tas de tâches, seuls quelques uns sont vraiment absolument nécessaires. Découvrons-les ensemble.

6.1.1. Un éditeur de texte

Cet outil est celui qui va nous permettre d'écrire notre code source. C'est un outil commun, indispensable pour programmer dans n'importe quel langage. En effet, même si certains langages offrent un interpréteur en ligne de commande qui permet de tester rapidement quelques lignes, dès lors que l'on veut écrire un programme un poil plus conséquent, et surtout qu'on veut **sauvegarder le code source**, passer par un éditeur de texte devient incontournable !



On parle bien ici d'un **éditeur** de texte, et non d'un **traitement** de texte, comme Word ou LibreOffice.

En théorie, n'importe quel éditeur ferait l'affaire : Notepad++, vim, gedit, Emacs, bloc-note de Windows, nano, Kate, etc. En pratique, plus notre éditeur est complet et configurable, mieux c'est. En effet, lorsqu'on programme, on apprécie d'avoir certaines fonctionnalités comme :

- **l'auto-complétion**, alias l'éditeur complète automatiquement le code que l'on est en train d'écrire, soit en suivant la logique interne du langage (par exemple, une parenthèse ouverte devra nécessairement être fermée, alors autant ajouter dès le départ la parenthèse fermante), soit en vous proposant des noms d'élément que vous avez déjà utilisés (par exemple, si une des portions de votre code s'appelle « ornithorynque », l'éditeur vous proposera de compléter automatiquement dès que vous taperez « orni ») ;

6. Il y a programmer, et bien programmer

- la **coloration syntaxique**, alias l'éditeur affiche les caractères du code dans différentes couleurs en fonction de ce qu'ils signifient dans le langage ;
- la **recherche / édition** pour rechercher et remplacer rapidement des portions de code, par exemple pour changer le nom d'un élément à tous les endroits où il est utilisé ;
- l'**indentation** automatique et réglable, alias l'éditeur formate automatiquement le code source pour aligner différentes portions faisant corps.

i

Notez que si les deuxième et troisième points sont presque universellement appréciés, tout le monde n'aime pas nécessairement les deux autres, certains trouvant même ces fonctionnalités gênantes. Si vous êtes débutant, le mieux est peut-être de prendre un éditeur qui offre toutes ces possibilités, et de les désactiver si vous n'en avez pas l'utilité.

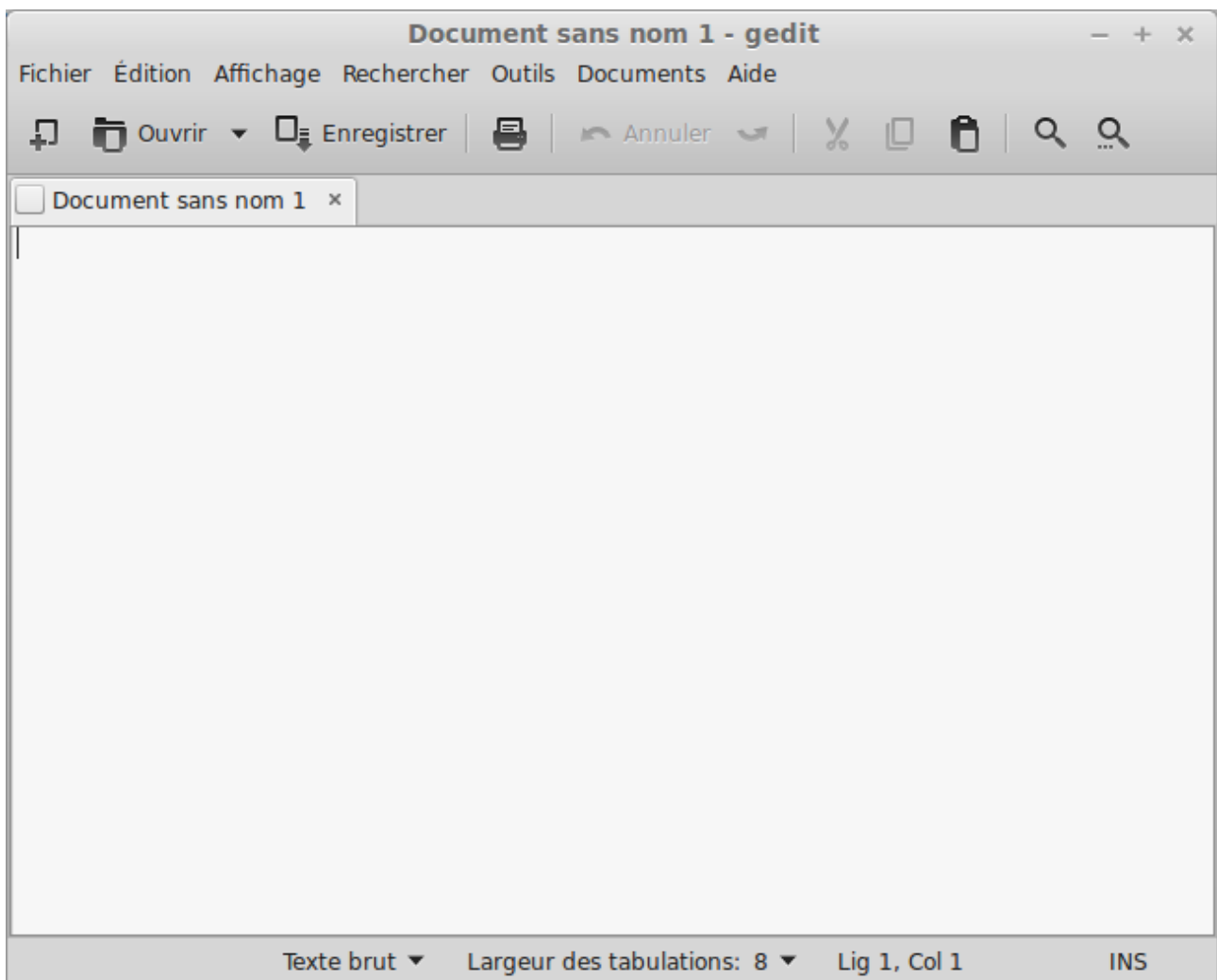


FIGURE 6. – L'éditeur de texte [gedit](#) est livré par défaut avec de nombreuses distributions GNU/Linux.

6.1.2. Un compilateur ou interpréteur

Cet outil est directement lié au langage que vous souhaitez apprendre, aussi aurez-vous beaucoup moins de marge de manœuvre pour le choisir. Par exemple, si en C++ vous avez le choix parmi

6. Il y a programmer, et bien programmer

gcc, Clang, LLVM, etc., au contraire, en Python, vous devrez utiliser l'interpréteur Python. Si vous utilisez un Unix/ide, tel que les BSD ou GNU/Linux, certains de ces outils, comme gcc, sont fournis par défaut. Sinon, ce sera à vous d'aller sur le site officiel du langage ou de l'outil pour le télécharger, l'installer et le configurer.

Si cela vous paraît compliqué, ne vous inquiétez pas : la plupart des tutoriels de programmation vous indiqueront quels outils utiliser et télécharger et comment le faire.

6.1.3. Un débogueur

Les développeurs sont susceptibles, comme tous les humains, de faire des erreurs. Certaines sont tellement vicieuses que les retrouver et les corriger prend du temps. Le débogueur est un outil qui va vous aider à les traquer. Il permet d'exécuter le programme pas à pas, de voir son état à chaque instant, ainsi que sa consommation de mémoire, de modifier en direct son état, de vérifier si une portion de code est bien exécutée ou non, etc. En fonction du langage que vous choisirez, le débogueur sera déjà disponible ou non.

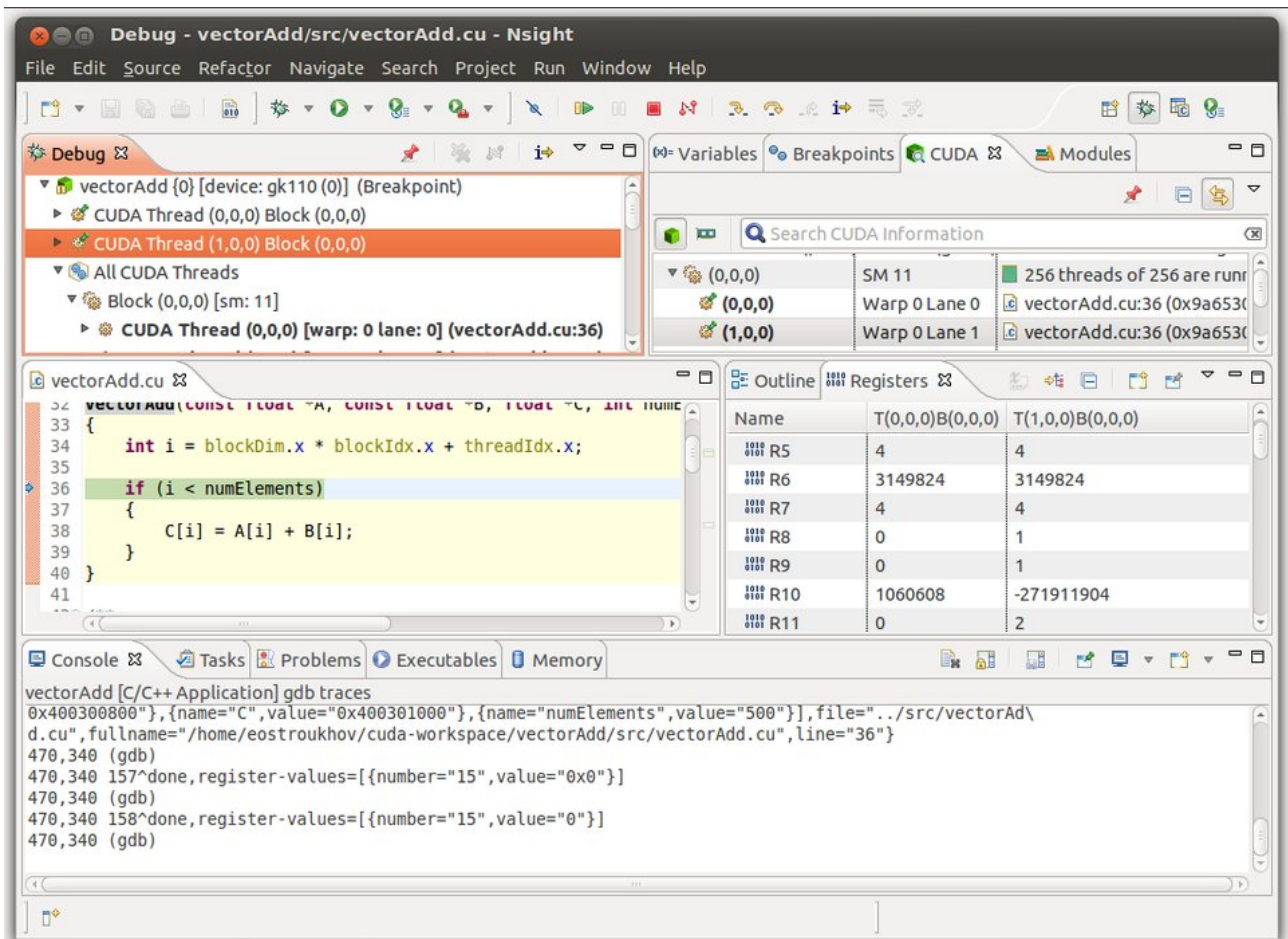




FIGURE 6. – Le débogueur [CUDA-gdb](#) de nVidia est une extension du débogueur gdb, lequel est présent dans toutes les distributions GNU/Linux, mais beaucoup moins *user friendly*.

6.2. Les IDE

Derrière ce sigle se cache le terme *Integrated Development Environment*, soit « environnement de développement intégré » en bon français. Un IDE est un regroupement de tous les outils que nous avons vus précédemment en un seul logiciel. Ainsi, dans la même fenêtre, on va pouvoir écrire le code puis, d'un clic de souris ou d'un raccourci clavier (au choix), lancer le programme puis le déboguer. Mais un bon IDE ne s'arrête pas là. Certains des plus puissants et des plus complets intègrent même un navigateur Internet (pour consulter de la documentation), ou des émulateurs (pour tester le code sur d'autres machines ou d'autres systèmes d'exploitation), et peuvent être complétés par des modules d'extension (*plugins*), etc.

Certains sont liés à un langage ou une plate-forme donnés, comme [Android Studio](#)  , qui est dédié au développement pour Android ; d'autres sont utilisables avec différents langages, comme [Eclipse](#)  . Certains sont payants, d'autres gratuits. Certains ont une interface minimaliste et austère, d'autres sont très complets. En bref, il y en a pour tous les goûts.

6.2.1. Que choisir ? Outils séparés ou IDE ?

C'est un grand débat qui fait rage parmi les développeurs. Chacun voit midi à sa porte. Certains vont dire que des outils séparés leur donnent plus de flexibilité, que les IDE ne sont pas assez configurables, qu'ils sont lourds et lents à démarrer, qu'ils ont pleins d'options en pagaille qui ne sont jamais utilisées, etc. D'autres vont dire que les IDE sont plus simples à utiliser, plus intuitifs, permettent au programmeur de moins se laisser distraire par des opérations et configurations manuelles pour mieux se concentrer sur le code, etc.

Je ne peux vous dire qu'une chose : faites votre propre choix.

6. Il y a programmer, et bien programmer

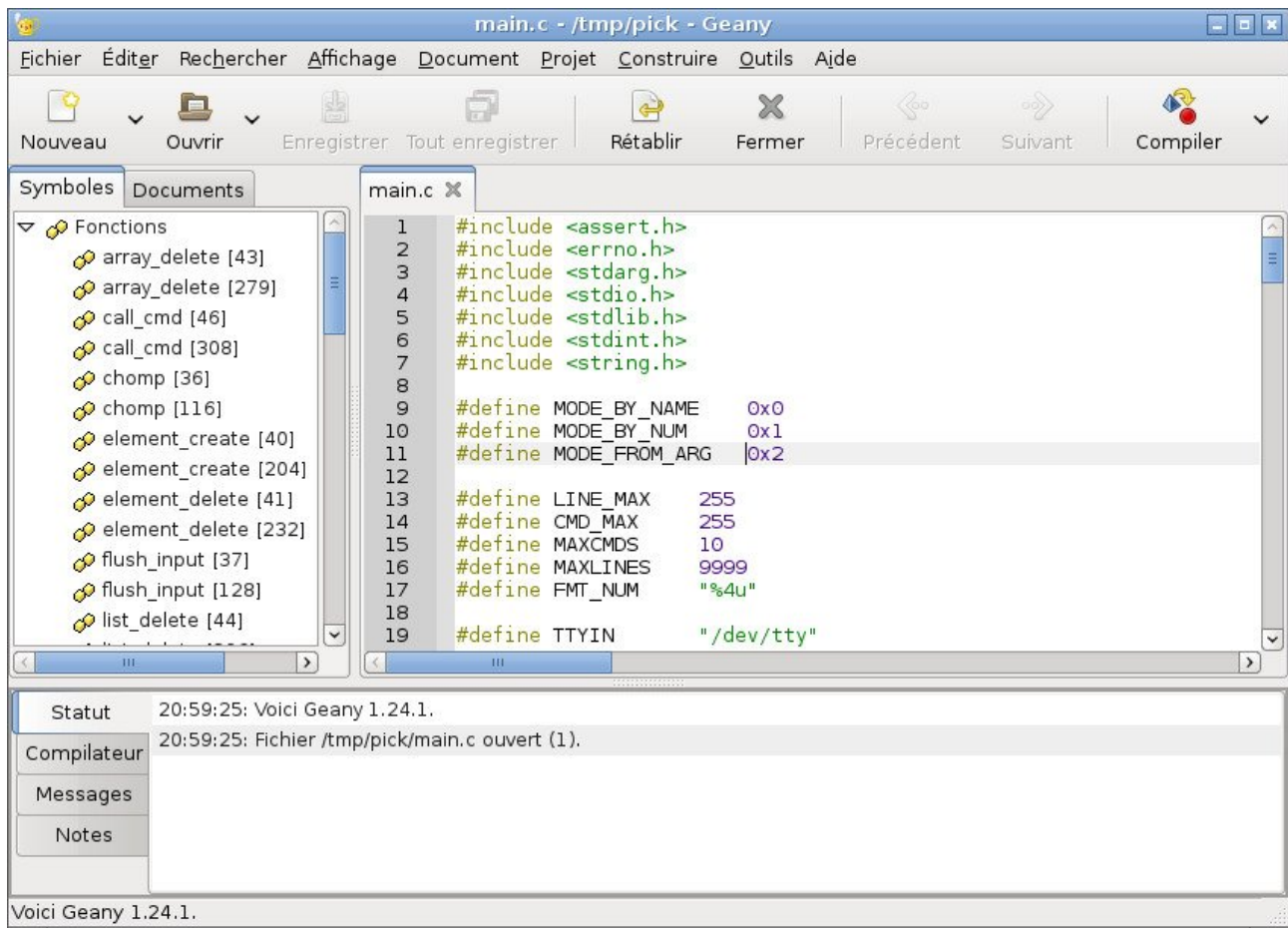



FIGURE 6. – Voici l’IDE généraliste [Geany](#) , conçu pour être relativement léger.

6.3. Quelques outils supplémentaires

6.3.1. L’algorithmique

Un bon programmeur ne se jette pas sur son clavier sauvagement : il prend le temps de réfléchir à ce qu’il va faire. Pour cela, il va d’abord concevoir des **algorithmes**. Ce sont en quelque sorte des recettes de cuisine informatiques, qui décrivent étape par étape la réalisation d’une action ou d’une commande. On trouve ainsi des algorithmes pour trier un ensemble d’éléments, trouver le chemin le plus court d’un point A à un point B, rechercher une information à travers Internet (Google, par exemple, applique un algorithme pour retrouver ce que vous avez demandé), etc. En bref, ils sont omniprésents en informatique.

Mais comment écrire un algorithme ? Et comment utiliser les bons algorithmes, les plus efficaces ? Je vous invite, en complément du langage de programmation que vous voudrez apprendre, à lire [L’algorithmique pour l’apprenti programmeur](#)  qui vous apprendra tout ce dont vous aurez besoin.


6.3.2. Les patrons de conception

Utiliser un patron de conception (ou *design pattern* en anglais) consiste à s'imposer de structurer son programme d'une certaine manière, ou à s'obliger à utiliser certains outils du langage préférentiellement à d'autres. Par exemple, les sites Web interactifs utilisent souvent le patron de conception appelé **MVC**. En très bref, cela consiste à séparer aussi hermétiquement que possible la communication avec la base de données (« modèle »), la production de la page qui sera affichée à l'utilisateur (« vue ») et le traitement des données renvoyées par celui-ci (« contrôleur »).

L'intérêt de la manœuvre ? D'une part, cela vous oblige à mieux concevoir les fonctionnalités de votre site Web, en vous forçant à les décortiquer. D'autre part, si vous décidez de changer d'outil de base de données, ou de transformer totalement l'apparence de votre site, ce modèle réduit drastiquement le risque que ces modifications introduisent des bogues dans le reste du fonctionnement.

De manière plus générale, un patron de conception est une solution longuement éprouvée et peaufinée à un problème donné. Elle est bien évidemment contraignante par certains aspects, mais quand elle est adaptée à votre situation (n'allez pas utiliser le **MVC** pour une bibliothèque de calcul distribué !), elle apporte généralement plus de bienfaits que de gêne.

6.3.3. Les structures logicielles

Une structure logicielle (ou *framework*) est un type particulier de bibliothèque logicielle. En effet, son but premier est d'encadrer votre programmation, en vous fournissant autant que possible tous les outils dont vous aurez besoin, afin que vous n'ayez pas à programmer en dehors d'elle¹³. Par exemple, [Django](#) ¹⁴ est un *framework* en Python destiné à faciliter la création de sites Web réactifs : il impose une structure, et propose des outils génériques dont tous les sites peuvent avoir besoin, tels qu'une interface d'administration, un espace membre et les services d'authentification qui vont avec, le moyen de traduire son site en plusieurs langues, etc.

Autre exemple, il existe plusieurs *frameworks* en JavaScript (jQuery ou angular.js, pour ne citer qu'eux) ayant un unique but : passer outre le fait que chaque navigateur n'en fait qu'à sa tête concernant ce langage, et que les mêmes actions doivent s'écrire différemment selon le navigateur utilisé par chaque visiteur de votre site Web. Pour ce faire, ils proposent une interface unique, et se chargent en coulisse de transformer cela en code compréhensible par chaque navigateur. Eh oui ! Comme la bibliothèque standard d'un langage !

Alors *framework* ou pas *framework* ? C'est là une autre des éternelles disputes entre développeurs. En pour, il y a le fait de pouvoir se consacrer plus à coder réellement votre logiciel, et moins à re-coder des outils déjà codés cent fois, et sans doute mieux que ce que vous seriez capable de faire. En contre, il y a l'obligation d'apprendre encore une couche d'abstraction, le fait qu'utiliser un *framework* peut s'apparenter à chasser la mouche au lance-roquette, et les limitations qu'induit toute structure.

Pour ma part, je vous dirai que coder avec ou sans *framework*, c'est comme faire confiance à un topo-guide ou préparer soi-même son parcours de randonnée.

13. Son but second étant de dominer le monde, mais n'est-ce pas là notre cas à tous ?

14. Attention, ce cours est un peu daté.



FIGURE 6. – Le site *Zeste de Savoir* utilise le *framework* Django pour son fonctionnement interne.

6.4. Les bonnes pratiques

Ce qui fait la différence entre un programmeur doué et un bon programmeur, c'est une question d'attitude : le fait de mettre en place un certain nombre de **bonnes pratiques**, qui vous faciliteront la vie, ainsi que celle des gens qui pourront interagir avec vos programmes. Vous pourrez trouver [ici](#) et [ici](#) deux explications en français sur ce que sont les bonnes pratiques en programmation.

Mais attention ! Vous remarquerez d'emblée que ces deux explications ne donnent pas les mêmes règles. Et c'est le gros défaut des « bonnes pratiques » : tout le monde n'est pas d'accord quant à la définition de celles-ci, quand elles ne sont pas carrément spécifiques à un langage donné. Globalement, elles se rangent en quatre grandes catégories.

Premièrement, **adopter plusieurs points de vue**. Cela passe, par exemple, par le fait de demander de l'aide quand vous rencontrez une difficulté, que ce soit sur Internet ou auprès de vos collègues expérimentés, de montrer son code à d'autres gens, quand il marche, des fois qu'ils auraient des idées pour le rendre encore meilleur, ou encore d'apprendre des langages fortement différents, afin de pouvoir plus facilement imaginer une manière originale d'aborder un problème.

Deuxièmement, **rendre son code réutilisable**, que ce soit par vous ou par d'autres. Cela passe par le fait (là encore, entre autres choses) d'utiliser des patrons de conception ou des techniques de programmation qui facilitent ce réemploi, d'user au maximum des possibilités de mise en forme du code source pour le rendre lisible, ou encore de documenter et de commenter son code.

Troisièmement, **adopter un flux de travail (*workflow*) efficace**. On parle là de tester souvent son code pendant le développement, tant qu'on peut encore savoir quelle modification a introduit une erreur donnée, d'utiliser un gestionnaire de versions (comme git, SVN, CVS, ou darcs), ou d'avoir une gestion claire des priorités dans les modifications à faire au code (par exemple, « la correction des bogues passe avant les évolutions »).

7. Et maintenant ?

Quatrièmement, **adopter des règles de codage** [↗](#). Il s'agit de faire des choix quant à la manière d'organiser et de nommer les différents fichiers du code source, de mettre en forme ce code source, de le commenter et le documenter, et de nommer ceux de ses éléments qui doivent avoir un nom. Autant être clair, si pour quelques rares langages il existe une convention obligatoire (par exemple la [PEP 8](#) [↗](#) en Python), il s'agit la plupart du temps d'une affaire de goûts, qui génère des débats sanglants et interminables entre développeurs. Par exemple, dans les deux documents fournis quelques paragraphes plus haut, on vous recommande de mettre tous les noms en anglais et de placer les accolades seules sur une ligne : ces deux pratiques vous vaudront les malédictions d'une partie des programmeurs. Je ne puis vous donner que deux conseils dans ce domaine.

1. Si vous devez participer à un projet où des règles de codage ont été adoptées, même de manière implicite, utilisez-les et non celles dont vous êtes coutumier/ère.
2. Ne vous mettez **jamais** dans la tête que les règles de codage que vous utilisez habituellement sont les meilleures : ce sont seulement celles que vous préférez, et vous vous éviterez des conflits en ne prétendant pas le contraire.



Si vous lisez l'anglais, le livre *The Pragmatic Programmer* [↗](#) peut vous offrir une première approche un peu plus détaillée de ces bonnes pratiques. Notez cependant que ce livre s'adresse plutôt à des gens qui se destinent au *métier* de programmeur qu'à ceux qui ne veulent le pratiquer qu'en passe-temps.

7. Et maintenant ?

Eh bien, l'étape suivante, c'est de choisir un premier langage de programmation à apprendre. Afin de vous orienter, laissez-moi vous présenter une petite liste des principaux langages qui pourraient vous intéresser, avec une explication succincte de leurs bons et mauvais côtés.



Avant de vous lancer dans la lecture, gardez à l'esprit que la qualité la plus importante pour un langage de programmation, c'est qu'il vous plaise à vous. Alors n'hésitez pas à en essayer plusieurs pour vous faire une idée !

Assembleur

Tout seigneur, tout honneur, l'assembleur *n'est pas* un langage pour débutant : sauf dans quelques situations très précises, et pas à la portée d'un débutant, il est toujours meilleur d'utiliser un autre langage. Si malgré tout vous êtes un(e) guerrier/ère de l'apocalypse, et que vous souhaitez vous frotter au père de tous les langages, ce langage vous enseignera beaucoup de choses cachées par les autres langages.

C

Le C est vraisemblablement¹⁵ le langage le plus utilisé au monde. Il faut dire qu'à l'exception de l'assembleur, c'est le langage qui vous laisse le contrôle le plus total sur votre machine : il est notamment très utilisé pour coder les systèmes d'exploitation. En outre, son grand âge (presque 45 ans, tout de même !) et sa popularité jamais démentie font qu'il dispose d'une

7. Et maintenant ?

quantité ahurissante de bibliothèques permettant de faire tout et n'importe quoi. La contrepartie à ce contrôle accru, c'est qu'il faut prendre soi-même en charge des choses que la plupart des langages gèrent « sous le capot », comme l'allocation de mémoire, et qu'il est donc beaucoup plus simple de tout casser. Zeste de Savoir est fier de vous offrir [un cours pour débutants](#) sur ce langage.

C++

Le C++ est le petit frère du C : il s'agissait à l'époque d'une simple extension de ce dernier, et sa syntaxe reste très similaire. Cependant, depuis les années 1980, il a bien grandi : le C++ moderne dispose d'outils très puissants pour utiliser la programmation orientée objet et même quelques bribes de programmation fonctionnelle. Le résultat est un langage très riche, et peut-être même trop riche : prétendre le maîtriser dans son intégralité est une gageure. Un cours est [disponible](#) sur Zeste de Savoir.

C#

Le C# est actuellement le langage par excellence pour programmer sous Windows et uniquement sous Windows. Il reprend un grand nombre de caractéristiques du Java (cf. plus bas), en tentant de minimiser ses principaux défauts. Mais surtout, c'est *le* langage de Microsoft : il est donc pleinement intégré dans l'environnement de développement par défaut de Windows, et constitue le moyen le plus simple d'accéder à toute la puissance du système d'exploitation. Mais par conséquent, et malgré de bons efforts pour essayer de le porter sous Linux, il reste essentiellement cantonné à Windows.

COBOL

Le COBOL est un vieux langage. Et cela se sent. Il est globalement plus difficile d'utilisation que les autres langages présentés ici (assembleur excepté, bien sûr). Cependant, pour des raisons historiques, il est encore très utilisé dans la banque, la finance et les assurances : pour cette raison, il a encore de beaux jours devant lui. Zeste de Savoir propose [un cours complet](#) sur ce langage.

Fortran

Avec le Fortran aussi, on fait de l'archéologie. Bien que sa syntaxe soit régulièrement renouvelée, on sent malgré tout sur lui le poids des années. Il reste cependant relativement vivace dans le domaine du calcul scientifique, celui pour lequel il a été conçu, à la fois à cause de spécificités dans sa conception qui n'ont guère été imitées, et pour des raisons historiques : de nombreux programmes de calculs lourds écrits dans les années 1980/1990 en Fortran sont encore en activité, et certaines bibliothèques logicielles en Fortran n'ont encore jamais été égalées en termes d'efficacité.

Haskell

Haskell est l'un des langages par excellence du paradigme fonctionnel. Ses qualités et défauts se résument globalement à celles du paradigme. En bien, il permet généralement d'écrire en quelques lignes ce qui en demande beaucoup plus dans d'autres langages, et il nécessite globalement moins de débogage. En mal, il est généralement difficile de l'apprendre après s'être habitué à un langage d'un autre paradigme, et certaines actions basiques (comme ouvrir un fichier ou afficher un texte à l'écran) sont plus difficiles à effectuer que dans les autres langages. Mais voici [le cours de référence](#) en français.

15. Il est à peu près impossible d'obtenir des chiffres exacts, mais toutes les estimations convergent vers ce résultat-là.

7. Et maintenant ?

Java

Le Java, comme on l'a vu, a la particularité d'être compilé en *bytecode*, lequel sera ensuite interprété par une machine virtuelle. Cela simplifie considérablement la création de programmes destinés à être utilisés sur plusieurs plate-formes et sous plusieurs systèmes d'exploitation. En particulier, Java est le passage obligé pour coder une application Android. Tout cela explique qu'il soit deuxième sur le podium des langages les plus utilisés. Pourtant, et bien qu'il y ait eu quelques améliorations dans la dernière version du langage, Java est réputé pour être très verbeux. Voici pour vous le [cours de ZdS](#) sur le sujet.

JavaScript

Le JavaScript est un langage controversé. Bien qu'ayant des qualités, il est réputé pour avoir de nombreux comportements aberrants, et pour implémenter une forme assez étrange de programmation orientée objet : globalement, il peut être difficile de l'utiliser sans un *framework* adéquat. Seulement, à l'heure actuelle, JavaScript est le seul et unique langage disponible dans tous les navigateurs Web¹⁶, et il est donc indispensable de le connaître pour faire un site Web interactif.

Perl

Le Perl est un langage principalement apprécié dans le monde de Linux et des Unixoides. En effet, il est principalement efficace pour créer de petites applications très puissantes en ligne de commande, et n'est pas très adapté pour créer des interfaces graphiques. En outre, ses possibilités d'expressivité sont telles que la phrase « Il y a plus d'une façon de le faire. » est devenu un mantra parmi ses utilisateurs. Mais elles constituent également son point faible : adepte des formulations très concises et doté d'une syntaxe parfois déroutante, un programme en Perl peut être assez dur à lire par quelqu'un qui n'en est pas l'auteur.

PHP

Le PHP domine largement le monde de la programmation Web : si le JavaScript est seul maître à bord côté client, le PHP, accompagné de son compagnon obligé SQL pour les bases de données, n'a que peu de concurrents côté serveur. Cette place de choix, l'importante communauté de ses programmeurs et sa documentation plantureuse en font un langage volontiers choisi par les débutants. Il faut bien cela pour compenser le fait qu'il soit globalement assez lent (mais la nouvelle version pourrait améliorer les choses), que sa bibliothèque standard manque cruellement de cohérence et qu'il soit facile de prendre de mauvaises habitudes de programmation. Un cours en bêta sur ce site n'attend plus que sa validation.

Python

Le Python est volontiers conseillé aux débutants, car il est — de l'avis général — facile à prendre en main. Doté de très nombreuses bibliothèques, conçu pour fonctionner main dans la main avec d'autres langages, et pourvu de multiples fonctionnalités empruntées à plusieurs paradigmes, il n'y a pas grand chose que l'on ne puisse pas coder assez simplement en Python. Cependant, cette facilité de prise en main a un prix : la richesse de la syntaxe. Là où Perl érige en principe de vie qu'il y a plusieurs manières d'écrire un même programme, Python s'efforce au contraire à ce qu'il n'y en ait qu'une : un bon code en Python ne vous laisse même pas choisir comment le mettre en forme ! En outre, il est souvent assez lent comparé à des langages comme C++ ou Haskell. Le [cours d'introduction à Python](#) de Zeste de savoir est à votre disposition.

16. Au détail près que chaque navigateur implémente sa propre version du langage : bien que cela porte la plupart du temps sur des détails du langage, c'est un des reproches les plus courants à l'égard de ce langage.

Ruby

Le Ruby est directement inspiré du Python, qui en retour lui emprunte régulièrement des innovations, et globalement, ils se ressemblent beaucoup. À tel point que savoir lequel des deux est le meilleur est un des trolls les plus vivaces de l'Internet. On pourrait dire, pour tracer à grands traits, que Ruby offre plus de liberté syntaxique et insiste plus sur son caractère orienté objet, alors que Python est d'un abord plus facile et supporté par une communauté plus importante. Et pour votre plus grand plaisir, Zeste de Savoir offre un [cours complet](#) sur ce langage.

Swift


Le Swift est un langage assez jeune, et encore sujet à des changements et ajustements. Il est cependant destiné à remplacer Objective-C comme langage par excellence des produits Apple, à la manière de C# sous Windows : en particulier, dans les années à venir, il pourrait bien devenir l'incontournable de la programmation d'applications sur iOS et OS X. Pour le reste, il s'agit d'un langage assez classique et proche du C dans sa syntaxe.

Et pour quelques langages de plus...

- **ActionScript** est le langage derrière les applications Flash encore courantes sur Internet : il peut être un moyen simple de créer un programme destiné au Web, mais la technologie est de plus en plus décriée, et ce langage n'est pas nécessairement un choix d'avenir. Vous pourrez cependant l'apprendre dans [ce cours](#).
- ZdS propose un [cours complet](#) sur **Ada**, un langage assez similaire au C dans ses possibilités, mais avec une syntaxe plus verbeuse et une attention accrue portée à la fiabilité des programmes : il fait partie, au même titre que le **Pascal**, de ces langages qui n'étaient pas mauvais en soi, mais qui pour diverses raisons historiques ont périclité au profit de certains de leurs concurrents.
- **Erlang** est un langage fonctionnel avec quelques structures tirées des langages impératifs, spécifiquement conçu pour faire de la programmation concurrente. En revanche, il est à proscrire si vous devez traiter beaucoup de texte. Le meilleur cours à ce jour est [celui-ci](#), malheureusement en anglais.
- **OCaml** est également un langage fonctionnel, plus classique que le Erlang. Il est souvent enseigné en France dans les prépas scientifiques ou les écoles d'ingénieur, sans doute en raison de son origine française. Si vous souhaitez l'apprendre, [ce site](#) fera votre bonheur.
- **Prolog** est le principal représentant de la programmation logique. À ce titre, il est volontiers utilisé dans la réalisation d'intelligences artificielles, bien qu'il soit pour le reste très méconnu.
- **Rust** est un langage assez jeune, porté par Mozilla. Son objectif à terme est de prendre la succession du C. Pour cela, il s'efforce de laisser autant de liberté que ce dernier dans la manipulation directe de la machine, tout en ajoutant des barrières solides contre le risque de tout casser.
- **Scheme** est un dialecte épuré de la grande famille du Lisp, une sorte d'OVNI de la programmation, reconnaissable entre mille par son usage *massif* des parenthèses. Il peut être intéressant à étudier en raison de son fonctionnement particulier et de son extrême flexibilité, et il est couramment utilisé comme langage de script dans divers logiciels, permettant ainsi de leur ajouter quelques fonctionnalités à la volée.

8. Conclusion

Cette liste des principaux langages ne vous aura peut-être pas permis d'arrêter un choix. Alors prenons le problème sous un autre angle, celui de l'utilisation que vous désirez faire de la programmation.

Pour programmer des **applications Web**, vous devrez apprendre deux langages (notez qu'il commence à être possible d'utiliser le même langage des deux côtés). Pour le côté client, JavaScript est à peu près inévitable, sauf à utiliser un langage qui génère du JavaScript, comme CoffeeScript ou Elm. Pour le côté serveur, PHP tient le haut du pavé, mais Python et Ruby se défendent bien. Java fait également partie des langages très populaires (un tutoriel est d'ailleurs [disponible](#) ) , ainsi que C#. Et depuis quelques temps, JavaScript est également utilisable côté serveur (grâce à NodeJS notamment).

Pour du **jeu vidéo** (hors ligne, sinon cf. plus haut), C++, Python et C# sous Windows sont sous les feux de la rampe. Ils sont cependant loin d'être les seuls, et tout langage qui permet assez facilement de produire une interface graphique peut convenir, tel que C, Java, Ruby ou encore Tcl/Tk.

Pour de **grosses applications utilitaires**, le marché est dominé par C++ et Java, bien que C# gagne du terrain sous Windows. Au contraire, pour de **petites applications utilitaires**, en particulier en ligne de commande, on trouvera volontiers C, Perl, Python ou Ruby.

Certains langages sont plus spécifiquement dédiés à l'une ou l'autre plate-forme. Ainsi, Java est indispensable sous Android, et l'on peut se faciliter la vie en utilisant C# sous Windows et Swift ou Objective-C sous iOS et OS X.

La **programmation système** impose des contraintes particulières, et C reste le maître incontesté de ce domaine. Il peut cependant parfois être utile de regarder du côté de l'assembleur ou de Ada.

Dans le domaine du **calcul scientifique**, Fortran reste le roi, mais un roi vieillissant. Il est de plus en plus concurrencé par C++, Python, ou des langages spécialisés comme Matlab et R, voire par les langages fonctionnels en général (qui dominent totalement le domaine de la preuve formelle).





Enfin, de très nombreux langages permettent de faire de la **programmation concurrente**, c'est-à-dire des programmes qui font travailler ensemble de très nombreux ordinateurs ou des ordinateurs ayant de très nombreux processeurs. Mais certains langages sont plus spécifiquement dédiés à ce type de tâches, Erlang venant en tête, suivi de Ada et Rust.

Notez bien que cette classification, faute d'exhaustivité, passe à côté de certains langages réputés, comme Haskell, et qu'une question plus précise posée sur les forums permettra peut-être de vous orienter vers un langage plus spécifiquement dédié à la tâche que vous voulez accomplir.



Ce cours a été rédigé par les auteurs listés en haut de page, mais il a également été préparé, lu, relu et approuvé par une fine équipe de rédacteurs de cours de programmation, j'ai nommé @imperio, @Karnaj, @Lalla, @mehdidou99 et @Taurre, avec le soutien moral de @paolo10 et @Rod. Merci à eux.

8. Conclusion

Ce cours est placé sous licence [BiPu L](#)  . Le logo est l'œuvre de Dominus Carnufex et est placé sous la même licence. Il a été réalisé à partir de [cette image](#)  de valco, et [cette image](#)  de Christopher Kuszajewski, toutes deux placées sous licence [CC 0](#)  . Les captures d'écran de logiciels sont utilisées au titre du droit à la citation, et restent sous la propriété intellectuelle de leurs auteurs respectifs.

Liste des abréviations

MVC Modèle Vue Contrôleur. 20